Advanced Programming Languages

Authors

Tibor Ásványi
Ákos Balaskó
Iván József Balázs
Balázs Csizmazia
Péter Csontos
Szabina Fodor
Attila Góbi
Hajnalka Hegedűs†
Zoltán Horváth
András Juhász
Attila Kispitye
Tamás Kozsik
Lehel István Kovács D.
Richárd Legéndi
Tamás Marcinkovics
Attila Rajmund Nohl
Judit Nyéky-Gaizler
Gábor Páli
Zoltán Porkoláb
Gábor Pécsy
Máté Tejfel
Szabolcs Sergyán
Balázs Zaicsek
Viktória Zsók

Advanced Programming Languages

Editor
Judit Nyéky-Gaizler

The book is supported by the TÁMOP-4.1.2.A/11-1/1 project.

Editor:
Judit Nyéky-Gaizler

Layout editor: Attila Kispitye
Publisher: Eötvös Loránd University
ISBN: 978-963-284-450-3

Contents at a Glance

Table of Contents

Introduction

Programming languages are thought by many to provide as a notation form for program description. This view does not take into account – or does not even know –, how *high level* or *user-centered languages* can aid in managing program complexity. Different languages with their possibilities suggest different programming approaches, so the common practice, which is still used nowadays in many places, is highly dangerous, when programming methodology is taught through particular programming languages, not independently from them – this could only lead to narrow concerning all the programming possibilities.

The goal of programming is to produce a *good quality software product*, so the education of programming must start with the general definition of the task and its solving program [Fot83]. Then based on this principle, the different concrete language tools should be acquainted to the programmers, which support the implementation. However, as it is questionable to teach the methodology through particular concrete programming languages, it also leads to a dead end, if the used programming language is said to be not important for the sake of the methodology. This is – as described by Bertrand Meyer [Mey00] – like "a bird without wings". The idea is inseparable from the possibilities of formulation. It is not a coincidence that in programming no single language has become dominant, nor that always newer programming languages are designed, which support even more the adaptation of different methodological concepts and requirements into practice.

Designers of programming languages must deal with three problems [Hor94]:

- The representation provided by the language must fit the hardware and the software at the same time.
- The language must provide a good nomenclature for the description of algorithms.
- The language must serve as a tool to manage program complexity.

Aspects of software quality

The software is a product, and as for every product, it has – as defined by many ([Mey00], [Hor94] and [LG96]) – different quality characteristics and requirements. One of the most important goals of the programming methodology is to specify a theoretical approach for creating good quality program products. The design and the evaluation of already existing programming languages are definitely influenced by methodological considerations.

Next, characteristics of "good" software will be discussed according to the work of Bertrand Meyer [Mey00]. After that, language features will be examined for supporting the methodology – through numerous programming languages.

Software quality is influenced by many factors. One part of these – such as reliability, speed, or ease of use – are basically perceived by the user of the program. Others – such as how easy it is to reuse some parts of it for a different, but similar problem – affect program developers.

Correctness

Correctness of the program product means that the program solves exactly the problem and fits the desired specification. This is the first and most important criterion, since if a program is not working like it should, other requirements do not really count. The elementary basis for this is the precise and the most complete specification.

Reliability

A program is called *reliable* if it is correct, and abnormal – not described in the specification – circumstances do not lead to catastrophe, but are handled in some "reasonable" way.

This definition shows, that reliability is by far not as a precise notion as correctness. One could say, of course with a more specific specification reliability would mean correctness exactly, but in practice there are always cases which are not covered by specification explicitly. That is why reliability is of high priority for the program product quality.

Maintainability

Maintainability refers to how easy it is to adjust the program product to specification changes.

The users often demand further development, modification, adjustment of the program product to new external conditions. According to some surveys 70 % of program product costs are spent on maintenance, so it is understandable that this requirement significantly affects the quality of the program. (This is

relevant especially if developing big programs and program systems, since for small programs usually no change is too complex.)

To increase maintainability, design simplicity and decentralization (to have independent modules) can be seen as the two most important basic principles.

Reusability

Reusability is the feature of the software products, that they can be partly or as a whole reused in new applications.

This is different to maintainability, since the same specification was modified there, but now the experience should be utilized, that many elements of software systems follow common patterns, and reimplementing already solved problems should be avoided.

This question is particularly important, not only when producing individual program products, but for a global optimization of software development, as the more reusable components are available to help problem solving, the more energy remains to improve other quality characteristics (at the same costs).

Compatibility

Compatibility shows how easy it is to combine the software products with each other. Programs are not developed isolated, so efficiency can go up by orders of magnitude, if ready software can be simply connected to other systems. (Communication between programs is based on some standards, such as, for example, in Unix.)

Other characteristics

From the quality characteristics of the program product, portability, efficiency, user friendliness, testability, clarity etc. are also important to pay attention to.

Portability regards how easy it is to port the program to another machine, configuration or operating system – usually to have it run in different runtime environments.

The *efficiency* of a program is proportional to the running time and used memory size – the faster, or the less memory is used, the more efficient it is. (These requirements often contradict each other, a faster run is often set off by bigger memory requirements, and vice versa.)

The *user friendliness* is very important for the user: this requires data input to be logical and simple, the output of the results must be clearly formatted.

Testability and *clarity* are important for the developers and maintainers of the program, without these the reliability of the program cannot be guaranteed.

Aspects of software design

Some of these requirements – the improvement of correctness and reliability – require primarily the development of specification tools. The easier it is to verify if a piece of program code is really an implementation according to the specification, the easier it will be to developed correct and reliable programs. The main role here have programming language features for specification (type invariant, pre- and postconditions) descriptions – this is supported for example by Eiffel [Mey00], by Ada 2012 [Nyek98] etc.

Implementation of another group of requirements – mainly maintainability, reusability and compatibility – can be best supported by designing the programs as independent program units having well defined interconnections. This is the basis of the so called *modular design*. (A module here is not a programming language concept, but a unit of the design.) This question will be handled in more detail in Chapter 9.3.

Our goal is to examine the features of different programming languages to support professional programmers in developing reliable software of good quality.

Study of the tools of programming languages

It is a natural question, why it is not enough to know *one* programming language, for what purpose it is good to deal with all the possible features of different programming languages. In the following – primarily based on the work of Robert W. Sebesta [Seb13] – we will try to summarize the advantages coming from this:

Increase of the expressive power

Our thinking and even abstraction skills are strongly influenced by the possibilities of the language used. Only that can be expressed, for which there are words. Likewise during program development and designing the solution, the knowledge of diverse programming language features can help programmers to widen their horizon. This is also true if a particular language must be used, since good principles can be applied in any environments.

Choosing the appropriate programming language

Many programmers have learnt programming through one or two languages. Others know older languages which are now considered obsolete, and they are not familiar with the features of modern languages. This could result in not selecting the most appropriate language if there would be more programming languages as options to choose from for a new task – since they do not know the possibilities the other languages could offer. If these programmers would know

the unique features of the available tools, they could make considerably better decisions.

Better attainment of new tools

Newer and newer programming languages will appear, thus quality programming requires continuous learning. The more the basic elements of the programming languages are known, the easier it will be to learn and keep up with progress.

In our book most examples are in Ada, C/C++ or Java language for certain language constructs, there are only a few chapters (except of course those about logical and functional programming) where these languages are not referenced in almost every paragraph.

Our book is aimed at facilitating primarily, the studies of university and college students to learn about programming languages, and to help the work of IT and computer specialists. Some degree of knowledge of informatics is a prerequisite to fully understand our book: readers must have already solved some programming tasks on some programming languages.

Acknowledgements

The authors wish to thank for the support of TÁMOP tender on developing teaching materials.

We also thank Zoltán Horváth, the dean of the Faculty of Informatics at the Eötvös Loránd University for permitting the usage of the infrastructure of the Faculty of Informatics. Without his kind contribution this work could not have been completed.

We would like to thank the generous assistance of the PhD students who helped us with their feedback to improve this new edition of the book.

In such a voluminous book – despite all the best efforts of the authors and the editor – there could be errors. We would like to ask You, dear reader, if such an error is found, please notify us via email addressed to *proglang@inf.elte.hu*. We also welcome every kind of constructive criticism.

The current version of the whole book can be found as a downloadable pdf at:
`http://nyelvek.inf.elte.hu/APL`

1 Language Design

In this chapter, we provide a general overview of the concepts of programming language design (such as syntax, semantics and pragmatics) and discuss the various implementation options (compiler, interpreter, etc.). We then discuss the evolution of programming languages. We identify the features of a good programming language. We also examine the consequences of the dramatic increase in the number of novel programming languages, how that explosion has affected the principles of computer programming and what historical and methodological categories the large number of languages can be grouped into. Finally, we analyze how external factors, such as programming and communication environments, have shaped the development of programming languages.

There are thousands of high-level programming languages, and new ones continue to emerge. However, most programmers only use a handful of those languages during their work. Then why are there so many languages? There are several possible answers to that question:

- **Evolution of programming paradigms**. Programming languages and the principles behind them are being continuously improved. The late 1960s and the early 1970s saw the revolution in "structured programming". In the late 1980s the nested block structure of languages such as Pascal began to give way to the object-oriented structure of C++ and Eiffel.
- **Different problem domains**. Many languages were specifically designed for a special problem domain. For example, LISP works well for manipulating symbolic data and complex data structures. Prolog is suitable for reasoning about logical relationships between data sets. Most of the programming languages can be used successfully for a wide range of tasks, but some of them are better than others in solving specific problems.
- **Personal preferences**. Different people like different things. Some people like to work with pointers, others prefer the implicit dereferencing of Java, ML, or LISP.
- **Expressive power**. The expressive power of a language is the spectrum of ideas that can be expressed using the given language. Though this could, in theory, be an important basis for comparison, the majority of languages are all suitable for implementing any algorithm (a feature closely related to Turing completeness). Therefore, the expressive power of the various languages is mostly equivalent.
- **Easy to learn**. The success of Basic was in part due to its simplicity. Pascal was taught for many years as an introductory language because it was very easy to learn.
- **Ease of implementation**. Basic became successful not only because it was easy to learn but also because it could easily be implemented on smaller machines with limited resources.

- **Standardization**. Almost every language in use has an official international standard, or a canonical implementation. Standardization of the language is an effective way of ensuring the portability of the code across different platforms.
- **Open source**. Many programming languages have open source compilers or interpreters, but some languages are more closely associated with freely distributed, peer-reviewed, community-supported computing than others.
- **Excellent compilers**. Fortran owes much of its success to extremely good compilers. Some other languages (e.g. Common Lisp) are successful, at least in part, because they have compilers and supporting tools that effectively help programmers.
- **Patronage**. Technical features are not the only relevant factors, though. Cobol and Ada owe their existence to the U.S. Department of Defense (DoD): Ada contains a wealth of excellent features and ideas, but the sheer complexity of implementation would have killed it without the DoD backing. Similarly, C# probably would not have received the same attention without the backing of Microsoft.

Clearly no single factor determines whether a language is ”good” or ”bad”. Therefore, the study and assessment of programming languages requires a careful look at a number of issues [Sco09].

1.1 Programming languages: syntax, semantics, and pragmatics

Programming languages are artificial formalisms designed to express algorithms in a clearly defined and unambiguous form. Despite their artificial nature, they nevertheless fully conform to the criteria of a language. Programming languages are structured around several descriptional/structural levels [Hor94]. Three such levels discussed below are syntax, semantics and pragmatics [GM10].

- **Syntax** describes the correct grammar of the language, i.e. how to formulate a grammatically correct phrase in the language.
- **Semantics** defines the meaning of a syntactically correct phrase, i.e. it gives meaning and significance to each phrase of the language.
- **Pragmatics** determines the usefulness of a meaningful phrase of the language, i.e. it defines how to use the given phase for a useful purpose within the program.

The three structural levels can be illustrated in the assignment *let year = 2013*. At the syntax level, the question is whether this formula is grammatically correct (let us assume that it is). At the level of semantics, the question is what this phrase means (in this case, the meaning is that the value of the variable *year* is set to 2013). At the level of pragmatics, the question is what this assignment

is used for (e.g. to calculate, by using another formula, the remaining value of a mortgage at the end of year 2013).

As programming languages are bona fide languages, their structural levels are very similar to those of natural languages. Indeed, a novel written in a natural language can be analogous to a program written in a programming language. At the syntax level, "fishes swim in the ocean" and "suitcases drive pine trees" are both correct. Yet, at the level of semantics, the latter one is wrong due to the lack of an appropriate meaning. At the level of pragmatics, the former sentence would make sense as part of a story on a little mermaid but it would most likely not fit into a technical guide on how to survive for a week in the Saharan desert.

In technical terms, syntax defines how programs are written and read by programmers, and parsed by computers. Semantics determines how programs are composed and understood by programmers, and interpreted by computers. Finally, pragmatics guides programmers in how to design and implement programs in real life [Wat06].

In the following sections, we will discuss each of the above structural levels of programming languages in more detail.

1.1.1 Syntax

As mentioned above, *syntax* in principle corresponds to the grammatical rules of the language. Like natural languages, programming languages are also sets of characters (symbols) of a predefined alphabet. At the lowest level, *syntax* requires definitions of the sequences of characters that constitute the smallest logical units (words or tokens) of the language. Once the alphabet and the words have been defined, *syntax* describes which sequences of words constitute legitimate phrases, the smallest meaningful units of the language. At a higher syntactic level, strings of those phrases combine into sentences or statements, which are then again combined into program modules or entire programs.

The syntactic rules of a language specify which strings of characters are valid, i.e. grammatically correct. The theoretical basis of syntactic descriptions date back to the mid 20th century. In the 1950s, the American linguist Noam Chomsky developed techniques to describe syntactic phenomena in natural languages in a formal manner. Though his descriptions originally used formalisms designed to limit the ambiguity present in natural languages, this formalism also applies to the syntax of artificial languages, such as programming languages [GM10]. Shortly after Chomsky's work on language classes and structures, the ACM-GAMM group begun designing ALGOL 58, one of the early programming languages. John Backus, a prominent member of this group introduced a new formal notation for specifying programming language syntax. This notation was then modified by Peter Naur; this revised method of syntax description is now known as the Backus-Naur form or *BNF*. Though the development of BNF occurred independently from Chomsky's work, it is remarkable that the basic

principles of BNF are very similar to those of one of Chomsky's language classes, the so-called context-free languages [Seb13].

For the different lexical elements of programming languages, see Chapter 2.

1.1.2 Semantics

While syntax only concerns itself with the appropriate format of the language, semantics is a higher level feature that deals with the meaning and significance of the given phrase [GM10]. The meaning of a phrase can be very diverse, such as a mathematical function, a relationship between program components, or an exchange of information between the different parts of the program and the environment, etc. The semantics of the programming language describes what processes the computer will follow during the execution of the program. The description of the semantics of a programming language is more complex than its syntactic description. This complexity is caused by technical problems in describing abstract features, as well as by the need to balance between the opposing requirements for exactness and flexibility of implementation [GM10]. Indeed, it is relatively easy to design exact semantics if only one route of implementation is expected. However, as soon as the implementation platform changes, additional questions arise which further complicate the semantic definition. It is also relatively difficult to describe semantic issues in computer language. Therefore, most semantic definitions are provided in natural languages and are then implemented/translated to a computer language [GM10].

1.1.3 Pragmatics

Semantics defines whether a given phrase is meaningful, i.e. whether it can be interpreted and executed, but it does not tell whether the phrase is used for any purpose. The level of pragmatics ensures that the program composed of meaningful phrases makes sense and that it is indeed a useful tool for a given purpose [GM10]. The precise description of the pragmatics of a programming language is difficult, if at all possible. This is in part due to the highly abstract nature of pragmatics. In addition, pragmatics deals with the purpose or use of a syntactically and semantically correct phrase. While both syntax and semantics may be clearly defined and unambiguously understood, the same phrase may be used for a number of different purposes, and its uses may change during the use of the language. Therefore, no single definition of the pragmatics of a given phrase is possible. One component of pragmatics is programming style. While it is relatively easy to clearly describe some programming style issues (such as the avoidance of jumps or gotos), others are more of vague guidance than clear instructions. Undoubtedly, pragmatics is an integral part of the concept of programming languages and it strongly affects the usefulness of a given programming language for a particular purpose.

In this book, we will discuss semantic and pragmatic issues in detail, while little emphasis will be placed on the syntax of the various programming languages. Readers interested in syntactic issues of a given language are referred to the vast literature on the technical details of the different programming languages.

1.2 Implementation of computer programs

Besides the above issues of syntax, semantics and pragmatics, the overall performance of a computer program also strongly depends on how the program is implemented on a given run-time environment. This *implementation level* therefore is added on top of the above three levels. A program written in a given programming language can eventually be implemented using several separate and even conceptually different implementation approaches. Nevertheless, most programming languages are designed for a given implementation strategy and there is little communication between the different strategies in the case of a given language. Though implementation is in most cases beyond the programmer's scope and perspective, the actual implementation may strongly influence the eventual efficiency of the program, and thus the possible ways of implementation may also determine the choice of the most suitable programming language. Here we will outline the different strategies for the automated translation and implementation of programs developed using higher-level programming languages. The most widely used implementation strategies use one of the following methods:

- compiler implementation;
- pure interpretation;
- hybrid implementation systems.

Compiler implementation

In the case of *compiler implementation*, the program is first translated to machine language to generate a code that can later be executed directly on a computer. The original program code is called the source code while its language is known as the source language. The resulting machine-executable code is the object code and its language is the object language. The translation of the source code to machine language is called compilation, which is completely separated from the execution of the program. This approach has several advantages, mainly in large-scale industrial program development. Given that no re-translation of the source code is required, the execution of the program is very fast. Another advantage is that the final executable program can be distributed without distributing the source code, thus providing protection for the programmer's intellectual property rights. Disadvantages, on the other hand, include the compilation process itself which is rather slow. The program needs to be re-compiled every time the source code is altered, and there is a limited number of opportunities for checking and correcting the code. However, the widely available professional code writing,

compilation and de-bugging tools make the compiler implementation approach a very viable strategy overall.

The process of compilation takes place in several phases, the most important of which are shown in Figure 1.1 and discussed in the next paragraphs.

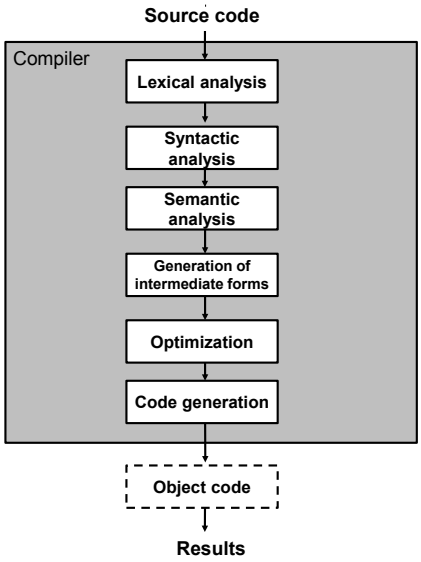


Figure 1.1: Organisation of a compiler

Lexical analysis - The aim of lexical analysis is to read the program text and to group the characters (symbols) into meaningful logical units called tokens. The input text of the source program is scanned in a sequential manner, taking a single pass to recognize tokens. No further analysis of whether, for instance, the separators or the number of attributes are correct, is performed at this point.

Syntactic analysis - Once the list of tokens has been constructed, the syntactic analyzer (or parser) attempts to construct a derivation tree (or parse tree), a structured composition of the input string (the source code), in line with the grammatical restrictions of the language. At the end of syntactic analysis, each unit (leaf) of the derivation tree has to form a correct phrase in the given language.

Semantic analysis - The derivation tree, which is a structured representation of the input string, is subject to checks of the language's various context-based constraints. It is at this stage that declarations, types, number of function parameters, etc., are processed. As these checks are performed, the derivation tree is complemented with the relevant additional information and new structural complexities are generated.

Generation of intermediate forms - In this phase, an initial intermediate code is generated from the derivation tree. This intermediate code is not yet in the

object langage since a substantial amount of code optimization – independent of the object language – has to be performed, and this optimization can best be done without restrictions of the object language.

Code optimization - The code obtained in the first translation attempt is usually inefficient. Therefore, several steps of optimization need to be performed at this phase. This includes removal of the redundant code, optimization of loop structures, etc. All this optimization precedes the generation of the object code.

Generation of object code - Once an optimized intermediate code has been generated, it has to be translated to the object language to obtain the final object code. This will be a machine-readable code that will be directly executed by the computer. An important part of the object code generation is the register assignment.

Pure interpretation

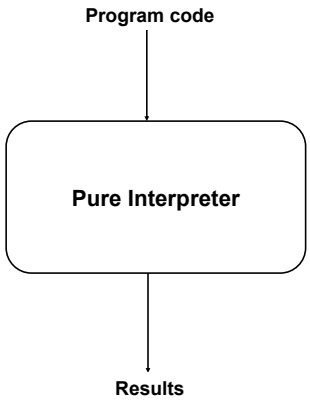


Figure 1.2: Pure interpretation

A conceptually different approach from compiler implementation is that the program is interpreted by another program, called an interpreter, every time the program is executed. This interpretation occurs parallel to the execution itself, and thus no separate translation of the entire program to machine code is performed, and no executable machine code is generated. In principle, the interpreter simulates a machine that is capable of dealing with high-level programming languages and statements rather than with low-level machine code only. Since such a machine does not physically exist (it is only simulated by the interpreter), the execution environment generated by the interpreter is often called a *virtual machine*. The advantage of this approach is that it makes the execution and optimization of the program code relatively easy. In particular, the de-bugging of programs in pure interpretation languages is straightforward since run-time error messages can directly be connected to the units of the original

program code. On the other hand, the pure interpreter approach is not quite suitable for large-scale industrial development of highly complex and structured programs due to the time consuming nature of the interpretation of the entire program code at every instance of program execution.

For the functioning of an interpreter, see Figure 1.2.

Hybrid implementation systems

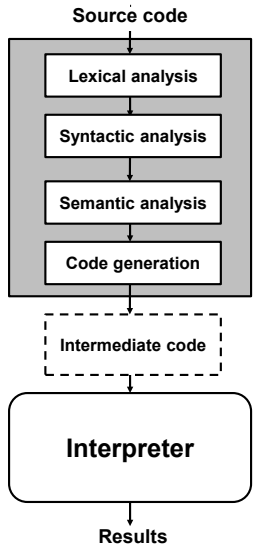


Figure 1.3: Hybrid implementation system

As described above, compiler implementation allows the fastest execution of the program but its compilation phase is time consuming and de-bugging is more difficult; on the other hand, pure interpretation allows immediate execution (without delay of compilation) and de-bugging is fast and straightforward, but ultimately the execution of the program is slow. Some language implementation systems combine the two approaches so as to exploit the advantages of both the compiler and the interpreter systems. In such cases, the high-level language is translated (partially compiled) to an intermediate level code which is then executed by an interpreter of the intermediate code (the virtual machine). The language of the intermediate code is designed in such a way that it allows very fast interpretation for machine execution. As a result, the source code is translated only once in a faster manner than in the case of compiler implementation, and the resulting intermediate code is executed rapidly by the intermediate code interpreter (the virtual machine). A classical example of such a hybrid system is Java which first translates the Java source code to an intermediate

code called *byte code*, which is then executed by an interpreter approach using the Java Virtual Machine. A similar system is used by Perl, another hybrid implementation system. An additional advantage of such systems is that the intermediate code (e.g. Java byte code) is independent of the execution platform and can be run on virtual machines (e.g. Java Virtual Machines) implemented on any operating system. In addition, the intermediate code is different from the source code, and therefore, it can be distributed without compromising the intellectual property linked to the source code.

The process used in a hybrid implementation system is shown in Figure 1.3. After the discussion of the various features of program design and implementation, we next describe the emergence of the programming languages from a historical and evolutionary perspective.

1.3 The evolution of programming languages

1.3.1 The early years

Thousands of programming languages have been developed over the last 50 years, but only the ones with the best features have received wider recognition. Every language is judged on the basis of its features. Initially, languages were developed for specific purposes which limited their scope of use. However, as computer use became widespread, the languages had to be adjusted to cater for many different interests and needs.

Though building computing machines dates back to the ancient Greeks, the first "true" computer program was written for the Analytical Engine by a mathematician called Ada Lovelace to calculate a sequence of Bernoulli Numbers. A number of factors, including the synthesis of numerical calculation, predetermined operation and output, and ways to organize and input instructions in a manner relatively easy for humans to conceive and produce, led to the development of modern computer programming [Sam69].

The first high-level programming language, the Plankalkül was designed by Konrad Zuse [Zus72] for engineering purposes between 1943 and 1945. Plankalkül has shared many features with APL, a programming language to be developed later (named after the book *A Programming Language*), and relational algebra. It included assignment statements, subroutines, conditional statements, iteration, floating point arithmetic, arrays, hierarchical record structures, assertions, exception handling, and other advanced features such as goal-directed execution.

The example below shows a Plankalkül program which computes the maximum of three variables by calling the function max [Zus72]:

```
P1 max3 (V0[:8.0],V1[:8.0],V2[:8.0]) => R0[:8.0]
max(V0[:8.0],V1[:8.0]) => Z1[:8.0]
max(Z1[:8.0],V2[:8.0]) => R0[:8.0]
END
```

```
P2 max (V0[:8.0],V1[:8.0]) => R0[:8.0]
V0[:8.0] => Z1[:8.0]
(Z1[:8.0] < V1[:8.0]) -> V1[:8.0] =>Z1[:8.0]
Z1[:8.0] => R0[:8.0]
END
```

Each data item was denoted with V (variable), C (constant), Z (intermediate result), or R (result), an integer number to mark them, and a powerful notation was used to denote the data structure of the variable. Zuse used the term ”plan” to describe the current notion ”program”. The language supported bit, integer, floating-point scalar data, array and record data structures. It also included some advanced features of modern programming languages, such as iterative control statements and recursion. As Plankalkül was not implemented in Zuse's lifetime, it was only a theoretical contribution and it did not directly influence subsequent early languages.

The first electronic computers appeared in the 1940s and were programmed in machine language by sequences of 0's and 1's that explicitly defined the operations and the order in which they were to be executed. The operations were low-level ones, e.g. move data from one location to another, compare two values, etc. This kind of programming was very slow, error-prone and the ultimate code was difficult to understand and modify.

1.3.2 The move to higher-level languages

The first step towards more user-friendly programming languages was the development of the mnemonic assembly language in the early 1950s. Initially, the instructions in assembly language were just mnemonic representations of machine instructions. Later, macro instructions were added to assembly languages so that a programmer could define the instruction parameters.

A great step towards higher-level languages was made in the late 1950s with the development of FORTRAN (*The IBM Mathematical FORmula TRANslating System*) for scientific computation, Cobol (*COmmon Business-Oriented Language*) for business data processing, and LISP (*LISt Processing*) for symbolic computation.

FORTRAN was the first important high-level language, developed in 1957. It introduced symbolic expressions and arrays, and also procedures (”subroutines”) with parameters. In other respects, FORTRAN, in its original form, was fairly low-level; for example, a significant part of control flow was determined by conditional and unconditional jumps. FORTRAN underwent a significant change since its original design with the latest version standardized as recently as 1997.

ALGOL 60 (*ALGOrithmic Language*) was an important step forward even though it is no longer used. It was the first major programming language to be designed for communicating with algorithms, not just for programming a computer. It was developed mainly because IBM, the owner of the FORTRAN

programming language, refused to relinquish proprietary control over what it considered to be its sole property. This lack of freedom to improve FORTRAN motivated the computing community of the late 1950s to develop another language. The new language (ALGOL 60) had several important new developments, such as:

- The concept of block structure was introduced. This allowed "blocks" of program to be created that could later be nested into other program components;
- Two different means of passing parameters to subprograms were allowed: pass by value, and pass by name;
- Procedures were allowed to be recursive;
- Stack-dynamic arrays were allowed. A stack-dynamic array is one for which the subscript range or ranges are specified by variables, so that the size of the array is set at the time storage is allocated for the array.

In some ways, ALGOL 60 was a great success, in other ways, it was a dismal failure. It succeeded in becoming, almost immediately, the only acceptable formal way of communicating algorithms in computing literature. Every imperative programming language designed after 1960 owes something to ALGOL 60, most of them being its direct or indirect descendants. ALGOL 60 was the first language that was designed to be machine independent. It was also the first language whose syntax was formally described. This successful use of the BNF formalism encouraged the development of several important fields of computer science. The structure of ALGOL 60 also affected machine architecture. On the other hand, ALGOL 60 has never become widespread, the main reason for which was that some of its features were rather difficult to understand.

COBOL was another important early high-level language. Its most important contribution was the concept of data descriptions, the forerunner of today's data types. Similar to FORTRAN, COBOL's control flow was fairly low-level. Just like FORTRAN, COBOL has also been significantly improved after its original design, the latest version having been standardized in 2002.

FORTRAN and ALGOL 60 were particularly useful for the purpose of numerical computation, whereas COBOL for that of commercial data processing. PL/I (*Programming Language One*) was an attempt to design a general-purpose programming language by merging features from the above three languages. On top of the existing ones, it also introduced many new features, including low-level forms of exceptions and concurrency. The resulting language was complex, incoherent, and difficult to implement. The PL/I showed that simply adding many new features on top of others is not the right way to make a programming language more powerful and suitable for widespread use.

BASIC (*Beginner's All-purpose Symbolic Instruction Code*) was designed at Dartmouth College by two mathematicians, who in the early 1960s produced compilers for a variety of dialects of FORTRAN and ALGOL 60. They decided

to design a new language that would use terminals as the method of computer access. The goals of the system were as follows:

- It must be easy to learn and use for inexperienced people (such as students);
- It must be pleasant and friendly;
- It must consider user time more important than computer time.

The original BASIC language had only fourteen different statement types, and a single data type. It was a very limited programming language, and thus easy to learn. The designers of the language decided to make the compiler available free of charge so that the language would become widespread. The introduction of the first microcomputers in the mid-1970s made BASIC very popular in the wider public, and especially among young computer fans.

From the late 1960s to the late 1970s many new programming languages emerged. Most of the language paradigms currently in use were invented at that time.

- Simula was the first language designed to support object-oriented programming.
- C was an early systems programming language.
- Smalltalk (mid 1970s) provided a complete design of an object-oriented language.
- Prolog was the first logic programming language.
- ML built a polymorphic type system on top of LISP, and was the basis for statically typed functional programming languages.

Parallelly, in the 1960s and 1970s there was also a considerable debate on the merits of "structured programming", which essentially meant programming without the use of GOTO. This debate concerned language design: some languages did not include GOTO, which forced structured programming by the programmer. By now, nearly all programmers agree that, even in languages that allow GOTO statements, it is better not to use it at all, or only in some exceptional situations.

Pascal was developed around 1970 by Niklaus Wirth as an improvement and simplification of ALGOLW. It was the most widely used educational language until the end of the 1980s. One of the unique features of Pascal was that it was the first language which, preceding Java by nearly 20 years, introduced the concept of intermediate code as an instrument for program portability. A Pascal program was translated into P-code by the Pascal compiler, which was also written in Pascal. P-code was a language for an intermediate machine with a stack architecture, which was then implemented in an interpretative way on the host machine. In this way, to port Pascal to a different machine, it was only necessary to re-write the P-code interpreter. In addition, Pascal was also

implemented in a compilative way that did not use an intermediate machine, thus allowing greater efficiency.

The 1980s was the time of relative consolidation. C++ combined object-oriented and systems programming. Objective-C added Smalltalk-style messaging to the C programming language, and it became the main programming language used by Apple for the OS X and iOS operating systems. The government of the United States standardized Ada, a systems programming language derived from Pascal and used by defense contractors.

One important trend in language design during the 1980s for programming large-scale systems was that programmers placed an increased emphasis on the use of modules, or large-scale organizational units of code. Modula-2, Ada, and ML all developed notable module systems in the 1980s, although other languages, such as PL/I, already had extensive support for modular programming. Module systems were often wedded to generic programming constructs [Ben06].

The rapid growth of the Internet in the mid-1990s created opportunities for new languages. Perl, originally a Unix scripting tool first released in 1987, became common in dynamic web sites. Java came to be used for server-side programming, and byte code virtual machines became popular, too. This era began to witness the spread of scripting languages (Python, Ruby, PHP). These did not directly descend from other languages, but rather featured new syntax and more liberal incorporation of features, thus making these scripting languages more productive than others. They came to be the most prominent languages used on the World Wide Web.

The evolution of programming languages continues in both industry and academic research. Current trends include the following features:

- Massively parallel languages for utilizing several thousand processors (e.g. graphics processing units or GPUs) and supercomputer arrays including OpenCL;
- Open source as a developmental philosophy for languages;
- New languages (e.g. XML) designed to describe special data sets such as documents;
- Constructs to support concurrent and distributed programming.

1.3.3 The future of programming languages

What is the future of programming language design? Predicting the future is notoriously difficult, and extrapolating recent trends is not easy, either. In the last 20 years, two second-generation Lisp programmers published two remarkable and influential studies on the evolution of programming languages, which came to different conclusions. In his essay "The End of History and the Last Programming Language" [GG96], Richard Gabriel was puzzled by the fact that very high-level, mathematically elegant languages such as Lisp have not caught on in the industry, whereas less elegant, and even semantically uncertain languages

such as C and C++ have become the standard. He concluded that the evolution of programming languages is driven by human and social factors (such as ease of learning and understanding) rather than by technical or conceptual principles. A decade later, Paul Graham described a different trend in his book Hackers and Painters [Gra04]. He believed that the most influential recent languages, such as Java, Python, and Ruby, have added features that move them further away from C, and closer to Lisp [Lou11].

Table 1.1 shows the popularity of programming languages based on the Tiobe index, a widely used measure of popularity of programming languages, calculated from the number of search engine results for queries containing the name of the language.

Programming language	Position (July 2012)	Position (July 2007)	Position (July 1997)
C	1	2	1
Java	2	1	5
Objective-C	3	46	-
C++	4	3	2
C#	5	7	-
Visual Basic	6	4	4
PHP	7	5	-
Python	8	8	23
Perl	9	6	6
Ruby	10	10	-

Table 1.1: Ranking of the top 10 programming languages during the last 15 years (source: *www.tiobe.com/index.php/content/paperinfo/tpci/index.html*)

Based on those statistics, it is hard to tell whether programming languages are moving mainly towards highly structured languages such as Java and C#, or towards more flexible ones such a C or C++. There are some clear trends, though, e.g. the rise of Objective-C thanks to the development of iPhone and iPad. As long as new computer technologies keep arising, there will be room for new languages and new ideas, and the study of programming languages will remain as fascinating and exciting as it is today [Lou11].

1.4 Programming language categories

Overall, we can identify some clear trends in the history of programming languages. One has been a trend towards higher levels of abstraction. The mnemonics and symbolic labels of assembly languages abstract away from operation codes and machine addresses. Variables and assignment abstract away from inspection and updating of storage locations. Data types abstract away from storage structures. Control structures abstract away from jumps. Procedures abstract away from subroutines. Packages achieve encapsulation, and thus improve modularity. Generic units abstract procedures and packages away from the types of data on which they operate, and thus improve reusability.

The many existing languages can be classified into families based on their model of computation. There are six basic computational models that describe most programming languages today:

- Imperative or procedural languages;
- Applicative or functional languages;
- Rule-based or logical languages;
- Object-oriented languages;
- Concurrent programming languages;
- Scripting languages.

The boundaries between the above families are blurred; a functional language may, for instance, be object-oriented.

1.4.1 Imperative or procedural languages

Imperative or procedural languages are command driven or statement-oriented languages. The first programming languages imitated and abstracted the operations of a computer. The first such computer was the von Neumann model, which had a single central processing unit that sequentially executed instructions that operated on values stored in the memory. Therefore, the typical features of a language based on the von Neumann model were the following: variables representing memory locations, and assignments allowing the program to operate on the memory locations.

A programming language that is characterized by the sequential execution of instructions, the use of variables representing memory locations, and the use of assignment to change the values of variables is called an imperative language. The syntax of such languages generally has the following form.

```
Statement_1;
Statement_2;
...
Statement_n;
```

1.4.2 Applicative or functional languages

The functional paradigm is based on functions over types such as lists and trees. The pioneer of functional languages was LISP, which demonstrated remarkably early that good programs can be written without resorting to variables and assignments. ML and Haskell are modern functional languages. They treat functions as ordinary values, which can be passed on as parameters and returned as results from other functions. Moreover, they incorporate advanced system types, allowing us to write polymorphic functions, i.e. functions that operate on data of a variety of types. ML (like LISP) is an impure functional language, since it does support variables and assignments. By contrast, Haskell is a pure functional language.

The fundamental characteristic of the languages in this paradigm is that they treat computation as the evaluation of mathematical functions and avoid state and mutable data. They emphasize the application of functions, whereas the imperative programming style emphasizes changes in state. Once an environment is fixed, an expression always denotes the same value.

1.4.3 Rule-based or logical languages

Logic programming is based on a subset of predicate logic. Logic programs infer relationships from the values, as opposed to computing output values from input values. Prolog was the pioneer of logic languages, and it is still the most popular one. In its pure logical form, however, Prolog is rather weak and inefficient, so it has been extended with extra-logical features to make it more useful and user-friendly as a programming language.

A well-known slogan originally by R. Kowalski captures the concepts that underpin the activity of exact programming: Algorithm = Logic + Control. According to this "equation", the specification of an algorithm, and therefore its formulation in programming languages, can be separated into two parts. Firstly, the logic of the solution is specified. Here "what" must be done is defined. Secondly, the aspects related to control are specified, and therefore the "how" of finding the desired solution is clarified. The programmer who uses a traditional imperative language must take account of both components. Logic programming, by contrast, implies, by definition, the separation of these two aspects. The programmer is required, at least in principle, to provide a logical specification only. Everything related to control is relegated to the abstract machine. Using a computational mechanism based on a particular deduction rule (resolution), the interpreter inspects the space of possible solutions, looking for the one specified by the "logic", defining in this way the sequence of operations necessary to reach the final result.

The basis for this view of computation as logical deduction can be traced back to the work of K. Gödel and J. Herbrand in the 1930s. It was not until the 1960s that a formal definition of this process was provided by A. Robinson and it took ten years to realize that formal automatic deduction of a particular kind could be interpreted as a computational mechanism. As a result, the first programming languages in the logic programming paradigm were created, Prolog being one of them.

Today there are many implemented versions of Prolog, and there exist various other languages in this paradigm (as far as applications are concerned, those including constraints are of particular interest). All of these languages allow the use of constructs permitting the specification of control for reasons of efficiency. Since these constructs do not have a direct logical interpretation, they make the semantics of the language more complicated, and partly sacrifice the purely declarative nature of the logic paradigm. This notwithstanding, we still use logic programming languages, even the "impure" aspects, which require the

programmer to do little more than formulate (or declare) the specification of the problem to be solved. In some cases, the resulting programs are unusual in their brevity, simplicity and clarity [GM10].

1.4.4 Object-oriented languages

The object-oriented paradigm has acquired enormous importance over the last 20 years. Object-oriented languages allow programmers to write reusable code that operates in a way that mimics the behavior of objects in the real world; as a result, programmers can use their natural intuition about the world to understand the behavior of the program and to construct appropriate code. In a sense, the object-oriented paradigm is an extension of the imperative paradigm. The difference is that the resulting program consists of a large number of very small pieces whose interactions are carefully controlled and yet easily changed. Moreover, at a higher level of abstraction, the interaction among objects via message passing can map nicely to the collaboration of parallel processors, each with its own allocated memory. The object-oriented paradigm has essentially become a new standard, much as the imperative paradigm was in the past.

The concepts of object and class had their origins in Simula, yet another ALGOL-like language. Smalltalk was the earliest pure object-oriented language, in which entire programs were constructed from classes.

C++ was designed by adding object-oriented concepts to C. C++ brought together the C and object-oriented programming communities, and thus became very popular. Nevertheless, its design is clumsy; it inherited all of C's shortcomings, and it added some more of its own.

Java was designed by drastically simplifying C++, removing nearly all its shortcomings. Although primarily a simple object-oriented language, Java can also be used for distributed and concurrent programming. Java is well suited for writing applets (small portable application programs embedded in Web pages), as a consequence of a highly portable implementation (the Java Virtual Machine) that has been incorporated into all major Web browsers. Thus Java has enjoyed a symbiotic relationship with the Web, and both have experienced enormous growth in popularity. C# is very similar to Java, apart from some relatively minor design improvements, but its more efficient implementation makes it more suitable for ordinary application programming.

1.4.5 Concurrent programming languages

A number of languages have been designed to support concurrency,[1] beginning with PL/I in the mid-1960s and including the contemporary languages Java and C#. The most important design issues for language support for concurrency

[1] Concurrency is a property of systems in which several computations are simultaneously executed, and potentially interacting with each other.

are the competition, cooperation synchronization, message passing, shared resources (including shared memory). Such languages are sometimes described as Concurrency-Oriented Languages or Concurrency-Oriented Programming Languages (COPL).

Today, the most commonly used programming languages that have specific constructs for concurrency are Java and C#. Both of these languages fundamentally use a shared-memory concurrency model. Of the languages that use a message-passing concurrency model, Erlang is probably the most widely used in industry at present [BA06].

Many concurrent programming languages have been developed more as research languages (e.g. Pict) rather than as languages for production use. However, languages such as Erlang, Limbo, and Occam have seen industrial use at various times in the last 20 years.

1.4.6 Scripting languages

Scripting is a paradigm characterized by the following features:

- use of scripts to glue subsystems together;
- rapid development and evolution of scripts;
- modest efficiency requirements;
- very high-level functionality in application-specific areas.

Scripting is used in a variety of applications, and scripting languages are correspondingly diverse. Several scripting languages were originally developed for specific purposes: csh and bash, for example, are the input languages of job control (shell) programs; awk was intended for report generation; PHP and JavaScript are primarily intended for the generation of web pages with dynamic content (with execution on the server and the client, respectively). Other languages, including Perl, Python, Ruby, and Tcl, are more deliberately general purpose. Scripts, "programs" written in scripting languages, typically are short and high-level, are developed very quickly, and are used to glue together subsystems written in other languages. So scripting languages, while having much in common with imperative programming languages, have different design constraints.

Later in this book, an entire chapter is devoted to each of these programming language paradigms.

1.5 Influences on language design

There are several factors that influence the basic design of programming languages. The most important of these are computer architecture and programming design methodologies. The basic architecture of computers has had a major effect on language design. Most of the popular languages of the past

50 years have been designed around the prevalent computer architecture, called the von Neumann architecture, after one of its originators, John von Neumann [Seb13]. In a von Neumann computer, both data and programs are stored in the same memory. The central processing unit (CPU), which actually executes instructions, is separate from the memory. Therefore, instructions and data must be piped, or transmitted, from memory to the CPU. Iteration is fast on von Neumann computers because instructions are stored in adjacent cells of memory [Seb13].

Years	Influences	New Technology
1951–1955	Hardware:	Vacuum-tube (valve) computers simulated finite-capacity Turing machines; mercury delayline memories
	Method:	Assembly languages, basic concepts: subprograms, data structures
	Languages:	Experimental use of expression compilers
1956–1960	Hardware:	Magnetic tape storage, core memories, transistor circuits
	Method:	Early compiler technology, BNF grammars, code optimization, interpreters, dynamic storage methods and list processing
	Languages:	FORTRAN, ALGOL, LISP
1961–1965	Hardware:	Families of compatible architectures, magnetic disk storage
	Method:	Multiprogramming operating systems, syntax-directed compilers
	Languages:	COBOL, ALGOL 60, SNOBOL, JOVIAL
1966–1970	Hardware:	Increasing size and speed and decreasing cost, microprogramming, integrated circuits
	Method:	Time-sharing systems, optimizing compilers, translator writing systems
	Languages:	APL, FORTRAN 66, COBOL 65, ALGOL 68, BASIC, SIMULA 67, SNOBOL4, ALGOL W
1971–1975	Hardware:	Microcomputers, mass storage systems, distributed computing
	Method:	Data abstraction, formal semantics, concurrent,embedded, and real-time programming techniques
	Languages:	Pascal, COBOL 74, PL/I (standard), C, Scheme, Prolog
1976–1980	Hardware:	Microcomputers,mass storage systems, distributed computing
	Method:	Data abstraction, formal semantics, concurrent, embedded and real-time programming techniques
	Languages:	Smalltalk, Ada, FORTRAN 77, ML
1981–1985	Hardware:	Personal computers, workstations, video games, local-area networks, APRANET
	Method:	Object-oriented programming, interactive environments, syntaxdirected editors
	Languages:	Turbo Pascal, Smalltalk-80, Prolog, Ada 83, PostScript
1986–1990	Hardware:	Age of microcomputer,engineering workstation, RISC architectures, Internet
	Method:	Client/server computing
	Languages:	FORTRAN 90, C++, SML
1991–1995	Hardware:	Very fast, inexpensive workstation and microcomputers, massively parallel architectures
	Method:	Open systems, environment frameworks
	Languages:	Ada 95, Tcl, Perl, HTML
1996–2000	Hardware:	Computers as inexpensive appliances, Personal digital assistants, WWW, Cabel-based home networking, Gigabyte disk storage
	Method:	E-commerce
	Languages:	Java, JavaScript, XML
2001-2005	Hardware:	Mobile devices, smartphones spread fast, wireless networks
	Method:	Mobile operating systems iOS, Android
	Languages:	Objective-C, Java 5, .NET, C# 2.0
2006-2010	Hardware:	High capacity networks, low cost computers and storage devices, mobile phone based networks
	Method:	Web/cloud applications
	Languages:	Django, Java 6, C# 3.0

Table 1.2: Some influences on programming language development. [PZ01]

The late 1960s and early 1970s brought an intense analysis of both the software development process and programming language design. An important reason for this research was the shift in the major cost of computing from hardware to software, as hardware costs decreased and programmer costs increased [Seb13]. The new software development methodologies that emerged as a result of the research of the 1970s were called top-down design and stepwise refinement. In the late 1970s, a shift from process-oriented to data-oriented program design methodologies began. The next step in the evolution is object-oriented design. Object-oriented methodology begins with data abstraction, which encapsulates processing with data objects and hides access to data, and adds inheritance and dynamic method binding. Inheritance is a powerful concept that greatly enhances the potential reuse of existing software [Seb13].

Process-oriented programming is, in a sense, the opposite of data-oriented programming. Although data-oriented methods now dominate software development, process-oriented methods have not been abandoned. On the contrary, a good deal of research has occurred in process-oriented programming in recent years, especially in the area of concurrency. These research efforts brought with them the need for language facilities for creating and controlling concurrent program units [Seb13]. We summarize the important changes and trends in programming language design in Table 1.2.

1.6 Principles of programming language design

1.6.1 Features of a good programming language

Even experts in the field disagree on what makes a great language, which is why so many novel ideas abound in the area of programming language design. A list of requirements and goals of programming language design are provided below [Sco09]. Importantly, different people involved in software design and use have very different preferences which all need to be taken into account. The following criteria will be grouped based on the different aspects and preferences of the different users of programming languages. These criteria are slightly modified from books by Alan Tucker and Ellis Horowitz. The criteria are grouped into three main categories:

- criteria relating to ease of using a language;
- criteria relating to software engineering;
- criteria relating to performance.

Criteria relating to ease of using a language

Programming languages are used by programmers who write programs. Thus, a good language should make it easy for a programmer to express what needs to be done. Several criteria contribute to making a language easy to use.

Well-definedness

The first criterion is well-definedness. Both the syntax and the semantics of the language should be clearly defined.

Syntax answers the question "What forms does the language allow?"

This is important, so that a programmer knows how to construct statements that will be accepted by the interpreter. If the syntax definition is ambiguous, then the programmer may have to resort to trial and error. Even worse, different interpreters for the same language may differ in their interpretation of an ambiguity, leading to portability problems.

Semantics answers the question "What does this form mean?"

The importance of this for the programmer is obvious. Ambiguity here may again force the programmer to resort to trial and error. A classic example of ambiguous semantics is the "dangling else" problem:

```
if (B1)
    if (B2) S1
    else S2
```

where B1 and B2 are boolean expressions and S1 and S2 are statements It can be interpreted as:

```
if (B1)
    if (B2)
        S1
    else  -- else goes with second if
        S2
```

or as

```
if (B1)
    if (B2)
        S1
else -- else goes with first if
    S2
```

ALGOL handled this by forbidding the `then` part of an `if` from being another `if`. Some newer languages (e.g. FORTRAN77, Modula-2, Ada) require an explicit `end if` to terminate an `if .. then .. else`, also avoiding the problem. Most languages resolve the ambiguity as Java does, by matching the `else` with the nearest `if` that has no `else` [Sco09].

Consistency with commonly used notation (expressivity)

Expressivity in a language can refer to several different characteristics. It more commonly means that a language has relatively convenient, rather than cumbersome, ways of specifying computations. For example, in C, the notation

`count++` is more convenient and shorter than `count=count+1`. This point can be illustrated best by looking at some violations of this criterion.

In writing mathematical expressions, certain conventions are normally understood with regard to operator precedence. For example,

```
3*x+2
```

is normally understood to mean

```
(3*x)+2
```

Most programming languages adhere to conventional rules of operator precedence, but some do not. For example, in APL the unparenthesized expression would be interpreted (by starting the calculation from the right) as

```
3*(x+2)
```

Another important criterion is that typo should not radically change the code's meaning. This possibility was notoriously illustrated by the software controlling an early space exploratory Venus probe, in which the intended FORTRAN code `DO 1 I =1,25` (which introduces a loop with control variable I ranging from 1 through 25) was mistyped as `DO 1 I =1.25` (which assigns 1.25 to an undeclared variable named `DO1I`).

PL/I was heavily and justifiably criticized for failing to control its complexity. A notorious example is the innocent-looking expression "25+1/3", which weirdly yields 5.33. PL/I used complicated (and counterintuitive) fixed-point arithmetic rules for evaluating such expressions, sometimes truncating the most significant digits.

Many people have criticized C, for example, for the common confusion between the assignment operator (=) and the equality test operator (==).

Good facilities for input/output

Older languages such as COBOL, FORTRAN and PL/I have complicated built-in input/output facilities. Being built-in and inextensible, they make a vain attempt to be comprehensive, but often fail to provide exactly the facilities needed in a particular situation. By contrast, modern languages such as C++, Java, and Ada have no built-in input/output at all. Instead their libraries provide input/output units (classes or packages) that cater for most needs. Programmers who do not need these units can ignore them, and programmers who need different facilities can design their own input/output units. These programmers are not penalized (in terms of language or compiler complexity) by the existence of facilities they do not use.

Uniformity

That is, similar constructs should have similar meanings. In the C family of languages (including Java), parameters to functions are normally passed by value. Thus, given the C function definition

```
int f(x)
int x;
{
     x = 2 * x;
     return x;
};
```

and a call to the function

```
int a = 2;
b = f(a); // a still has the value 2 here
```

The assignment to the formal parameter x in f has no effect on the actual parameter a. But if the parameter is an array, then it is passed by reference instead. Thus, given the very similar function definition:

```
int f(x)
int x[];
{
     x[0] = 2 * x[0];
     return x[0];
};
```

and a call to the function

```
int a[2];
a[0] = 2;
b = f(a); //The value of a[0] is now 4!
```

The assignment to the formal parameter x in f will alter the first element of the actual parameter a.

Orthogonality

The term orthogonality refers to attributes of being able to combine various features of a language in all possible combinations, with each combination being meaningful. Suppose a language has the four primitive data types, namely integer, float, double and character, and two type operators, namely array and pointer. If the two type operators can be applied to themselves and to the four primitive data types, a large number of data structures can be defined.

Orthogonality is closely related to simplicity. The more orthogonal the design of a language, the fewer exception the language rules require. Fewer exceptions mean a higher degree of regularity in the design, which makes the language easier to learn, read, and understand.

Generality

A debatable criterion is whether a language should be general, i.e. capable of tackling any type of problem. Carrying this too far can lead to failure. For example, in the late 1960s, IBM promoted a language called PL/I that was intended to replace FORTRAN, COBOL and ALGOL - among others - by incorporating facilities that would allow one to do everything one could do with FORTRAN and COBOL, with the elegance of ALGOL. This attempt, however, failed miserably and PL/I was unable to gain a significant programmer base.

Easy to learn

Being easy to learn is an important feature of a programming language designed for widespread use. Several of the features we have already considered contribute to this, e.g. consistency with commonly used notation, uniformity and orthogonality. Others, on the other hand, may make the language more difficult to learn. Especially, an abundance of a large number of features tends to make a language hard to use. Therefore, in case of some larger languages, the developers design subset languages, i.e. smaller versions of the language that include the necessary features while excluding less widely used ones.

Criteria relating to software engineering

Beyond ease of use, it is important that a programming language supports the development of correct software, even when writing large systems. The next group of criteria we consider pertains to support for engineering high-quality software.

Reliability

A program is said to be reliable if it performs according to its specifications under all conditions. Several language features have a significant effect on the reliability of programs.

- Type checking. Type checking is simple testing for type errors in a given program, either by the compiler or during program execution. Type checking is an important factor in language reliability. The earlier the errors are detected, the less expensive it is to make the required repairs. Java requires type checking of nearly all variables and expressions at the time of compilation. Types and type checking are discussed in depth in Chapter 5.
- Exception handling. The ability of a program to intercept run-time errors, take corrective measures and then continue running is a great aid to reliability. Ada, C++, and Java include extensive capabilities for exception

handling. Exception handling is much more difficult (though not impossible) in many other widely used languages, such as C and FORTRAN. Exception handling is discussed in Chapter 8.

- Aliasing. Loosely defined, aliasing means having two or more distinct referencing methods, or names, for the same memory cell. It is now widely accepted [Sco09] that aliasing is a dangerous feature in a language. Some kinds of aliasing can be prohibited by the design of a language.

Modularity

A large software project is typically constructed of modules, each of which interacts with the rest of the system in certain well-defined ways. Some languages, such as Modula-2 and Ada, provide more sophisticated features to support modular software, as we shall see later in the case of Ada.

One of the great strengths of object-orientation is the modularity inherent in the definition of a class.

Support for separate compilations

For small programs, it is common for the entire program to reside in a single file that is compiled as a single unit. However, for larger programs, it is almost essential to allow the program to be spread over multiple (sometimes several thousand) files compiled separately. In this way, when a change is made, only the affected file(s) need to be recompiled.

Many languages support this by adding a separate step to the program building process called linking.

Support for abstraction

Abstraction means the ability to define and then use complicated structures or operations in ways that allow many of the details to be ignored. Abstraction is a key concept in contemporary programming language design. Abstraction is a key factor in the writability of a language. Programming languages can support two distinct categories, namely, process abstraction and data abstraction.

- Process (or control) abstractions simplify properties of the transfer of control, that is, the modification of the execution path of a program based on the situation at hand. Examples of control abstractions are loops, conditional statements, and procedure calls.
- Data abstractions simplify for human users the behavior and attributes of data, such as numbers, character strings, and search trees.

Abstractions are also categorized in terms of levels, which can be viewed as measures of the amount of information contained (and hidden) in the abstraction. Basic abstractions collect the most localized machine information.

Structured abstractions collect intermediate information about the structure of a program. Unit abstractions collect large-scale information in a program.

Different programming languages provide different types of abstractions, depending on the intended applications for the language. In object-oriented programming languages such as C++, Object Pascal, or Java, the concept of abstraction has itself become a declarative statement - using the keywords virtual (in C++) or abstract (in Java). After such a declaration, it is the responsibility of the programmer to implement a class to instantiate the object of the declaration. Functional programming languages commonly exhibit abstractions related to functions.

Ideally, the type structure can be extensible, allowing the programmer to easily create and use new data types to fit the problem at hand. FORTRAN is an example of a language that is particularly weak on this, having only arrays as structured types - no records or pointer variables. Thus, what would be done with structs/classes in C-like languages will have to be done with individual variables in FORTRAN, and linked structures can only be implemented by using arrays of nodes - dynamic storage allocation is not possible. There is no facility for declaring new data types, either. A number of other languages share these shortcomings, including APL and BASIC.

Some languages (e.g. Ada, C++) even allow the standard operators to be redefined for user-defined data types.

Provability

The language lends itself to using formal methods to prove the correctness of a program. It is possible to construct a program proof by embedding precondition and postcondition assertions into the program. Unfortunately, two characteristics found in many programming languages tend to make constructing proofs difficult.

- The goto statement complicates proofs, because one cannot be sure what preconditions apply to a statement if it can be reached in more than one way.
- The possibility of two variables being aliases for one another complicates proofs.

To facilitate proofs, some languages do not have a goto statement and others have sufficient control structure flexibility to make its use practically unnecessary.

Criteria relating to performance

In the early days of computing, when computers were extremely slow and short of storage, languages like FORTRAN and COBOL were designed with numerous restrictions to enable them to be implemented very efficiently. Much has changed

since then: computers are extremely fast and have vast storage capacities; and compilers are much better at generating efficient object code.

A language's efficiency is strongly influenced by its conceptual basis. Some concepts such as dynamic typing, parametric polymorphism, object orientation, and automatic deallocation (garbage collection) inherently require significant resources. Logic programming is inherently less efficient than functional programming [Sco09], which in turn is inherently less efficient than imperative programming. The language designer must decide whether the benefits of each concept outweigh its cost.

Sometimes an interaction of concepts has paradoxical effects on efficiency. For example, in an imperative language, selective updating of an array is "cheap", while in a functional language an array-transforming operation would be "costly" (having to copy all the unaffected components). Conversely, in an imperative language, sharing of list components is inhibited (by the possibility of selective updating), while in a functional language sharing is always possible. Therefore, imperative languages prefer arrays while functional languages prefer lists.

Friedrich Bauer has suggested a useful principle: a language should not include a concept if it imposes significant costs on programs that do not use it. This principle is respected by Ada and Java exceptions: we can implement exceptions in such a way that a program that throws no exceptions runs as fast as it would do if exceptions did not exist. This principle is also respected by Ada's concurrency control abstractions. It is not respected by Java's synchronization features, which are designed to support concurrency, but which impose a cost on every object, even in a sequential program [Wat06].

Fast interpretation/compilation

In general, the more complex the syntax of the language, the longer a program of a given length will take to compile. When developing large programs, it is nice to have a separate compilation facility that allows the program to be spread over several files.

Efficient object code

This goal conflicts with the goal of fast interpretation. An optimizing compiler spends extra time during compilation to produce better object code. This is nice for production software, but is not as pleasant during program development. Some languages (for example gnu C) are supported by two compiler versions - a fast "checkout" compiler that produces less than optimal code, but which can be used during debugging; and an optimizing compiler that is slower but produces production-quality code.

Portability

One of the original reasons for adopting higher level languages was the desire to be able to move a program from one type of computer to another without rewriting it. To some extent, all higher-level languages achieve this goal; but some do much better than others.

Most important for portability is the existence of a well-defined and accepted language standard. Many languages have been standardized by formal bodies such as ANSI or ISO. For others, the original report by the language author may serve as a standard, and some languages have no clear-cut standards at all.

Standardization by itself is not enough, though, even when the standard is adhered to. Certain characteristics of the underlying machine have a way of showing up unavoidably in the implementation. For example, every machine has a basic word length which determines the range of integers that can be processed by regular machine instructions. Historically, microprocessor systems often used 16-bit integers; today many systems use 32 bits, and the 64-bit systems have also become widely used recently. A program which relies on the range of integers available on one machine may not run correctly on another machine whose range of values is smaller.

Possible solutions

The programming language evaluation criteria provide a framework for language design. However, that framework is self-contradictionary. Hoare (1973) states in his paper that ”there are so many important but conflicting criteria that their reconciliation and satisfaction is a major engineering task”.

Efficient code versus reliability

Ada language definition demands that all references to array elements be checked to ensure that the indexes are in their legal ranges. This adds a great deal to the cost of execution of Ada programs that contains large numbers of references to array elements. C does not require index range checking, so C programs execute faster than the semantically equivalent Ada, although the Ada program is more reliable [TN06].

Readability versus writability

APL includes a powerful set of operators for array operands. Because of the large number of operators, a significant number of new symbols had to be included in APL to represent the operators. Many APL operators can be used in a single long, complex expression. One result of the high degree of expressivity is that, for applications involving many array operators, APL is truly writable. Indeed, a huge amount of computation can be specified in a very compact program [TN06].

Flexibility versus safety

Pascal variant records allow a memory cell to contain different types at different times. For example, the cell may contain either a pointer or an integer. So a pointer value put in such a cell can be operated on as if it were an integer, using any operation defined for integer values. This provides a loophole in Pascal's type checking that allows a program to do arithmetic on pointers, which is sometimes convenient. However, this unchecked use of memory cells is, in general, a dangerous practice [TN06].

1.6.2 Language design

In the design of a new language, certain matters require careful assessment well before any consideration is given to the details of the design. The first and most important question that must be asked is whether it is necessary to design a new language. Is there an existing language that can be used to satisfy the requirement? Even if it requires a new implementation, implementing an existing language is easier and faster than designing and then implementing a new language.

The language designer's first problem, therefore, is a judicious selection of concepts. What to omit is just as important a decision as what to include as defining the success or failure of a programming language is very complex. A language is successful if it satisfies any of the following criteria:

- Achieves the goals of its designers.
- Attains widespread use in an application area.
- Serves as a model for other languages that are themselves successful.

When creating a new language, it is essential to decide on an overall goal for the language, and then keep that goal in mind through the entire design process.

Nevertheless, it is extremely difficult to describe good programming language design. Even recognized computer scientists and successful language designers offer conflicting advice. Niklaus Wirth, the designer of Pascal, advices that simplicity is paramount [Wir74]. C. A. R. Hoare, a prominent computer scientist and ALGOL designer, emphasizes the design of individual language constructs [Hoa73]. The designer of C++, Bjarne Stroustrup notes that a language cannot be merely a collection of neat features [Str94].

Horowitz suggested the following ten-step protocol to design a new programming language [Hor94]:

1. Choose an application area;
2. Make the design committee as small as possible;
3. Choose some precise design goals;
4. Release version one to small set of people;
5. Revise language definition again;

6. Build a prototype compiler;
7. Revise language definition again;
8. Write the manual;
9. Write a good compiler and distribute it;
10. Write primers.

1.7 The standardization process

We already emphasized the importance of the existence of a well-defined and widely accepted language standard. Documentation for the early programming languages was written in a informal way, in ordinary English. However, programmers soon became aware of the need for the more precise description of a language, and argued for the type of formal definitions used in mathematics. A further reason for a formal definition was the need for machine or implementation independence. The best way to achieve this was through standardization, which requires an independent and precise language definition that is universally accepted. Standards organizations such as ANSI (American National Standards Institute) and ISO (International Standards Organization) have published definitions for a number of languages including C, C++, Ada, Common Lisp, and Prolog.

Once a language is in widespread use, it becomes very important to have a complete and precise definition of the language so that compatible implementations may be produced for a variety of hardware and system environments. The standardization process was developed in response to this need. A language standard is a formal definition of the syntax and semantics of a language. It must be a complete, unambiguous statement of both. Language aspects must be defined clearly, while those aspects that go beyond the limits of the standard must be designated clearly as ”undefined”.

A language translator that implements the standard must produce code that conforms to all the defined aspects of the standard, but for an undefined aspect, it is permitted to produce any convenient translation. The right to define an unstandardized language, or to change a language definition, may belong to the individual language designer, to the agency that has sponsored the language design, or to a committee of the American National Standards Institute (ANSI) or the International Standards Organization (ISO). The FORTRAN standard was originated by ANSI, the Pascal standard by ISO. The definition of Ada is controlled by the U.S. Department of Defense, which funded the design of Ada. New or experimental languages are usually controlled by their designers.

When a standards organization decides to sponsor a new standard for a language, it convenes a committee of people from industry and academia who have a strong interest in and extensive experience with that language. The standardization process is not easy or smooth. The committee must decide which

dialect, or combination of ideas from different dialects, will become the standard. The committee members approach the task with different notions of what is good or bad, and have different preferences. Agreement at the start is rare, and the harmonization process may take several years. This was the case with the original ISO Pascal standard, the ANSI C standard, and the new FORTRAN-90 standard.

After a standard is adopted by one standards organization (ISO or ANSI), the definition is re-evaluated by the other. In an ideal situation, the new standard is accepted by the other one, as well. For example, ANSI adopted the ISO standard for Pascal nearly unchanged. However, smooth sailing is not always the rule. The new ANSI C standard was rejected by some ISO committee members, and a number of amendments had to be performed during the standardization process. The first standard for a language often clears up ambiguities, fixes obvious defects, and defines a better and more portable language. For instance, ANSI C and ANSI LISP standards do all of these. Programmers writing new translators for these languages must then conform to the common standard. Implementations may also include words and structures, called extensions, that go beyond anything specified in the standard.

1.8 Summary

In this Chapter, we have first introduced general concepts about programming language design and implementation options, all of which strongly determine the overall efficiency of a programming language. We then went through the history of programming languages. We have seen how new languages inherited successful concepts from their ancestors, and how they sometimes introduced new concepts of their own. We have discussed how major programming paradigms evolved. We have identified a number of technical and economic criteria that must be taken into account when selecting a language for a particular software development project. All this information will be of key importance in the following chapters in this textbook.

1.9 Exercises

Exercise 1.1. Write an evaluation of a programming language you are familiar with, using the criteria described in this chapter.

Exercise 1.2. C requires a semicolon to be placed between the *then* and *else* branches of a conditional if-then-else statement, whereas this is prohibited in Pascal. What are the pros and cons of the two regulations?

Exercise 1.3. Certain programming languages distinguish between uppercase and lowercase characters in identifiers. What are the pros and cons of this design decision?

Exercise 1.4. FORTRAN does not require all variables to be declared before being used. What problems may result from this during syntax processing?

Exercise 1.5. Explain the different factors that determine the overall cost of a programming language.

Exercise 1.6. Describe, in your own words, the concept of orthogonality in programming language design.

1.10 Useful tips

Tip 1.1. Consider the following:
overloading, memory allocation, support of data abstraction, different ways of using complex conditionals for loops, etc.

Tip 1.2. Consider what semicolons are usually used for and whether this use is justified within an if-then-else statement. Also think about a case when you later add an *else* clause to an existing if-then statement. What happens to the semicolon?

Tip 1.3. Consider whether additional information about the case of letters is necessary or useful. Does this extra information positively or negatively affect the readability of the program code?

Tip 1.4. Ask yourself what would happen to typographical errors in FORTRAN.

Tip 1.5. The different aspects of the cost of a programming language are a) the cost of deployment, b) the cost of maintenance, and c) the cost of support.

Tip 1.6. Orthogonality is the property that means "Changing A does not change B". An example of an orthogonal system would be a radio, where changing the station does not change the volume and vice-versa. A non-orthogonal system would be like a helicopter where changing the speed may change the direction.

1.11 Solutions

Solution 1.1. We will consider the Java language.

Readability

In terms of readability, Java has some issues with simplicity with respect to readability. There is feature multiplicity in Java as shown in the textbook with the example of

```
count=count+1, count++, count+=1, ++count
```

being four different ways to increment an integer by one. Another problem is operator overloading since Java allows some operators such as the + sign to add integers, floats, and other number types.

Control statements in Java have higher readability than Basic and Fortran programs because they can use more complex conditionals like for loops. There is no need for goto statements that have the reader leaping to other lines of code that could be far away or out of order. However, the use of braces to designate the starting and stopping points of all compound statements can lead to some confusion.

Writability

Java has a fair bit of orthogonality in that its primitive constructs can be used in various different ways. Because Java is an imperative language that supports objects object oriented programming, it can be fairly complex. Java supports data abstraction so it would be easier to create a binary tree in Java with its dynamic storage and pointers than in a language like Fortran 77. Java also has a for statement which is easier than using a typical while statement. Java is a high level programming language so specifying details like memory allocation are unnecessary due to Java's dynamic array system.

Solution 1.2. Semicolons are mostly used between two statements, either to separate them (Pascal) or to terminate the preceding statement (C). Since a conditional if-then-else statement is a single statement, there is no justifyable reason to place a semicolon between the then and else branches. In this theoretical sense, the Pascal version is more appropriate.

On the other hand, when you want to add a new else branch to an existing if-then statement in Pascal, you need to go back to delete the semicolon in the preceding line; if you forget this, a syntax error is generated. This problem does not occur in C since the semicolon can remain at the end of the then branch. Therefore, the C version is more practical.

Solution 1.3. Considering the distinction between uppercase and lowercase characters in identifiers.

Pros:

The same words can be used in different meanings depending on the use of uppercase or lowercase letters. E.g. in Java *Byte* is a class whereas **byte** is a primitive type. We can also differentiate between constants and variables or dynamic and static names.

Cons:

Case sensitivity may lead to small hard to detect differences between identifiers.

Note that a different situation is when both uppercase and lowercase characters are allowed but no distinction between them is made at the syntactic level. A typical such example is Visual Basic.

Solution 1.4. In case of typographical errors, the compiler will not know if this is an error or a new variable, therefore it may generate a new variable instead of reporting the syntax error.

Solution 1.5. The different aspects of the cost of a programming language are:

- The cost of deployment;
- The cost of maintenance;
- The cost of support.

Solution 1.6. Orthogonality is the property that means "Changing A does not change B". An example of an orthogonal system would be a radio, where changing the station does not change the volume and vice-versa. A non-orthogonal system would be like a helicopter where changing the speed can change the direction.

In programming languages this means that when you execute an instruction, nothing but that instruction happens (very important for debugging). There is also a specific meaning when referring to instruction sets.

2 Lexical elements

The common characteristic of source codes from different programming languages is that they are made of as sequences of symbols from a given set. The structure of these sequences is described by the lexical and syntactic rules of the given programming language. The basic language units called lexical elements *are the building blocks of program units. In this chapter we examine from what kind of symbol sets lexical units can be built, to which level this process is standardized for each of the programming languages, how the identifiers of these languages are constructed, and which numerical-, character- and text literals are allowed. We discuss applicable comment forms in source code, since this can also affect the reliability of our programs.*

Source code is made of one or more *compilation units*.[1] Compilation units are built from sequences of *lexical elements*.

Lexical elements are defined by given rules as character sequences separated by delimiters. So lexical elements include delimiters, identifiers, numerical-, character- and text literals, and comments.

2.1 Symbol sets

Symbol sets usable in source codes define not only the program text, but also control data input. For this reason, standardization of these symbol sets is a key factor for portability.

Computers manage and communicate data in binary form based on bits in groups of 8, called *octets*. Therefore, the value range of an octet is an integer number between 0 and 255, which is normally given in decimal, octal or hexadecimal form for better readability. Octets are often called bytes, but mind the difference: although an octet is represented with 8 bits (that is with a byte), interpreting it as a byte means the above mentioned positive value range, but on 8 bits negative values could also be encoded by assigning a sign bit, or using different coding methods (like BCD[2] or two's complement).

There are many conventions on how an octet or a sequence of octets represent data. Naming conventions are used exchangeably for the number of octets and of their representing bits. For example, 4 consecutive octets (32 bits) often represent a real number using some standardized encoding, or in ASCII one, in UTF two octets (16 bits, 2 bytes) implement characters.

It is important to distinguish between *character set*, *character code* and *character encoding*. We define these according to Jukka Korpela's study about characters [Kor02].

[1] About compilation units see Chapter 4.

[2] Binary Coded Decimal: low and high 4 bits of a byte represent a decimal digit each.

The *character set* is simply the set of all allowed characters. Nothing is presumed about the internal representation of the characters within the computer. The set does not even require an ordering of the characters, this must be defined separately. Character sets are normally defined by enumerating the name and the visual appearance pairs of their elements. Keep in mind, that a set could contain different characters with the same visual appearance, like the Latin capital `A`, the Cyrillic capital `А` and the Greek capital Alfa (`Α`).

Examples:

```
EXCLAMATION   !      QUESTION_MARK   ?      SEMICOLON   ;
```

character set element names (or shorter: character names) are rather identifiers than definitions for them. These names can usually contain letters from `'A'..'Z'`, spaces and underscores. The same characters can have different names in different character set definitions. Character names presumably suggest some general meaning and hint usage scope; but are advised that usage possibility could be much broader.

Character code is a mapping usually given in tables, which define a mutually unambiguous correspondence between the elements of the character set and integer numbers. This means, that a unique numeric code, so called *code position* is given for all the set members. The code mapping is seen as one contiguous table (irrespectively of the actual numbers of defining tables given), which is indexed by the code positions. Synonyms for code position are *code value*, *code point*, *code-set value*, or simply just *code*.[3]

Character encoding is an *algorithm* to define a digital format for handling characters. It maps sequences of character codes on sequences of octets. In the simplest case every character is mapped to an integer in the range 0–255, using their character code as octets. This allows, of course, only for a maximum of 256 characters this way.

A character code table directly defines a character set, and the character encoding is often given by character codes (and the defining character set). Logically the *character set* is primary for providing the set of characters. It also gives the character codes, the numeric values assigned to the characters –, for example, in the ISO 10646 character coding the codes of the characters `'a'`, `'ä'` and ‰ (the thousandths sign) are 97, 228 and 8240. The character encoding defines how character codes are encoded as octet sequences. For example, one possible coding of ISO 10646 uses two octets for every character encoding `'a'`, `'ä'` and the ‰ sign with the octet pairs of (0, 97), (0, 228) and (32, 48). Using some notions ambiguously can lead to problems, such as *character set* can mean the character set, but also the character codes, or sometimes the character encoding.[4]

[3] It is not required that mapped character codes should cover a contiguous integer range. Actually most of the character codes have “holes”, empty code positions, which are mapped for control sequences or are reserved for future use.

[4] Using the notion *character set* for the meaning of “coding” is troublesome.

The most widely used internationally accepted and standardized character codings are ASCII, EBCDIC, ISO 8859-1, ISO 8859-2 and Unicode with multiple possible encodings. The growing demand for specific national characters played an important role to establish these new standards.

2.1.1 The ASCII code

To understand why the introduction of the ASCII in 1963 had such a big impact, it is worth mentioning that before this period of time different computers were unable to communicate with each other. Every manufacturer had their own method to represent the letters of the alphabet, numbers and control codes. "Characters were represented in computers in more than 60 different ways. This was a real tower of Babel." – explained Bob Bemer[Bra99], who actively participated in the development of ASCII, and is also known as the "father of ASCII".

ASCII stands for *American Standard Code for Information Interchange*, acts as a "common denominator" for every computer nowadays, which could have nothing else in common. It took more than two years, till this codeset suggested by the ANSI (American National Standard Institute) had been established. Today this is the most common encoding. It is so prevalent, that an "ASCII file" now simply denotes a text (that is not binary) file, even if its encoding is something different. Most encodings contain ASCII as a subset. The first 32 codepoints and the codepoint 127 define *control characters*, like *line feed* (LF) or *escape* (ESC). The actual printable part of the character set is shown in Table 2.1. (The whole codetable is in the Appendix 18.1.)

␣	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Table 2.1: Graphical characters of ASCII. The first character is the space

In the original standard the value range 128–255 was not used, but later as the code points were running out of various extensions were introduced. Such a widely used extension is shown in the Appendix 18.3. The ISO 8859-1 or Latin-1 and the ISO 8859-2 or Latin-2 character sets are actually extensions of ASCII. So, the original ASCII is often referenced as US-ASCII.

2.1.2 The EBCDIC code

Beside ASCII, there is another widely used encoding system, the EBCDIC (*Extended Binary Coded Decimal Interchange Code*), which was introduced by

IBM in the sixties. Curiously, it includes some non-used codepoints in the value range 0–255, and the codes for the characters 'a'..'z' and 'A'..'Z' are following the usual ordering, but not continuously. Moreover, there are character codes also wedged in here. For example, the code of the character 'r' is 153, after that 7 empty codepoints follow, then after the code 161 for '~'the alphabet continues with the character 's' from code 162. This encoding has at least 6 slightly different forms. The most commonly used complete codetable is presented in Appendix 18.4.

2.1.3 The ISO 8859 family

The need for special characters in various languages resulted in the creation of ISO 8859-1, ISO 8859-2 and more encodings, which assigned the codes between 128 – 255 which were not used by ASCII to some special – mostly graphical – characters. The ISO 8859-1 or Latin-1 is probably the most popular 8 bit encoding standard in the EU countries. It contains most of the accented letters from western Europe, such as 'ö', 'ô', 'ò', and also some other special characters: '©', '¿', '<', '¡', '£' etc. It does not contain – among others – 'ő' and 'ű' which are important characters in Hungarian. The complete Latin-1 table is shown in Appendix 18.1.

The special Hungarian characters ('ő', 'ű') are introduced in the ISO 8859-2 or Latin-2 standard. Most of the middle- and east European accented letters such as 'ľ' or 'š'are included in this system. However, some special characters from Latin-1 such as '©', '¿", '<', '¡', '£' etc. had no place in it. Hungarians were lucky though because a Latin-2 encoded Hungarian text read in Latin-1 coding is – although not perfect – still understandable, only the letters 'ő' and 'ű' will be shown as 'õ' and 'û'. (The Czechs or Poles were not that fortunate.) The complete Latin-2 table is in Appendix 18.2.

ISO 8859-3 or Latin-3 serves the needs of the south-European languages (such as Maltese, and also Esperanto), ISO 8859-4 or Latin-4 defines special characters for the north-European languages (Estonian, Lithuanian, Latvian, Greenland, and Lapp). A separate codetable, the ISO 8859-5 helps the users of Cyrillic letters such as Bulgarian, Byelorussian, Russian, Serbian, etc., another the basic Arabic character set users (ISO 8859-6), still different ones are used for Greek (ISO-8859-7), and for Hebrew (ISO-8859-8).

For more about the ISO 8859 encoding, see the study *The ISO 8859 Alphabet Soup* from Roman Czyborra [Czy98a].

2.1.4 The Unicode standard

To get around the problems in multilingual environments caused by the former standards, *ISO 10646* standard was introduced in 1993.[5] It defines UCS, the

[5] The number of this standard relates to 646, which is the ISO equivalent of ASCII.

Universal Character Set, which is a huge, ever growing character set. Tens of thousands of characters were defined, according to common principles, giving each of these characters standard unique names. It also contains, as subsets most of the character sets described above, like ASCII and Latin-1, the accented characters of middle- and eastern-European languages, the IPA (International Phonetic Alphabet), Greek, Cyrillic, Georgian, Armenian, Hebrew, Arabic letters, the character sets of languages from the Indian continent, the symbols used by the Chinese/Japanese/Korean writings, mathematical operators, various graphical characters, and also special OCR symbols used to process checks. For details about all of the character groups see the compilation made by Jukka Korpela [UCS02]. About the coding problems of the eastern-Asian writings the study by Steven J. Searle titled *A Brief History of Character Codes in North America, Europe, and East Asia* is worth reading [Sea02].

It follows from the size of the set, that most of the characters occupy other code positions than in the previous standards.

Unicode is a standard by the Unicode Consortium, which is fully compatible with the character set and coding defined by ISO 10646.[6] It also determines some encodings: first, it was suggested that the 8 bit implementation with the value set of 0–255 should be extended to 16 bits on the range 0–65535. This is, for example, what Windows NT uses. After the introduction of the original idea it was concluded, that this is too much, and at the same time too little, because for the first 256 characters – and this is often the case – 8 bits would be sufficient, on the other hand not even a 16 bit implementation would cover all the emerging coding needs of the world. That is why different Unicode standard encoding formats were introduced, like UTF-8, UTF-16 etc., to encode character sequences in a system independent way. (These are often called "character sets", but they describe coding algorithms, and not really character sets!) For emailing for instance the Internet Mail Consortium recommends the UTF-8 standard.

In practice, the name Unicode is more prevalent than the exact name ISO 10646 and is used in different descriptions as equivalent to that of the ISO standard. For more about the Unicode encoding see the study of Roman Czyborra titled *Unicode Transformation Formats: UTF-8 and Co.*[Czy98b].

2.2 Symbol sets of programming languages

Most of the early programming languages allowed only the letters of the English alphabet, digits and some special characters, like brackets, semicolon, etc. in lexical elements.

Nowadays the need is emerging that programs should be able to handle all the special characters used by different nations, and there should be a way to

[6] Originally only with the first 2^{16} characters of ISO 10646 defined by the Basic Multilingual Plane (BMP).

define the ordering of those according to the actual alphabetical order in these languages.

By evaluating the symbol sets of programming languages, one should study, how and at which level they answer the following questions:

- How precise is the definition of the allowed symbol set?
- What kind of symbols are allowed within the source code?

Pascal

The standard of *Pascal* from 1990 (ISO 7185:1990) only defined minimal requirements: "the character range `'0'..'9'` must be ordered and continuous", "the range `'A'..'Z'` (if exists) must be ordered, but need not be continuous", and the same for the range `'a'..'z'`. The possible set of characters is not specified, and different implementations are allowed to use different character sets [Cat01].

ISO 646 is suggested, but not mandatory.

This incomplete regulation seriously influences the portability of programs written in the Pascal language. Consider the following examples: if a loop iterates over the character range from `'a'` to `'z'`, or the code of the characters `'A'` and `'1'` are compared according to character set ordering, the achieved results will differ in EBCDIC and ASCII systems.

Ada

It is worth noting what major changes the standard of *Ada* went through.

The allowed encoding was already rigorously fixed in 1983, but only for the ASCII character set (ISO 646).

In 1995 the actual version of the standard specified generally all 65536 characters from the ISO-10646-1 character set for every Ada 95 source code, but – as we will discuss later – only the 193 defined graphical characters were allowed in identifiers.

In 2012 the new standard already allows "the entire coding space described by the ISO/IEC 10646:20112003 Universal Multiple-Octet Coded Character Set." [Ada12]

C++

The *C++* standard from 1998 (ISO/IEC 14882:1998) allowed only a relatively limited character set for source codes (91 graphical characters plus the vertical and horizontal tabulator, linefeed, newline and space). The character encoding was precisely specified: the code values must come from a subset of ISO 10646, and must include the whole ASCII. This was a major step compared to the standard from 1990, which did not specify any encoding at all. A new feature, the universal character names were also introduced.

Java and C#

The specification of *Java* and *C#* allows the use of any Unicode characters within the source code. A new feature, the *Unicode sequence* is introduced to specify Unicode characters in the form: '\uXXXX', where each X denotes a hexadecimal digit. The designers of C# also advise the usage of UTF-8 for encoding.

Delimiters

In most of the programming languages – such as Pascal, C++, Ada, Java – delimiters include the space, tab and new line characters. In the early programming languages – such as FORTRAN or ALGOL 68 – space was not always considered as a delimiter. Let's check out with a small example, how this seemingly insignificant decision combined with automatic variable declaration could severely affect program reliability:

Imagine the following FORTRAN code: `DO 10 I = 1.5` which was intended to implement a loop, but the comma separator was mistyped as a point. The compiler will not complain, as this is syntactically valid, and will be interpreted as an assignment of the value `1.5` to the variable named `DO10I`.

Delimiters could be only one character long, or could consist of a sequence of characters, for some possibilities see Table 2.2.

2.3 Identifiers

Identifiers are used to name variables, types, subprograms, etc. in a source code. Let's review how different programming languages handle the following questions:

- Which characters are allowed to be used in identifiers?
- What is the allowed syntax of the identifiers?
- Are letters handled case sensitively?
- Is there a limit on the length of identifiers?
- Are there reserved (key-) words, which cannot be redefined by the programmer?

Most of the programming languages allow only the letters of the English alphabet ('A'..'Z', 'a'..'z') and digits ('0'..'9') in identifiers. That is exactly what the ISO 7185 Pascal standard from 1990 specifies. For better readability, numerous languages extended this set first with the underscore ('_'): Extended Pascal (ISO 10206), CLU, Eiffel, C++ (ISO/IEC 14882:1998). There is a difference in the allowed syntax if the underscore symbol is treated as a normal letter, or as an extra special character.

In certain languages some special characters can be used to specify additional information about the variable identified on its beginning or ending. For example,

	Pascal	C++	Java	Ada 95	CLU	Eiffel
space, tab, linefeed	Yes	Yes	Yes	Yes	Yes	Yes
Single character	`& ' ( ) + - * / : . ; < = >`	`& % ^ ' ( ) + - { } | ~ [ ] \ " * / : . ; , < = > ! ?`	`( ) { } [ ] ; , . + - * / : < = > ! ~ ? : & | ^ %`	`& ' ( ) * + , - . / : ; < = > |`	`( ) { } [ ] : ' . $ " , \ + - * / < = > ~ & |`	`; , : . ! = ( ) [ ] { } " + - $ % /`
Character sequence	`:= >= <= <> << ** ..`	`-> ++ - .* ->* == <= >= && || << >> <<= >>= != .. += -= &= ^= |= :: *= /=`	`== <= >= << != && || ++ - >> >>> += -= *= /= &= |= ^= %= <<= >>= >>>=`	`=> .. ** := /= >= <= << >> <> -`	`:= <= >= ~< ~<= ~= ~>= ~> || ** //`	`- !! /= -> .. << >> := ?=`

Table 2.2: Table of delimiters

in Perl the prefix symbols $, @, or % denote a scalar, indexed or associative variable. In BASIC, the last character of the identifier also specifies the type of the variable.

The first step to support the definition of identifiers on native languages was made by the designers of Ada 95: allowed letters of identifiers are those from BMP Row 00 (that is the first 256 code points) which have the prefix "Latin Capital Letter" or "Latin Small Letter" in their name, so for example `Távolság` or `Sebesség` are valid. On the other hand, the accented letters `'ő'` and `'ű'` were not allowed, because their codes are outside of this valid range.

The new standard of Ada in 2012 - following the already existing rules in Java - supports "any character whose General Category is defined to be Letter"[Ada12], thus allowing to write identifiers in every possible languages.

Java was the first language, where in identifier names all letters and digits are allowed of every possible languages, such as Cyrillic, Greek, Hebrew, Arabic, Chinese, Thai, Indian, Japanese, Georgian, Armenian or Hungarian.[7] So in Java e.g., "$\alpha\beta\gamma$" or `Felső` are valid identifiers. The same principle is followed by the identifiers in C#.

[7] For the complete specification see [sun03a].

2.3.1 Allowed syntax

In most of the programming languages identifier names start with letters, followed by letters or digits. The usual BNF syntax description of this rule is the following:

⟨*letter*⟩ {⟨*letter*⟩|⟨*digit*⟩}

This is how valid identifier names are formed, for example, in ALGOL 60, FORTRAN, Pascal, Modula, C and C++. From these languages in Pascal, C and C++ the set of letters was extended with the underscore. Java, as already stated, has a much bigger set of letters, which also includes the '_' symbol. For example, in Java `El_Niño`is a valid identifier.

In other languages, such as Ada, the set of letters is also bigger, but the '_' symbol is not a letter, nevertheless it can be used within identifiers names to arrange long names, with the following syntax:

⟨*letter*⟩ { ['_'] ⟨*letter*⟩ | ['_'] ⟨*digit*⟩ }

2.3.2 Distinction between lower and upper case letters

The question of the distinction lower from upper case letters arose already in the first programming languages. Some of the early languages (such as FORTRAN, COBOL) solely accepted upper case – only this could be punched on cards. Compilers later allowed the equivalent usage of lower case, and then simply converted to upper case during compilation.

Designers for one of the trends of programming languages argued, that according to the design principle stating that the same appearance must have the same meaning, this requires not to distinguish lower from upper case letters. For this reason, the names *dog*, *DOG*, or *Dog* must identify the same object. This viewpoint was declared by the designers of Pascal, Ada, or even Visual BASIC. It is interesting to know, that these languages apply the same principle even to accented letters, owing to that, the identifiers *KÖRTE* and *körte* are same.

Other designers – started from ALGOL 60 and ALGOL 68, continued by C, C++ and Java – declared that it is practical to consider every letter as distinct. Mixed case identifiers can be avoided in C by using only lower case identifiers. But in Java the standard library itself uses mixed case identifiers, like *parseInt* for the method to convert a string to an integer number, which cannot be called neither *ParseInt* nor *parseint*.

An unusual approach was chosen by Bertrand Meyer, the designer of Eiffel. Here lower and uppercase letters are equivalent, but the same name can be used for an object and its type, so *apple: APPLE* is a valid declaration [Mey91].

2.3.3 Length restrictions

The first programming languages used only single character names.

This came naturally, since in mathematical equations variables used to be named like `a`, `b`, `x` or `y`, and the first programmers were mainly mathematicians or engineers.

In FORTRAN 66 and even in FORTRAN 77 identifiers had a length limit of 6 characters, which was raised to 31 in FORTRAN 90.

The later programming languages – like the standard of C++ from 1998, Ada, Java or C# – do not imply any length restrictions, identifiers can be of any length. Compiler implementations can define a limit to keep symbol tables from overgrowing, but this should cause no problems for "normal" programmers.

2.3.4 Reserved words

Programming languages reserve, in most cases, certain words for special meanings. These words are used as statements and also to arrange source code. Branches, for example, usually start with the word `if`, loops with `while` or `for`, etc.

These words are called *keywords*, and in most of today's programming languages are also *reserved*, that is they are not allowed to be used as identifier names.[8] The absence of reserved words may result in unclear and hardly maintainable programs, causing huge interpretative problems eventually at later modification attempts of such source codes. In FORTRAN, for example, if the `REAL` keyword stands on the beginning of a statement line followed by an identifier, it means the declaration of a `REAL` type variable. But if the same `REAL` word is followed by the assignment symbol, the compiler would interpret it as a variable name:

```
REAL APPLE
REAL = 2.5
```

After these declarations in the above example, the usage of the `REAL` word is ambiguous and it could easily get mixed up if it is a keyword, or a variable name.

Numerous languages introduce so called *predefined names*, which could be seen as between keywords and reserved words in a sense. For example, in Ada the *Integer*, *Float*, *Boolean* elementary built-in types or the *True* and *False* values were defined not as language keywords, but as part of the Standard library. So theoretically, they could be redefined. Such possibility of redefinition is sometimes beneficial, as with this code

type *Integer* **is range** *−999_999. .+999_999;*

[8] Part of the literature calls standard words, what we defined as keyword here, and uses keyword for not allowed reserved words (see e.g. [Cso96]).

our program becomes independent of the actual running environment. On the other hand, it is better to be careful:

True: Integer;

declarations like this could be misleading later for every reader of the source code.

As an interesting feature of C#, every keyword prefixed with the `@` symbol can be used also as an identifier (for example: **@bool**). "Normal" identifiers must not start with this special character. This feature was introduced to sustain program portability and interoperability in the .NET environment. Symbol names starting with the `@` character are actually stored without this prefix. So even if a source code from another language uses, for example, the keyword **sealed** as an identifier, there is an easy way to reuse the same code in C# utilizing this feature.

2.4 Literals

Literals are the used constants in the source code.

2.4.1 Numeric literals

Numeric literals practically consist of only digits `'0'..'9'` and letters `'A'..'F'`, in some languages the `'_'` (underscore) symbol is also allowed for arrangement but has no effect on the numeric value.

The structure of numbers

For numeric literals basically integer and real numbers are distinguished.

Integers

Decimal *integers* are usually plain sequences of digits, with the following BNF syntax:

`['+'|'-']` ⟨*digit*⟩ `{`⟨*digit*⟩`}`

Examples: `−123` or `456789`. In Ada or Perl the latter can also be specified in the form `456_789`. Eiffel rules formulate, that for any given `'_'` symbol it must be followed by exactly 3 digits on its right side.

In the C, C++, Java or C# languages to indicate the difference between **int** and **long** types the letter *l* or *L* could be appended to the end of the integer number. That is why the type of *2* is **int**, but of *2L* it is **long**. To distinguish **byte** and **short** types, there is no similar indication. In expressions an explicit typecast can be used for this. The *u* or *U* postfix can be used likewise to mark **unsigned** values.

Real numbers

Real numbers differ mostly from integers by containing a decimal point (.). The general syntax of valid *real numbers* is the following:

['+'|'-'] ⟨*digit*⟩ {⟨*digit*⟩} '.' ⟨*digit*⟩ {⟨*digit*⟩} [⟨*exponent*⟩]

The exponent is started usually with the letter `E`, followed by a possible premonitory sign, and the exponent value. The represented value would be: $\text{sign} \times \text{number} \times 10^{\text{sign\_exponent} \times \text{exponent}}$. For example, the $-$`123.456E+3` form represents the `-123456.0` real value.
To the above almost every language adds some specialty:

Pascal

In Pascal using an exponent always specifies a real number, so the *12E+3* form denotes the real value of *12000.0*, even if it does not contain a decimal point [Cat01].

Ada

In Ada integers also have an exponential form, where the exponent must not be negative. So `1E6` gives the integer value 1 000 000.In the exponential form for real numbers negative exponents can be used [Ada95].

Eiffel

In Eiffel, for real numbers one digit and a decimal point are enough, digits are not needed on both sides of the decimal point. (Example: $-$`1.`)

C++, Java

In C++ and Java real number literals have the type **double** by default (*-12.3*, *12.3e4*) unless the letter *f* or *F* is appended to them at the end, which declares the value to be of type **float** (*3.141592f*, *2.9e-3f*).

The **double** type can be emphasized by appending the letter *d* or *D* ([CPP98] and [Nyek08]). The *NaN* (Not a Number) value was also introduced. This is also the result of *0.0/0.0*.

Java

In Java the *zero* value has a premonitory sign: *0.0*, and *-0.0*. These values are equal, but in some cases they give different results, e.g.: *1.0/0.0* results positive infinity (*POSITIVE_INFINITY*), but *1.0/-0.0* gives negative infinity (*NEGATIVE_INFINITY*). In computations *POSITIVE_INFINITY* and *NEGATIVE_INFINITY*

are used according to mathematics rules: adding or subtracting a finite number does not change their value, multiplying them together gives *NEGATIVE_INFINITY*. Their sum is *not defined* [Nyek08].

Allowed bases

In most of the programming languages real numbers are given in decimal, integers in decimal, in hexadecimal, sometimes in octal and binary form. There are languages, where this is extended by the possibility to specify real numbers on different bases, or use other bases for any numbers. Allowed digits depend of course on the base used, so, for example, in the octal base only the digits '0'..'7', in hexadecimal the digits '0'..'9' and the letters 'A'..'F' as "extended" digits can be used. Octal and hexadecimal formats are recommended for bit pattern manipulation.

Pascal

In Pascal, integers are normally in decimal format, but can be in hexadecimal format by prefixing the *$* symbol before the digits. A hexadecimal integer can have at most 8 digits. Examples: *+124* and *−5* are decimal, *$1 $A00* and *−$FFFF* are hexadecimal integers. Real numbers can be given only in decimal base [Cat01].

Ada

In Ada, the base of the number system can be any number between 2 and 16, and real numbers could be specified also on other bases. For other basis than decimal, the following syntax must be used:

⟨*base*⟩'#'⟨*number*⟩'#'[⟨*exponent*⟩]

The base number comes before the digits of the number, which sequence is opened and closed by a # symbol. The base and the exponent is always specified in decimal format [Nyek98].
Example:

```
2#1011#              -- Binary integer.
16#F.FF#E2           -- Hexadecimal real in exponential format.
2#1.1111_1111_111#E11 -- Number and exponent have the same base,
                     -- so the last two values are both 4095.0.
```

C, C++

In C or C++ floating numbers can only be given in decimal format. The base of the integers is determined by the prefix of the number:

- *0* starts octal numbers, like: *0377*,
- *0x* starts the hexadecimals, like: *0xFF*, *0xC0B0L*,
- any other form is interpreted as decimal.

If the running environment implements `int` in two's complement format on 16 bits, the value given as *0xffff* would be -1. However, if more bits are used to store integers, the same *0xffff* literal evaluates to 65 535 [CPP98].

Java

In Java, numbers are specified like in C. It is strictly regulated, how octal and hexadecimal digits should be mapped onto 3 and 4 bits, the numeric value must be padded with 0-s from the left to 32 or 64 bits, and two's complement format must be applied [Mic03]. For example, $0xFFFFFFFF = -1_{10}$, but in Ada: $16\#FFFFFFFF\# = (2^{32}-1)_{10} = 4\_294\_967\_295_{10}$

Mathematica

It is interesting to mention, that in Mathematica any base between 2 and 36 is allowed, and beside `'0'..'9'` all 26 letters of the English alphabet can be used as extended digits. To convert the decimal integer `n` to the base `a` ($a \leq 36$) `BaseForm[n, a]` can be used, and the expression `a^^n` for the other way around.
Example: `30^^Mathematica` = 13 207 019 439 499 570. [ST96]

2.4.2 Characters and strings

For character constants usually ′ (single quote) symbols are used – like `'A'` –, some languages also allow to specify character codes directly. Certain languages like C, C++, Java, etc. define some standard named characters and introduce the '\' "escape" notion to specify them. For further details see Table 2.3.

Strings (or character sequences) are text literals constructed from the allowed character set, usually surrounded by double or sometimes by single quotes. The allowed character sets are discussed at the beginning of this chapter, for the supported character and string types of different programming languages see Chapter 5 and 6.

2.5 Comments

Program maintainability and transparency are greatly improved by commenting the source code. In these commentaries the author describes the role of each part, and the main implementation steps of complex algorithms. The compiler simply ignores normal comments, so the syntactical validity and the interpretation of the program will not be altered. Following comment methods are supported:

Code	Description
`\n`	newline
`\t`	tabulator
`\b`	backspace
`\r`	return
`\f`	form feed
`\\`	backslash
`\'`	single quote
`\"`	double quote
`\ooo`	octal character code (`0`–`377`)
`\uhhhh`	hexadecimal Unicode character (`0`–`0xffff`)

Table 2.3: Escape sequences in Java

From a mark in a special column till the end of the line

In FORTRAN, if the first character in the line is a 'C', the whole line is commented out. This principle is sometimes offered also in assembly languages. In high level languages, thanks to more unbound formats, this became deprecated. It is a problem that this way it is not possible to place a comment right next to the code, on its right side.

Special marks at the beginning and end of the comment

This is fairly common, offered by numerous programming languages, so even longer code parts could be commented out. In ALGOL 60, for example, comments begin with the **comment** keyword and go on till the ';' (semicolon) symbol. In Pascal, comments are enclosed between the symbol pairs `(*` and `*)` or { and }. This method allows inserting a comment even in the middle of a statement, like:

if *List_ptr* = **nil** *(* the list is empty *)* **then** ...

The same is true for C and C++: the comment delimiter pair is the `/*` beginning and `*/` end sequence.

As you can see, the syntax used for comments can even affect program reliability: a missing "end-comment" tag could make the compiler to ignore the whole section till the end of the next comment.

Special mark at the beginning of the comment – comment ends at the end of the line

In this case the program code is more robust, protected against possible influences of a mistake mentioned above. That is why C++ also introduced this form of comments: the special tag // marks the beginning of the comment, which ends at the end of the same program line.

In Ada or in Eiffel, program reliability is of primary importance, that is why here *only* comments from the special tag −− till the end of the same program line are allowed.

In many modern programming languages, there are tools to support *internal documentation*. These can produce an extract of the source code, which contain the method and function specifications and the contents of some – with specific symbols tagged – remarks.

In Java, comments are the same as in C++, but there is also the possibility of *documentation comments*:

/** documentation comment */

this form of documentation comments can be extracted by the *javadoc* utility.

In Eiffel, the development environment supports internal documentation, so as the documentation of a class we can query all the comments after the specification line of the attributes, the pre- and postconditions, and also the class invariant.[9]

In C#, for comments the possibilities of C++ (//, / ∗ ... ∗ /) can be used, and also the tag /// for one liner documentation comments. These documentation comments must/ought to comply with the XML standard, so documentation generator utilities can properly manage them.

2.6 Summary

In this chapter we examined the symbol sets from which lexical units can be built; how this process is standardized for each of the programming languages; how the identifiers of these languages are constructed, and which numerical-, character- and text literals are allowed.

2.7 Exercises

Exercise 2.1. Which of the following identifiers is the most readable, which would you prefer to use in your programs? Explain!

```
MaxNumberOfEmployees          max_number_of_employees
MAXNUMBEROFEMPLOYEES          MAX_NUMBER_OF_EMPLOYEES
mnoe                          m_n_o_e
u32MaxEmployees               maxEmployees
```

Exercise 2.2. Mention a programming language, in which nested comments are allowed! What could be the reason for the low number of such languages?

Exercise 2.3. In which case is it justified to use another than the decimal base for integer and real numbers?

[9] This is called the *short* form of a class, for more information see [Mey91].

2.8 Useful tips

Tip 2.1. The usable syntax of the identifiers is determined by the given programming language. For example the case, even the length and the allowed character set of the identifiers were strongly regulated by the early languages, modern languages allow more possibilities nowadays. However, naming styles and conventions for identifiers should be followed for every language. Compliant names should always be as informative as possible.

Tip 2.2. The lexical parser normally just discards everything after a comment start until a comment end sequence. If nested comments are allowed, the content of that comment can not be simple discarded, it must be examined even on a higher semantic level just to be able to distinguish between proper nesting levels.

Consider a C++ style comment notation, and inspect following nested comment situations:

```
/*  commented out
    string Str = "aslfkalfksnflkn*/*aslkfasnflkn";
Thus, the above string constant should be ignored, and
should not cause any problems related to comment deliminators!
*/

/* another comment starts here
// this nesting works, it is a different kind of comment
/*/
// so is this comment still nested or not?
```

Of course different comment constructs can usually be nested since the different comment ends will not be intermixed during lexical parsing.

Tip 2.3. Think of specifying bitmask data, like Unix style file permissions!

2.9 Solutions

Solution 2.1. The naming styles and conventions for identifiers should be followed for every language. Compliant names should always be as informative as possible. To denote multi-word identifiers, usually CamelCase is used as a practice of writing compound words such that each part begins with a capital letter. For better grouping and separation of the different parts, underscore or hyphen may be used. A good example for a more informative naming convention of identifiers is the so called ”Hungarian notation”, in which the identifier indicates its type or intended use by additional prefixes. There are actually two types of this notation (see the last two examples in the exercise), differing in the prefix encoding the actual physical data type (such as `u32` for unsigned 32 bit integer) or the logical data type or purpose (such as `max` for an upper limit).

Solution 2.2. The lexical parser normally just discards everything after a comment start until a comment end sequence. If nested comments are allowed, the content of that comment can not be simple discarded, it must be examined even on a higher semantic level just to be able to distinguish between proper nesting levels.

This needs an additional and very tolerant semantic analyzer, since the contents of the comments can be even syntactically incorrect, still the nesting levels of comments must be balanced not to cause unwanted effects such as commenting out bigger portions of the working code. This is usually too much effort for established languages, since introducing nested comment support could break backward compatibility. On the other hand, if exactly this is desired to comment out large portions of code with its own comments, other language features should be utilized (such as **#if** *0* and **#endif** of the C preprocessor).

This is the reason why nested comments are usually supported by specific or fresh languages without the need to be backward compatible, such as Rexx, Modula-2, Modula-3, Oberon, Haskell, Frege, Newspeak, D or Ocaml.

Solution 2.3. Since machine code is using a binary representation, for accessing bitwise data, numerical basis other than decimal are much more applicable. Especially the multiple of 2 as base is usually used for this kind of representation, such as the binary format to specify specific bitmasks, 8 for octets for grouping bits by 3 like the Unix style file permissions, or by 4 to address half bytes with hexadecimal numbers.

3 Control structures, statements

In imperative programming languages, statements describe the basic steps of the programs. The programmer issues them to implement the state space change of the program. Execution of the program is a sequence of these statements. Programming languages provide a variety of features to change this execution order. In this chapter, flow control structures of the imperative languages will be reviewed through their development progress, outlining the historical background and the calculation model behind them.

In this chapter we discuss flow control structures and basic statements in imperative programming languages. Statements describe the fundamental steps of programs, while control structures, often realized as statements, allow controlling the execution order of statements. To understand flow control, we must know how microprocessors in von Neumann computers work by executing machine code stored in memory in their order of occurrence. This is called *sequential control*, or sequential execution order. This sequential execution order can be modified with four fundamental control transfer statements. These modifiers are the following:

- Unconditional transfer of control;
- Conditional transfer of control;
- Subroutine call: procedure and function call (in object-oriented languages: method call). In some programming languages recursion[1] is becoming ever more popular;
- Return from the subroutine.

In addition to these there are other control structures, which are usually language dependent, but can be expressed with the four basic modifiers listed above. An example of that is multiway branching: expressed by the *case* statement in some languages and by the *switch* statement in others.

3.1 The job of a programmer

Programmers are often asked what they do at work. A detailed, technical answer is more or less meaningful depending on who has raised the question. Someone without any background in IT will perhaps just nod, and ask themselves, why society needs this. If we think about it carefully, an IT expert solves real-life

[1] For program codes the synonym self-invoking is also used for recursion. Recursion is a broader concept, as simple self-invoking, so we will use this throughout our book.

problems. This answer will make sense to everyone and will explain why society desperately needs their service. The job of a programmer is problem solving: to reach a desired target state – the solution – by changing the initial state. Finding the solution is controlled by some fixed rules. Rules describe how to get from one state to the other. The method of solving a given problem or more precisely, a problem class is called an algorithm.

An *algorithm* must fulfill some basic requirements:

- It must be described with clearly defined steps;
- It must be executable step by step;
- It must be finite (both its description and execution);
- Every description must be precise: the computer is only capable of executing precisely described steps;
- The algorithm always starts from a well defined state described by input data, and reaches a well defined endstate.

Various tools can be used to describe an algorithm, depending on the needs and resources available. These include sentence-like descriptions, flow-diagrams, D-diagrams, block diagrams and structograms (also called Nassi-Schneider diagrams), and perhaps the most important method from the point of view of this book: textual description using programming languages.

Next we provide an overview of the basic application of these tools. With the help of the Euclidean algorithm, the greatest common divisor of two natural numbers will be determined. The basic idea behind this algorithm is this: the greatest common divisor of two numbers is at the same time the divisor of their difference.

3.1.1 Sentence-like description

Sentence-like description describes the steps of an algorithm using common phrases and sentences. This method considers the least whether the words of a sentence are meaningful for a computer, although precision is an important requirement here too. The sentence-like description of the Euclidean algorithm may be formulated as follows (two numbers are given, their greatest common divisor must be determined):

1. Compare the two numbers.
2. If they are equal, the result is at hand: both numbers give the greatest common divisor of the original two numbers.
3. It they differ, the smaller must be subtracted from the greater.
4. Continue from step 1.

The description above defines with sufficient precision the method needed to compute the greatest common divisor, but it would be hard to have a computer execute the steps given in a natural language. The above algorithm may be

modified as to use the features of the module operation (to prove the equivalence of these two algorithm-variants is left for the Reader as an exercise):

1. Divide the two numbers, p with q being p the greater number. The remainder r will be between 0 and $q - 1$.
2. If r (the remainder) is 0, q is the greatest common divisor. Otherwise move the former q into p, and r into q.
3. Continue from step 1.

3.1.2 Flow diagrams

As describing the algorithm on paper, we may use a two-dimensional representation. This ensures a clear description method, but this representation is still far from the concepts of programming languages.

In flow diagrams, the execution steps are written in rectangular boxes, and execution order is determined by arrows between the boxes. Conditional branches are represented by rhombus shaped boxes: the condition is written into the box, from which two arrows can point outwards: one arrow takes the execution if the condition in the box holds, the other if the condition does not hold. These arrows can be labeled by the branch they denote.

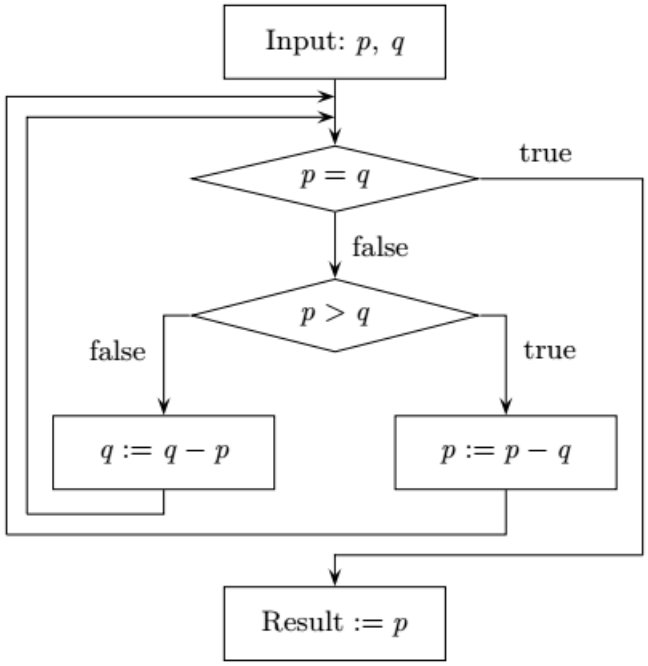


Figure 3.1: The Euclidean algorithm in a flow diagram

Since they are tangled, flow diagrams are hard to understand and are hard to implement in concrete programming languages. These programs are also difficult to modify.

The Euclidean algorithm can be described with flow diagrams as shown in Figure 3.1.

3.1.3 D-diagrams

Unsatisfied with the tangled flow diagrams, Edgser W. Dijkstra introduced a reduced description set with elemental structures which have only one possible outward (following) execution path in every case (of course, the end of the program is an exception as here the execution stops). After Dijkstra, these flow diagram elements are known as *D-diagrams*.

1. A simple operation is a D-diagram.
2. If A and B are D-diagrams, then their sequential execution (first A, afterwards B) is also a D-diagram, if c is a condition, then *"if c condition is true, then A, otherwise B"*, *"if c conditions is true, then A"*, *"as long as the c condition is true, execute A in loop"*, and *"execute A, then unless the c condition becomes true, execute A in loop"* are also D-diagrams.
3. There are no other D-diagrams (requirement: at most 1 outward execution path from all constructions).

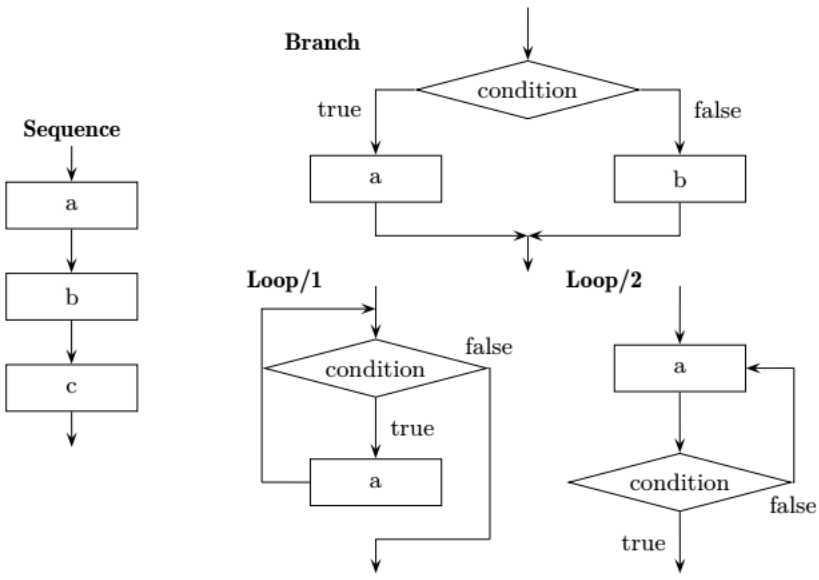


Figure 3.2: D-diagrams

D-diagrams are shown in Figure 3.2. For loops there are two types, the entry and exit controlled (pre- and post-test) variants.

The limitations of the D-diagrams have caused program complexity and difficulty to grow truly proportional to their length. The wide knowledge of these patterns makes this kind of description also significantly easier to read than traditional flow diagrams. The implementation of control structures in programs in accordance to these principles, the breakdown of the program to sub-programs, the declaration of expressions, variable types and their operations, are together all known as structured programming.

Böhm and Jacopini [BJ66] proved that every algorithm – described with flow diagrams – may be described by using D-diagrams alone. This kind of programming style is called programming without **goto** (the **goto** statement is used by many programming languages for transferring execution control).

3.1.4 Block diagrams

A simple operation written in a (rectangular) box is a block diagram. Boxes drawn sequentially will be executed in a sequence. Branches specify two boxes and a condition. The condition controls which box will be executed. For the false case of the condition, the box is not required to be given. The whole branch is boxed in, so it can be used anywhere where a box is allowed. In the case of loops an inner box holds the operations to be repeated (the loop body). This is surrounded by an outer box, which contains the loop condition.

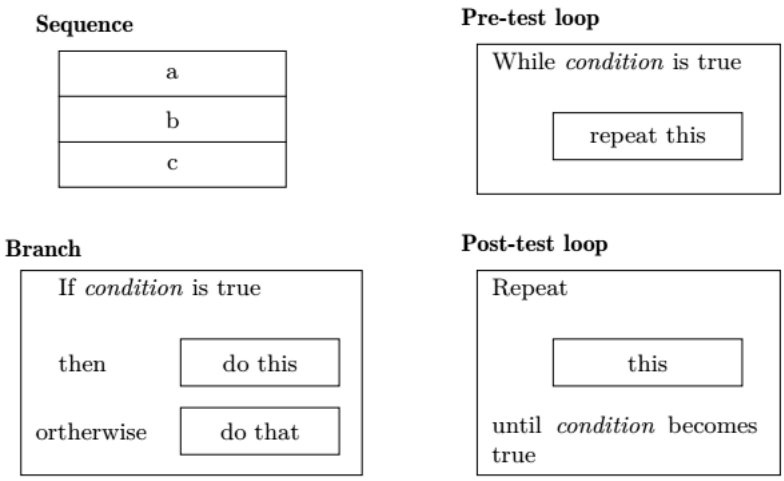


Figure 3.3: Block diagram building blocks

The building blocks of block diagrams are shown in Figure 3.3.

3.1.5 Structograms

Structograms are another tools used for describing algorithms. This involves writing operations into boxes, and arranging program structures from these boxes. Boxes holding the operations can be easily nested; nesting can demonstrate the structure of the program (similarly to the block diagrams). Their advantage to block diagrams is their restricted form, which is generated more easily by programs or word processors.

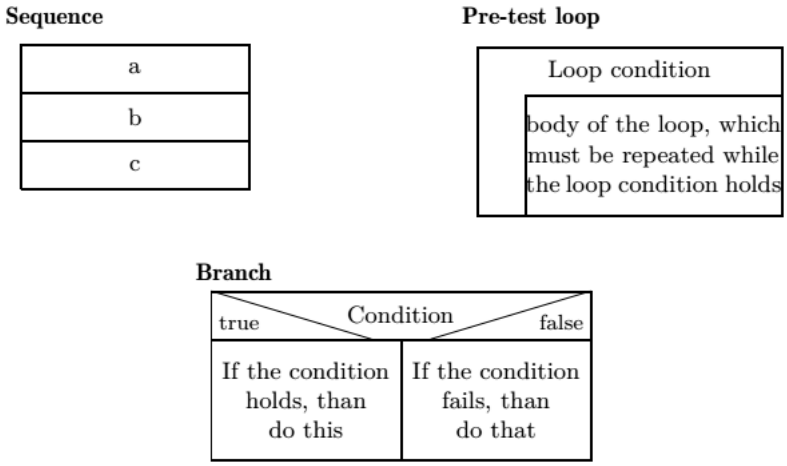


Figure 3.4: Structogram building blocks

The building blocks of the structograms are shown in Figure 3.4.

3.2 Implementation in assembly

Programming languages are often categorized as low or high level languages. Low level languages are usually created for a specific computer or architecture: statements here correspond to the instruction set of the microprocessor on the target architecture. These languages are called assembly. Assembly statements relate directly to machine code instructions. Compilation from the assembly programming language to machine code is done by an assembler. The instruction set of a high level – algorithmic – language is independent of that on any given computer architecture: before execution, a compiler must create assembly and machine code from the source code.

Next we present the implementations of the Euclidean algorithm in Pascal and LMC languages.

3.2.1 The solution in Pascal

Following is a Pascal implementation of the Euclidean algorithm (a basic knowledge of the Pascal programming language is a prerequisite for understanding the implementation to follow):

```
procedure gcd(p, q: integer; var result: integer);
begin
    while q > 0 do
    begin
        if p > q then p := p − q
        else q := q − p;
    end; (* while *)
    result := p
end; (* gcd *)
```

3.2.2 LMC

The Little Man Computer (LMC) is an instructional model of a von Neumann architecture computer, created by Dr. Stuart Madnick at MIT. Components of this simplified model are:

- The number system used for data consists of three decimal digits, representing integers in the range −500 to 499. Negative values are encoded in ten's complement, which is computed for negative numbers by adding 1000 to it and which leaves non-negative values unchanged.
- Mailboxes: this is the working memory of the model. The address range is limited to two digit decimals (00 – 99), each Mailbox can hold one unit of data or instruction code.
- Instruction Location Counter: a two digit display with the address of the next Mailbox to evaluate. Programs start at the address 0, with the push of the Reset button. Leaving the end of the address range $(99 + 1)$ causes abnormal program termination.
- In and Out Baskets: the input and output communication ports of the model. Data can only be read in from In, and can only be output to the Out Basket, one at a time. The handling of multiple subsequent data is implemented in a First In First Out (FIFO queue) fashion. Signed data is converted automatically to the internal ten's complement representation at Input and vice versa at Output.
- Calculator: temporary data storage for arithmetic operations. Its value range is the same as of Mailboxes (−500 – 499); the supported operations are addition and subtraction. Numerical under and overflow lead to abnormal program termination.
- Little Man: works inside the above defined architecture and performs the following operations rigorously:

1. Reads the current value of the Instruction Location Counter.
2. Goes to the mailbox with that number and reads its content.
3. Pushes the Counter incrementer button to advance (by one) the value of the Instruction Location Counter.
4. Interprets the last read mailbox content and executes its value as an operation code.
5. If not stopped by the operation before, continues with step 1.

Note that incrementing the Instruction Location Counter occurs before executing the current operation, so that branching can land at the desired location.

The above components are illustrated in Figure 3.5.

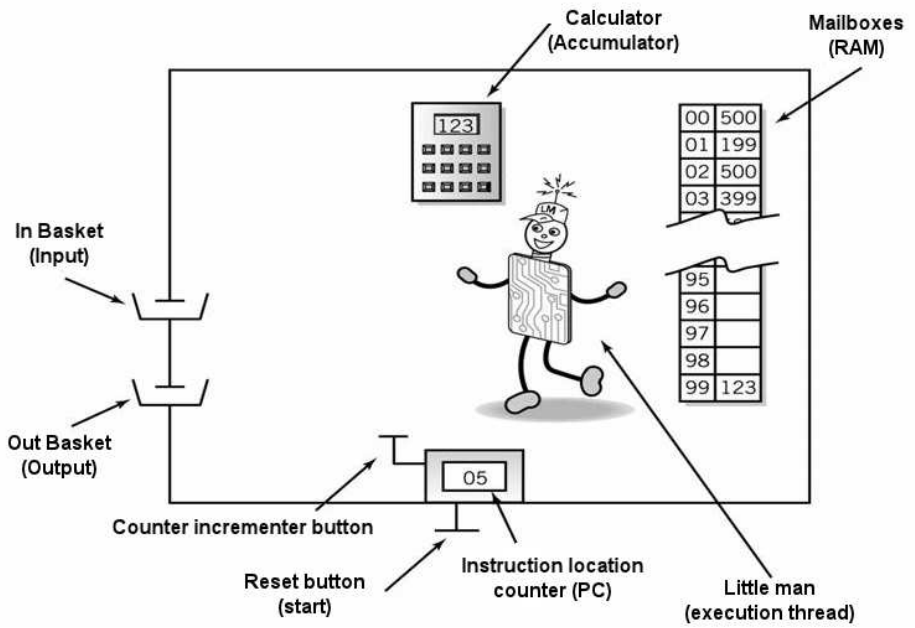


Figure 3.5: System architecture of LMC

The above architecture resembles the functional organization defined by von Neumann, as there is a control unit (the Little Man and the Instruction Location Counter) to execute instructions, an arithmetic unit (the Calculator) to perform calculations, and a memory (the Mailboxes) to hold both programs and data (this is known as the stored program concept) in a linearly addressed (with a two digit sequential number) location space.

Execution of an LMC program needs the following preparations:

1. Instructions (machine code) must be loaded into the mailboxes, starting from address 00.
2. Input data must be placed into the In Basket in proper order.
3. By pressing the Reset button, the execution starts, the Little Man wakes up and performs his duty.
4. The result will appear in the Out Basket.

The instruction codes of the LMC tell the Little Man what to do. These codes are the machine code of this architecture and are stored as ordinary data within the Mailboxes. Therefore each instruction is made of 3 decimal digits, the first representing the command to perform, and the next two digits addressing the mailbox for the operand of the command (this is called indirect addressing).

Machine code	*Mnemonic code*	*Instruction description*
000	HLT	Halt: stops the execution and gives the little man rest.
1xx	ADD xx	Adds the value of mailbox xx to the current value in the calculator (result stays in the calculator).
2xx	SUB xx	Subtracts the value of mailbox xx from the current value of the calculator (result stays in the calculator).
3xx	STO xx	Stores the actual value of the calculator into mailbox xx.
4xx	STA xx	Stores the address portion (last 2 digits) of the actual value of the calculator into those of mailbox xx.
5xx	LDA xx	Loads the actual value of mailbox xx into the calculator.
6xx	BRA xx	Branch (unconditional): sets the instruction counter to the given address (xx). That is, mailbox xx will be the next place for the execution to continue.
7xx	BRZ xx	Branch if zero (conditional): if the actual value in the calculator is zero, then sets the instruction counter to the given address (xx), otherwise does nothing.
8xx	BRP xx	Branch if positive (conditional): if the actual value in the calculator is zero or positive, then sets the instruction counter to the given address (xx), otherwise does nothing.
901	INP	Takes the input value from the In Basket, and puts it into the calculator.
902	OUT	Copies the actual value of the calculator to the Out Basket.
	DAT xxx	Assembler instruction to load the value xxx into the next available mailbox.

Table 3.1: LMC instruction set

Encountering a non-defined instruction code leads to abnormal program termination. Due to this and the von Neumann stored program concept, special care must be taken to prevent program execution to reach pure data within the memory.

As seen by the DAT instruction, LMC also supports assembly level programming. In such cases, the LMC assembly program is written in plain text source format, using only mnemonics for the instructions. Each line can have a label at the beginning, which can be used as target for other instructions. Comments are also allowed after the instruction operand at the end of each line. Compilation from LMC assembly to machine code is the task of an LMC assembler.

The LMC assembly implementation of the Euclidean algorithm takes the following form:

```
        INP         ; 00 901 input p
        STO p       ; 01 308 store p
        INP         ; 02 901 input q
        BRZ end     ; 03 705 while q > 0
        BRP loop    ; 04 810
end     LDA p       ; 05 508 result is p
        OUT         ; 06 902
        HLT         ; 07 000
p       DAT         ; 08
q       DAT         ; 09
loop    SUB p       ; 10 208 compute q - p in calculator
        BRZ end     ; 11 705
        BRP loop    ; 12 810 if q > p, q := q - p
        ADD p       ; 13 108 else
        STO q       ; 14 309
        LDA p       ; 15 508
        SUB q       ; 16 209
        STO p       ; 17 308 p := p - q
        LDA q       ; 18 509
        BRA loop    ; 19 610
```

The above code is an instructional example to demonstrate the features and capabilities of low level programming languages.

3.2.3 Comparison of the solutions in LMC and Pascal

There are several basic differences between Pascal and LMC programs:

- Pascal (and other high level language) programs are portable. They can be adapted without much modification, but programs in low level languages are bound to specific hardware: conversion to a new microprocessor architecture requires the whole program to be rewritten for the new processor.

In some cases, simulation of the other processor is supported on hardware level, but this decreases the efficiency of the program.
- Low level programs describe machine instructions, like moving values from and to registers and memory locations. In high level languages, higher level abstraction is used, e.g. variables for memory/register abstraction. In high level language programs there is no need to deal with the register or memory management of the executing computer. These tasks can be left for the compiler.
- Low level programs are harder to read. The LMC code example above was short and well commented, the implemented algorithm is widely known, but modification of a some 10 000 line assembly program without proper commenting may cause considerable difficulty.

Finally, low level programs have readable source code for programmers which for execution is turned into machine code by the compiler, that is, it becomes a sequence of bits. Programming directly in machine code (hard "coding") is extremely difficult, but before of low and high level languages were developed, this was surely the only option.

3.3 An elementary approach

By examining algorithm descriptions, mathematicians have designed various calculation models which require a minimal abstraction from the hardware and software side, yet allow an easy algorithmic formulation and description of the problems. Next we present one such method, characterized by two main features:

- Potentially infinite memory to store variables;[2]
- Potentially infinitely large memory for the storage of the program itself.

Only natural numbers (arbitrary large, including zero) are regarded as data. This is not a limitation, as most of the data types can be converted into this set. Input and output are not a concern: input and output of the algorithms are assumed to be stored in memory. This calculation model forms the basis of the so-called *while-programs* model, in which the following higher level operations are implemented.

3.3.1 Elements of the while-programs

While-programs consist of four basic statements:

1. Resetting the value of a memory compartment:

   $$x := 0$$

[2] "Potentially infinite" means for variables or for their representing memory that by enumerating memory compartments as $r_1, r_2, r_3, \ldots r_n$, n – the number of available memory compartments – can have an arbitrary large value according to the program's need.

2. Incrementing the value of a memory compartment:

   $x := x + 1$

3. Decrementing the value of a memory compartment (by definition decrementing $x = 0$ gives $x - 1 = 0$):

   $x := x - 1$

4. Looping until the value of the memory compartment x equals that of the memory compartment y:

   while $x \neq y$ **do**
   ... (Loop body to execute.)
   wend

These statements are very simple, still many things can be expressed with them.

3.3.2 Higher level operations

Now we move on to examine how useful and well known elements may be implemented as while-programs. By this we mean to demonstrate the great expressive power of the while-program language.

1. Loading an arbitrary positive ($n \geq 0$) constant into a register ($x := n$):

   $x := 0;$
   $x := x + 1;$
   ...

   The incrementing statement $x := x + 1$ must be executed n times in all.

2. Copying the value of variable x into variable y ($x := y$):

   $y := 0;$
   while $y \neq x$ **do**
   $y := y + 1;$
   wend

3. Adding the value of variable y to that of variable x ($x := x + y$):

   $s := 0;$
   while $s \neq y$ **do**
   $x := x + 1; \quad s := s + 1;$
   wend

4. Addition ($z := x + y$):

   $z := x;$
   $z := z + y;$

   See the examples for loading a constant into a variable, and adding a value to that variable.

5. Subtraction ($z := x - y$, where the result will be 0, if $x < y$): Implementation is like that of the addition (see above), but instead of incrementing $x := x + 1$, decrementing $x := x - 1$ is applied.
6. Now that addition and subtraction have been defined, based on them, multiplication and modulo division can be easily implemented (with multiple addition and subtraction).
7. Values of logical data types can also be implemented. Let 0 and 1 be their possible values. 0 denotes the logical false, 1 the true value.
8. Conditional statement with while-programs can be implemented like this: The statement to be expressed is the following:

   if $\langle condition \rangle$ **then** S **end**

   where $\langle condition \rangle$ is the value of a logical expression (0 or 1), and S is the statement to execute depending on the condition. The equivalent while-program would be this (s is an auxiliary variable):

   $s :=\langle condition \rangle$;
   while $s \neq 0$ **do**
   S;
   $s := 0$;
   wend;

9. The extended conditional statement can also be expressed with while-programs: The statement to be expressed is the following:

   if $\langle condition \rangle$ **then** S_1 **else** S_2 **end**

   where $\langle condition \rangle$ is the value of a logical expression (0 or 1). The equivalent while-program would be this:

   $s_1 :=\langle condition \rangle$;
   $s_2 := s_1$;
   while $s1 \neq 0$ **do**
   S_1;
   $s_1 := 0$;
   wend;
   while $s_2 \neq 1$ **do**
   S_2;
   $s_2 := 1$;
   wend;

10. Calling non-recursive subprograms can be implemented by copying its statements. Recursive subprograms must first be converted – with the help of a stack – to non-recursive, than the previous case – regarding non-recursive subprograms – must be followed.

3.3.3 Considerations

This elementary approach of a computation model is highly theoretical and serves mainly educational purposes. However, most implementation methods introduced here can be found elsewhere in real life applications, such as in the micro-codes of RISC processors.

The simplicity of the instruction set of this model allows highly efficient implementations. This explains why the seemingly complex use of a relatively few basic statements can outperform simple solutions based on more common and widely applicable statements.

3.4 Control approaches

Program execution can be carried out in different ways according to the description of the used programming language. In the following we describe some control approaches used most widely in programming languages. Some of these approaches are mutually exclusive, others may be applied simultaneously. The merging of these approaches is a challenging task and a widely researched field.

Our focus here is on those approaches only which are used in execution control. More specifically, in this chapter we will deal with control structures of imperative languages, declarative (logical) and functional programming languages will be handled in other chapters.

3.4.1 Imperative programming languages

The *imperative programming approach* is an abstraction of real computers, based on the model of von Neumann computers and Turing machines with the concept of registers and memory. Variables and the assignment is a programming abstraction of memory content change. In the imperative approach a variable denotes a memory compartment. These variables are named, can be assigned a value, which may then be changed.

The names and values of the variables in the program, the actual execution point (the actual operation being executed) together form the *program state*. A program being executed may be characterized by *sequence of state-transitions* (Turing-machine model). Transitions between the states can be described as a sequence of assignments and control statements (see relevant sections in [Wir73]).

Based on this, imperative programs can be described as consisting of states and state-transition statements. Methods can be defined as the abstraction of a sequence of state-transitions. Imperative programming applied together with methods is called *procedural programming*.

3.4.2 Declarative and functional languages

In the imperative approach, programs consist of sequences of implementing statements targeting a certain goal; by contrast, declarative and functional languages follow a significantly different approach.

In *declarative (and logical) languages*, in contrast to the imperative languages, no method is given as how to solve a task; only the problem to be solved is specified. As the emphasis is on exact description, specification needs the most attention. Finding the solution is the job of the runtime environment. Such languages are, for example, Prolog and SQL, which will be discussed in other chapters.

The *functional model* also requires exact specification, but the goal of the program is to compute a – mathematical – function. The result of the program is the result of this function. In this model, some elements common in the imperative model (e.g. outputting partial results) are considered side effects during the computation of the function. For more in details, see [McC85] and [Bow02].

3.4.3 Parallel execution

Program execution can be performed by one execution thread, but some programming languages support parallel execution: in this case, execution is performed by multiple threads. In declarative languages this problem will not occur, since it is the job of the runtime environment to handle and implement the given task; parallel execution can be at most suggested, for example, in the form of a synchronization specification. In imperative languages, parallel execution can be easily modeled: by specifying multiple execution threads with various program execution points.

Execution control is handled and supervised usually by the help of the runtime environment (the operating system). The operating system ensures the scheduling of each execution thread (execution state). Parallel programming theory is discussed in more detail by Manna [MP91]. Practical tools, such as the PVM library, are described in [PVM02].

3.4.4 Event driven programming

There was a paradigm shift in the 1990s regarding the interactions between the runtime environment and the control structure of a program. This is not generally applicable for all the programming languages – programming environments –, but must be mentioned as far as regarding execution control is concerned.

In the 1990s the *algorithmic execution approach* was common. The basic idea behind it was that after starting the program, it executes specific tasks (this can be anything from compiling another program, to managing janitor records), and if needed, input will be acquired from the user.

The general structure of the program code following the *algorithmic model* is the following:

```
Main program:
  Variable declarations
  Initialization
  Prepare data to be processed
  Input data from user
  Process data
  Output result
  Continue processing
  End of program.
```

From the 1990s onwards, mainly due to the spreading of graphical windowing systems in interactive programs, the algorithmic execution approach was gradually replaced by the *event driven programming approach*. After an initialization part (which includes, for example, the display of a graphical user interface) the program gets notified by the operating system in the form of so called *events* about what is happening in the environment (the user pressed a key or moved the mouse; or another program signaled the deletion of a file). The general structure of the program code following the *event driven model* is the following:

```
Main program:
  Variable declarations
  Initialization
  Loop until last event read
    Input next event from the operating system
    Process event according to its type
  End of program.
```

The main difference between the two models is how the services of the environment – such as the windowing system, or the operating system – are used. In the algorithmic model, the problem-solving algorithm is emphasized, the environment is used only to access specific services (such as input user data from the keyboard). In the event driven model, the environment (such as the windowing system) plays s passive service provider role, as well as an active control-relevant role: it monitors user actions and notifies the programs in the form of events, managing their global control.

The structure in the event driven program model above is characteristic of most of the event driven techniques applied nowadays in programming languages. In some languages other different solutions are introduced for supporting event driven programming, these use the language features that were developed earlier to support the algorithmic model. The COBOL programming language, standardized in the 1970s supports event driven control with its *DECLARATIVES* exception handling mechanism (see Section 8.2.2.) by assigning *DECLARATIVES* paragraphs to each event.

3.5 Programming languages examined

In the following section we provide a brief description for the programming languages examined this far, which – in our opinion – best demonstrate the development in the field.

3.5.1 Sentence-like algorithm description: COBOL

The COBOL programming language had been standardized (in 1966 and 1974) before the appearance of the block structures: most part of the COBOL programs today are developed according to the COBOL-74 standard from 1974 (a smaller part is represented by the programs according to the COBOL-85 standard from 1985). The approach of the COBOL language should be seriously considered because of its simplicity and widespread use (until 2002 more than half of all the program lines used worldwide in programs were written in COBOL, this ratio will probably not change significantly even in our decade; perhaps only a slight decrease of its share over 60 percent can be predicted). Here the relevant parts of the COBOL-74 standard will be discussed.

A COBOL program consists of a sequence of English *sentences* according to the rules specified by the standard: these are the statements of the language. On the next higher level the COBOL program consists of *paragraphs*. The paragraph is a sequence of statements (or sentences according to the COBOL terminology). The end of a paragraph is indicated by the beginning of the next paragraph (or by the end of a so called section – see later). Paragraphs in COBOL are like procedures without parameters in other known programming languages. Execution starts at the first sentence of the paragraph and runs until its end (unless something other is specified at calling – see later). COBOL does not support recursion or parameterized paragraphs. Paragraphs of the COBOL program form so called *sections*. A section can be called as a subprogram. Executing a section means executing its paragraphs, starting from the first paragraph, running through even multiple paragraphs until the `EXIT` statement, which stops the execution of the section and returns control back after the calling point. COBOL programs are made of a sequence of sections.

In the next example two sections are defined. If the section `OTHER` gets the control, its statements are executed in their order of appearance. The execution continues after reaching the paragraph `PARA21` with its statements, then with the paragraph `PARA22`. At the end of this paragraph, reaching the `EXIT` statement causes the return from the execution of this section after its calling point. Paragraphs after the `EXIT` statement (such as `PARA2n` in the example) will not be executed automatically. However, such paragraphs can be called from the other paragraphs of the section. It is possible to call paragraphs from another section, but care must be taken, because the so called *overlay* sections are only stored in memory during their execution, and calling a paragraph in such a section from another section would cause a runtime error. This is used to aid virtual memory

management, which was useful, if it was not supported by the operating system and there was only a limited amount of system memory available for programs. Nowadays rapidly decreasing memory prices make this solution obsolete.

The following code snippet demonstrates the structure of a COBOL program:

```
ONE SECTION.
* This is a section in the program
      Statements.
      ...
 PARA11.
* This is the first paragraph of the section.
      Statements.
      ...
 PARA12.
* This is the next paragraph of the section.
      Statements.
      ...
      EXIT
 PARA1n.
* This is the next paragraph of the section.
      Statements.
      ...
OTHER SECTION.
* This is a section in the program
      Statements.
      ...
 PARA21.
* This is the first paragraph of the section.
      Statements.
      ...
 PARA22.
* This is the next paragraph of the section.
      Statements.
      ...
      EXIT
 PARA2n.
* This is the next paragraph of the section.
      Statements.
      ...
```

3.5.2 Structured programming: the Pascal language

The Pascal programming language was designed by Niklaus Wirth with the support of the features of structured programming in mind. Language design started in 1968. In education it was first introduced in 1972 in ETH Zürich. The definition of the standard Pascal language was published in 1973 [JW74].

The language defined enough features to implement programs designed with the methods of structured programming, but some extensions (like the Turbo Pascal environment from Borland, or the ST Pascal/68000 on ATARI ST computers) provided a richer set of control structures than the language standard required. The usage of these extensions greatly reduces the portability of the programs, but all of these can be substituted by other standard elements.

There is another Pascal language implementation also worth mentioning: the UCSD P-System, which was developed at the San Diego University in California in the late seventies. The main point of this system was system independence: the Pascal compiler produced the so called *P-code*,[3] which will be executed on a given hardware architecture by a P-code interpreter *virtual machine*. This system was developed in Pascal, and offered a complete event driven graphical user interface and numerous useful libraries for creating user friendly programs. Back then this system could not spread industry wide because of its pure performance. But the universities saw the potential in the educational use of the possibilities of this system – and at that time, at universities the very popular Apple-II computer family was suitable for running it. The basic idea was reused by the developers of the Java language in the mid 1990s, but for a much more efficient hardware – and the efficiency and success of this solution is now pretty obvious (like the virtual machine for P-code, the Java virtual machine is also based on stack-based technology).

The Pascal language supports block structures and the constructs of structured programming. It serves as an ancestor of numerous programming languages, such as Modula-3 [BW96] and Ada [Nyek98]. This chapter discusses the control structures of Pascal, and the extensions introduced by Modula-3 and Ada.

3.5.3 Portable assembly: the C language

After some well-known representatives of the Pascal-derived languages, the language elements of the C language and its descendants will be introduced, illustrating the crystallization of certain language concepts from the beginning of the 1970ss (the birth of the C language) until today (up to C#[4] [Sch02]). By examining the C-derived languages, control structures of three programming languages will be introduced – those of C, Java and C#.[5]

[3] The description of the P-code and a complete Pascal compiler can be found in [PD82]. The Pascal compiler is introduced, line by line, in this book, while details about the runtime environment can be found on the Internet on sites focusing on compilers, or in books on the web. This book was a classic in university and college lectures for about a decade, mainly due to its practical implications.

[4] To be pronounced as C sharp.

[5] The J# language, which is the newest Java implementation of Microsoft could be considered here too. However, as this language has the same control structures as the Java 1.1.4 version, it will not be discussed separately. In general, everything mentioned here about the Java language also holds for this language.

Choosing C as the common ancestor seems to be a natural choice. The incidence of the Java language and its applied security system is a serious reason for this language to be chosen. Choosing the C# language can be surprising at first considering the incidence of the C++ language [Str00]. But we chose the C# language instead of C++, as C# supports not only an equivalent, but a richer set of control structures than C++. Other serious argument in favor of C# is that in industrial applications it is simpler than C++. The same reasoning can be made, like mentioned before in the case of the Ada language, but the situation is not that taut. The Ada language – or more precisely, its predecessor – was originally developed in parallel by competitors under a multistage R and D competition organized by the USA government. The aim was to choose the most suitable language for their purposes. Once the Ada compiler and method-based approach were developed, upcoming projects were decisive factors for government-funded software projects and purchases [Nyek98] for many years. Nevertheless, as Ada gave very complex definitions for types and hid many details – like implementing the *access* types instead of pointers –, it created a psychological barrier for programmers. Noticing this in the middle of the 1990s the US government re-evaluated the usefulness of Ada, and added C++ to the list of "allowed" programming environments.

For many programmers, C++ is not less complex than Ada: Microsoft was aware of the problem and after its defeat in a trial about the Java language,[6] it began to develop and implement the C# language (making it an international standard, helping its spread this way).

In this chapter we focus on C# with regard to control structures, as C# provides a broader set of control structures than C++.

3.5.4 Everything is an object: the Smalltalk language

The Smalltalk language was developed by the researchers of Xerox PARC. It is an object-oriented language (in its terminology *sending messages* to objects is used instead of method calling): in Smalltalk everything is an object. As the development of a unified system was the goal, the operating system and the runtime environment were included in the form as special predefined objects and as some features, from which the most important are the following:

- Memory management (with automatic garbage collection);
- File management with special purpose objects;
- GUI management – providing a windowed, graphical user interface with mouse support;

[6] Microsoft supported the Java language with some extensions, like the *delegate possibility* introduced for procedural parameters. Sun, on the other hand saw in these extensions made by Microsoft and favored by their programmers the loss of the portability of Java, or the possibility of this, so Sun sued Microsoft. The lawsuit was won by Sun. Following this, in 2001 Microsoft introduced the C# language and the *.NET environment*, which was an outstanding basis for it.

- Keyboard management for standardizing user input;
- A real object-oriented debugging system, which helps Smalltalk programs to access themselves as objects.

3.5.5 Other examined programming languages

In this chapter many examples are taken from various other languages. These languages are the following:

Modula-3

the latest extension of Pascal and Modula-2 [BW96].

Ada

a safe language developed by the U.S. Department of Defense [Nyek98].

Eiffel

an object-oriented programming language developed by Bertrand Meyer, which is less used by the industry, but still widespread enough to be worth mentioning.

CLU

developed in MIT lead by Barbara Liskov, mainly for educational purposes. Significantly contributed with its approach to the development of the X Window System. For more details about this language, see [Lis81].

FORTRAN

a language designed for numerical calculations (about the same age as COBOL): although there are many active FORTRAN programs, and a large number of arithmetic, algebraic and other mathematical libraries were developed in FORTRAN, it has lost its importance by now [Fort03].

3.6 Assignment, arithmetic statements

In this section, the development of the control structures are shown while focusing on the possibilities and main characteristics of some carefully chosen languages.

3.6.1 Features of COBOL

In COBOL originally – for the sake of simplicity of the compiler – there was no assignment statement, which could assign a composite – such as bracketed – arithmetic expression value. The solution of the problem must be decomposed.

Loading constant values into variables can be achieved with the *MOVE* statement. The same statement loads the values of one variable into another. Consider the following example:

```
MOVE 23 TO A.
MOVE B TO C.
```

The first statement moves the constant numeric value 23 into the variable A. The second statement copies the value of the variable B into the variable C. COBOL sentences must be closed by a period.

Four basic statements can be used as arithmetic operations:

```
* Add A to B, store the result in C.
ADD A TO B GIVING C.
* Subtract A from B, store the result in C.
SUBTRACT A FROM B GIVING C.
* Multiply A by B, store the result in C.
MULTIPLY A BY B GIVING C.
* Divide A by B, store the quotient in C.
DIVIDE A BY B GIVING C.
```

The *GIVING* clause specifies where the result should be stored (in our examples always variable C was specified as the target). The *GIVING* part with the variable name can be omitted: in this case the standard specifies, which variable will hold the result (for example the statement

```
ADD A TO B.
```

will store the sum of variables A and B in B).

Newer versions of COBOL support the *COMPUTE* statement to evaluate more complex expressions. For example:

```
COMPUTE A = ( A + B - C / ( A * B ) - A * B ).
```

The value of the right side expression above will be stored in the variable A (which was specified on the left hand-side). Please note that only the four basic operations are allowed here: arithmetic functions (like sinus or cosinus or extracting a root) are not allowed – these must be implemented from elementary steps by the programmer, for example, by a decomposition of these functions.

3.6.2 Simple assignment: the Pascal language

The syntax of the assignment in Pascal is very simple. The statement is introduced by the := (the *colon-equals*) sign. On its left hand-side the name of the variable must be specified, to which a new value should be assigned, given on the right hand-side. All the Pascal-derived languages use the same token for assignment. In these languages the = (equal sign) is used to check the equality of two values as a comparison operator. Pascal does not allow multiple assignment,[7] other languages – like CLU – do allow it.

The general form of the assignment is as follows:

⟨*variable reference*⟩ := ⟨*expression*⟩;

By executing this statement, first the right hand-side expression gets evaluated, then its value is stored into the variable referenced on the left hand-side. An important characteristic of the assignment is the type compatibility of the right hand-side value with the type of the left hand-side variable. For the data types, see Chapter 5.

3.6.3 Assignment in C

Assignment together with function call belong to the expression-statement topic in C and the C-derived languages. Assignment is an operation which copies the value of the expression on its right hand-side (the right value) into the variable referenced on the left hand-side (a left value).

Please note that in languages derived from the C language the assignment usually functions as an *operator* (can appear in any *expression-statement*), so multiple assignment in these languages is a natural language feature. Also C-derived languages have a specialty: for assignment they use a simple equal sign instead of the := (colon-equals) sign. In these languages, the comparison for equality is denoted by two equal signs.

As the assignment is an operator, it has a return value: this will be the value from its right side. Considering

```
A = B = 23;
```

the statement assigns the value *23* to the variables *A* and also to *B* (performing a multiple assignment). Regarding its binding, this statement is equivalent to the following:

```
A = (B = 23);
```

In practice this means that *23* is assigned to *B*, and this will be also the result of the expression containing the assignment. The bracketed (and so first evaluated) expression has the value *23*, which will be stored into variable *A*.

[7] Multiple assignment is the statement when a value is stored into multiple variables at once.

In these languages, regarding expressions the assignment can be used together with other operators. For example the statement *A = A + 4 ;* can be written in the shorter *A += 4 ;* form.

3.6.4 Solution in Smalltalk

In Smalltalk the assignment is the only operation which is not performed by sending messages (method call) [GR83]. The syntax is like that in Pascal. The

```
A := 34.
```

statement, for example, ties the variable `A` to the object `34` (always by modifying the references).

But the expression-statement has to be interpreted differently. Objects and messages to be sent to them must be specified. The

```
1 + 2.
```

statement sends the message named `+` (calling this method) to the object 1 integer number, the parameter of this message is the `2`. The period at the end of the line is the closing character for Smalltalk statements.

3.6.5 Multiple assignment and the CLU language

CLU, as a language designed for research and programming-educational purposes must be mentioned because of *multiple assignment*, as this language allows the assignment to have on the left and right sides multiple (but the same count on both sides) variables and sources (for more details about this language, please see [Lis81]).

In CLU exchanging the values of the variables `X` and `Y` is a simple statement like this:

```
X,Y := Y,X
```

This code will work properly. In other programming languages (such as Occam), the auxiliary variables needed here are no concern for the programmers.

3.6.6 The role of assignment in programs

An assignment looks like a very simple statement, but it is the most important operation in today's programming languages: by examining source codes of some programming projects, the – not surprising – result was that more than half of the statements in these source codes were assignments. (For this examination a more than quarter of a million lines long COBOL source of an application

program system,[8] a derivative of the Linux 2.4 operating system, a multimedia server and a coding-decoding program were taken into account). This result is not so surprising, considering that only assignments take real steps toward the solution of the task, loops, for example, only specify the execution of some assignments or subprograms on a given state-space and range for multiple times.

3.6.7 The empty statement

The *empty statement* (which does nothing, but could have special roles, like `NOP` in Motorola 68000 assembly) can be seen as an assignment with the same variable on its left and right hand-sides. Most of the programming languages define an empty statement separately: in Ada the **null** statement, in Pascal and the C-derived languages a simple semicolon (although in the latter – namely C, C++, Java, C# – it has no major role). In Eiffel the semicolon can be used, but has no real role. In COBOL the empty statement is denoted by the line *`NEXT SENTENCE`*.

At first sight, the empty statement may seem useless (the `NOP` statement of the MC 68000 processor also cleared its pipeline, but in higher level programming languages programmers do not have to consider this). Nevertheless, as a statement it can be tagged with a label.

The designers of the CLU language developed their positions based on the above – namely that the empty statement is useless, so there is no such in CLU.

3.7 Sequence and the block statement

Sequence is a basic control structure implemented as the sequential execution of statements. In practice the order of execution for the statements is the same as their order of appearance.

The sequence as a control structure is supported from the beginning in all the imperative programming languages. As assignments describe a state change in the state-space of the program, more complex progress (changing more components of the program state) can be achieved by sequences.

In COBOL sequence can be implemented by writing statements into lines one after the other. Each statement of the sequence is closed by a period. COBOL follows the ”one line one statement” concept with the attenuation that long statements can be broken into multiple lines to support better program readability.

Most of the programming languages use the semicolon for a sequence of statements (seemingly to denote the end of the statement, but in practice the situation is more complex).

[8] In these COBOL programs, not only the `MOVE` statement, but also the value changing arithmetic statements were considered as assignments, since in other languages there are no such restrictions regarding assignments, as in COBOL.

The block statement is a special statement for grouping a section of code together to be handled as only a single statement. Blocks consist of possible declarations (if supported) and a sequence of statements.

To examine the historical development of the sequence as a control structure in programming languages, the following aspects should be considered:

- Can the sequence or block be empty?
- Is the semicolon for closing or for separating statements?
- Is there a way to declare block statements?
- Can a block statement have declarations, and if it can, whereabouts in the block (anywhere or only in the declaration part of the block)?

These are general aspects. By examining the relevant (see later) features of the programming languages listed as examples, the features of the given language will be presented. The specific elements of the given language will be examined, but missing or irrelevant/unsupported features will be left out. This way our book becomes more readable, and we can focus only on the essentials. It is clear that if a language does not support block statements, it would be useless to examine whether block statements could have declarations in it or not. The same is true for loops and branches discussed later. The Reader will be able to answer all the above questions after the detailed introduction of the elements in the specific languages (if something is not mentioned under the discussion at a given language, it is not supported by that language, unless noted otherwise).

In the following section, the possibilities of the *language family founder programming languages*, that is COBOL, Pascal and C – will be discussed in detail. In the case of these languages we intend to be comprehensive. In the more recent languages, we will highlight the new, innovative solutions/elements. Only those new elements – regarding control structures – will be discussed, which differ from that of the original language. If there has been no major change since the original language, this will be not noted separately. If the Reader wishes to gather more detailed information about a particular language, its official manual should be consulted. This will be indicated as literature reference in the appropriate places.

3.7.1 Block statement in Pascal

Control structures in Pascal have been defined – except for **repeat** loops – as having only one statement as their loop body. If multiple statements should go where only one statement is allowed by syntax rules, the block statement must be used. This denotes a sequence of statements framed by the **begin**. . . **end** keywords. The **begin** keyword starts the block, the **end** closes it. This causes the block to act as a single statement for the compiler, and executes as the sequence of its enclosed statements.

Reading Pascal programs and sources in derived languages, one might note the many semicolons, which are used to *separate* statements, as opposed to

the C-derived languages, where semicolons close the statements. In Pascal the statements of a sequence enclosed into a block statement must be separated by semicolons, but between the last statement and the closing **end** the semicolon is not needed, as there is nothing more within the block to separate. If there was an extra semicolon, it would not be an error, but would mean that the compiler assumes an empty statement between the closing semicolon and the **end** statement of the block. However, but this is without any consequences.

The general form of the block statement is the following:

begin
    $\langle statement_1 \rangle$; $\langle statement_2 \rangle$; ... ; $\langle statement_n \rangle$
end

These statements can be any statement, even inner blocks. Blocks in Pascal can be optionally *nested*.

The statements of the block get executed sequentially, in the order of their appearance. In Pascal declaring local variables within normal blocks is not allowed. This is only supported in procedures, functions, and in the main program.

3.7.2 Break with the tradition of Pascal: the Ada language

One salient feature which shows the break from the traditions of Pascal is the change of the role of the semicolon: in Ada, semicolons do not separate, but close the statements [Nyek98]. Therefore there must be a semicolon after every statement, even before (and after) the closing structured statements **end loop** and **end if**.

In Ada, the form of the block statement is also clearer than in Pascal or Modula-3. Consider the following example:

```
Swap:
declare
    AuxVariable: Integer;
begin
    AuxVariable := I; I := J; J := AuxVariable;
end Swap;
```

As can be seen, between the **declare** and the **begin** keywords new variables or if needed, new procedures or functions may be declared. Elements declared here can be seen only within the given block. Blocks can be named by labels before them – this name can be specified after the closing **end** keyword. This improves program readability a lot. Ada has introduced a special closing keyword for every structured statement: for example, for the **if** statement it is **end if**, for **loop** it is the **end loop**, for the multiway branching **case** it is the **end case** keywords (to close them). This leads to clearer, more readable source code for both the compiler and the programmer.

In Ada, to allocate memory for temporary big data structures (from the beginning of the block until its end) the block statement has been a very useful feature.[9] Nowadays this role tends to be neglected due to the effective compilation methods (thanks to data-flow analysis, the compiler knows all the information and features to minimize the allocation time for variables, and thus it can manage the allocation more efficiently than the programmer, since it has the most detailed information and special knowledge about the target architecture).

3.7.3 Characteristics of the C language family

The C and its derived languages have block structures: in place of the **begin**–**end** keywords in Pascal, opening and closing curly braces can be used. In C this only serves to structure statement groups, whereas in C++, C# or Java its role is the same as in Ada. Within a block defined in this way, new declarations can be placed, like in the **declare** block of Ada. Besides better readability, as in Ada, in all languages but C (C++, C# and Java), local declarations are allowed in *any* blocks, in any places. Unlike Ada, inner blocks are not divided into a declaration part and (following) statement body in these languages (except for C where no declarations at all are allowed in nested blocks): declarations can be placed anywhere between statements. This has the result that variables declared in blocks only take up memory until the end of their scope, that is, the end of their containing block most of the time. Previously to allocate memory for temporary big data structures, declaration at any place within the block was a good method. Nowadays this role is neglected because of effective compilation methods, as has been described above.

Semicolons in C, as in Java, C++ and C#, have a *statement closing* role: they close the statements, and do not separate them. An exception is the block statement, which, unlike the Pascal **begin**–**end** keyword pairs, is denoted by { and } braces in these languages. Block statements do not need to be closed by a semicolon.

3.7.4 Block statement in Smalltalk

The block is a very efficient construct in Smalltalk: *it represents executable code as an object*. Within the conceptual system outside Smalltalk, a block is an unnamed function (method), which can be passed to other objects as a message parameter, or it may receive a message too. Of course in Smalltalk a block usually has a name. The block can later be also referenced by name, which leads to the execution of the specific code:

[9] This answers the question if in Ada there is a block statement, and if it may have local declarations.

Consider the following example, a Smalltalk block to write to the screen:

```
|output|
. . .
output := [ :x | Transcript showCR:x ].
```

Block declarations are framed by square brackets, after the block name and the following := signs. The declaration of the block includes a list of parameters the block can receive, and after a horizontal line the body of the block follows. The block above sends the `showCR` message (to perform output) to the `Transcript` (predefined) object representing the screen, passing the received parameter to be output.

This block can be used in the following way:

```
output value:'hello'
```

In Smalltalk the use of blocks is very versatile. For example a push button on a GUI can be defined so that it can receive a block as a parameter, which will receive a message with a given name and parameter type at the push of the button.

3.8 Unconditional transfer of control

The unconditional control transfer is the earliest, and also the most controversial control transfer structure already introduced in FORTRAN0. In FORTRAN0 the statements may be marked with numerical labels,[10] and with the *GOTO* statement control could be passed from any point within the program to another arbitrary statement (marked with a numeric label).

What argument can be raised against the *GOTO* statement? As a matter of fact, all the following issues:

- Using it uncontrolled in modern languages makes the job of the compiler hard, even impossible, as it makes no sense to jump between independent blocks. Just think of it, what would happen to the execution stack in such a case? What would be the values of local variables? In a recursive function what nested level should be jumped into? A jumbled program (the so called ”spaghetti code”) makes it hard to understand its goal.
- The *GOTO* undermines the application of most of the correctness proving tools. Correctness proving methods work usually by dividing the program at the control structures, formulating invariants and with the help of these – applying logical methods – the correct operation of the program is proved. By *GOTO* statements and labels (if these can be applied anywhere) it is very hard to formulate invariants, so its usage is not recommended.

[10] This is where the original concept of the BASIC language comes in, to have all the statements numbered.

- The precise description of the meaning of programming languages containing the *GOTO* statement is much harder (but not impossible).
- This is not how the human mind works.

In some languages *GOTO* was forced to be used (as when exiting a loop), but most of the languages offer special solutions instead (see the **break** statement in C later).

Please note that these are general aspects. By examining the relevant features (to be discussed later) of the programming languages listed as examples, those features will be presented which differ from those already discussed at their ancestors (e.g. Modula-3 as an improvement on Pascal). Elements typical for the given language will be introduced, but aspects will not be discussed, which are not applicable, or which have already been described in detail at the ancestor of that language.

3.8.1 The features of COBOL

COBOL supports unconditional transfer of control by its *GOTO* statement. The general form of the *GOTO* statement is the following:

GOTO ⟨*paragraph name*⟩.

This causes execution to be transferred to the given paragraph. COBOL programs may be coded without the use of *GOTO*, as the language supports a wide variety of features to implement on any kind of program structures. COBOL also supports the usage of *computed* *GOTO* statements in the following form:

GOTO ⟨*list of paragraph names*⟩ *DEPENDING ON* ⟨*variable*⟩.

Executing this statement, the value of the variable after the *DEPENDING ON* acts as an integer index to select the target paragraph from the given list to jump to. A peculiarity of the COBOL language is the *ALTER* statement, which supports the making of self-modifying programs. *ALTER* behaves differently at its first execution than later. This statement will not be described here in detail as most of today's COBOL systems does not support it, or only to the detriment of efficiency. The main reason for this is that to implement self-modifying code, the so called code segment storing the executable form of the program must be modified. Its memory location is managed by most operating systems as read only, partly because of security reasons,[11] and partly because of code segment sharing: if multiple instances of a program are active, at runtime they can use the code segment shared together, as only data segments storing their data must be allocated separately and managed for every program.

[11] Security reasons here imply the protection of the code segment, which is not to be overwritten, not even accidentally, because this is often caused by memory management errors. With this solution, modern operating systems support debugging.

3.8.2 Unconditional transfer of control in Pascal

The Pascal language supports the unconditional transfer of control by its **goto** statement. For this, *labels* must be declared in the program, and only these can be targeted by jumping.

Pascal defines further restrictions on language and on compiler levels with regard to the **goto** use, which could be explained as a result of the block structure.

As a result of this, unconditional control transfer is only allowed within blocks. Some compilers allow jumping from a block into a containing outer block with **goto**, but jumping into a more nested inside block is usually not allowed. This is not surprising as the content of the execution stack coherent to the block structure will be the same by jumping within the same block; it may be reduced by jumping into an outer block, but by jumping into an inner block the values of the variables in that block must be appended, which cannot be performed in a meaningful way.

3.8.3 Modula-3: end of GOTO

In Modula-3 there is no unconditional transfer of control: there is no **GOTO** statement. The designers closed the discussion about whether the **GOTO** is needed or not. Their opinion is that the **GOTO** statement is unnecessary.

In Modula-3 a statement to transfer control back from procedures and functions is the **RETURN** statement. This statement ends the execution of the called subprogram (procedure or function), and transfers control back after the calling point. Behind **RETURN**, a value can be specified, which will be used by functions: this is how the return value of that function is specified. Please note that in Pascal there is no such a statement: a subprogram (procedure or function) ended after the execution of its last statement; the return value of the function had to be specified by an assignment to the name of the function (which had not triggered the return from the function, only after executing its last statement).

Another major instrument of control in Modula-3 is exception handling, which will be discussed in a later section of the book (see Section 8.3.5.). Instead of the **GOTO** statement in many places in this and in other languages, exception handling can be used to solve the task.

3.8.4 Special control statements in C

The C language supports the unconditional control transfer with the **goto** statement on language level. For this a label must be declared within a function by stating the name of the label followed by a colon. With the **goto** statement this

label can be a jump target. As the scope of such labels is only valid for their containing functions, only transfer of control within functions is supported.

label:
  . . . . statements . . .
  goto *label;*

For more complex control transfer solutions, the *setjmp* and *longjmp* functions from the C libraries can be used. These are not part of the language, but are usually implemented within the C libraries.

The *setjmp* call saves the actual size of the execution stack (but not its content) and stores the current execution point into its parameter. The *longjmp* function implements the unconditional control transfer. As its parameter, a state descriptor saved by *setjmp* must be specified. Executing *longjmp* resizes the execution stack to the saved size by *setjmp* (reducing the stack if needed), and passes execution after the referenced *setjmp* statement. This is not a direct part of the C programming language, but because of its popularity, knowledge and reference, as an example, is certainly useful. In this way jumping between functions is possible but uncontrolled.

The **continue** statement can be used in loops: it interrupts the execution of the loop body, and forces the next iteration of the loop to begin (unless the exit criteria gets fulfilled).

The **break** statement exits the innermost loop immediately. Execution continues after the loop. This **break** statement was used to leave branches at multiway branchings started with the **switch** keyword. Execution continued also on the next statement after the **switch** block.

The **return** statement exits a function and returns after the calling point. After the **return** statement the return value of the function can be specified. Functions without return values (declared with **void** return types) are known as procedures by other languages. When returning from such a procedure, no return value is allowed after the **return** statement.

3.8.5 New features in Java

Java programmers can use the same control structures as mentioned in the C programming language (except for the **goto** statement). A significant difference is that Java supports the logical (**boolean**) data type, so conditions must be all of logical type.

Java also supports labeled statements, but these can be targeted only by **break** and **continue** statements to specify which loop they refer to (that is, not only the innermost loop is affected). In this case after the **break** and **continue** statements the label (for example of a **for** loop) must be specified, which determines the loop these statement are referring to.

Another special feature of Java as opposed to C, is the *checking of the accessibility of statements*. For example, statements after an infinite loop are

useless, because they will never get executed. Thanks to modern compilation techniques, the Java compiler recognizes statements which can never be reached. Such statements cause the Java compiler to give error messages, warning the programmer about possible logical or algorithmic errors. For example, the following code

```
while (false) { x = 5; }
```

causes a compilation error as the loop body never gets executed.
In contrast, this statement

```
if (false) { x = 5; }
```

does not cause such an error message: the compiler simply leaves it out from the generated code. This is how conditional compilation works. Quite often some code must only be executed in debug mode (e.g., if a logical constant named *DEBUG* has the value **true**). Such a solution is shown in the following example:

```
static final boolean DEBUG=true;
...
if (DEBUG) { System.out.println("debugging here"); }
```

This method ensures that by setting the *DEBUG* constant to **false**, the debugging message will be left out of the production (not for debugging) version of the program.

3.9 Branch structures

In the first widespread programming language – FORTRAN0 – statements could be labeled by numbers, logical and arithmetic branching, and an unconditional control transfer (`GOTO`) statement was offered for the programmers.

The syntax of the *logical branching* is the following:

`IF (`⟨*logical expression*⟩`) L1, L2`

Its meaning: if the value of the logical expression is true, execution continues on the line numbered `L1`, otherwise on `L2`.

The syntax of the *arithmetic branching* is the following:

`IF (`⟨*arithmetic expression*⟩`) L1, L2, L3`

Its meaning: if the value of the arithmetic expression is negative, execution continues on the line numbered `L1`, if zero on `L2`, if positive on `L3`.

These structures resulted in unreadable code, so programmers searched for newer, more usable solutions. As a result of this, the *if-then-else branching structure* was introduced in ALGOL-60, which can be found in nearly all the later languages.

Regarding branching in programming languages the following aspects will be examined:

- Is there two-way branching within the language?
  - If yes, does the language clearly define where the ELSE branch belongs (avoiding the *dangling* **else** problem)?
- Is there multiway branching in the language (**case** or **switch** structure)?
  - If yes, what type can be the selector expression (on which the branching occurs)?
  - Must all the values of the selector expression appear in branches?
  - Can a default branch be specified which will be executed if the value of the selector expression does not match any branches?
  - Can the same value be specified in multiple branches, and if yes, what is the meaning of this (would this make the program nondeterministic)?
  - Is the order of the branches fixed, or can these be in any order?
  - Does the execution continue at the end of one branch in the following branch? This is also a language design philosophy question: *are branches independent program parts*, from which only one or none gets executed (this is the case with Pascal and also C#), *or are they entry points*, from which the following branches get executed in the order of their appearance. This is how the **switch** statement in C, C++ and Java works, but not in C#.
  - What can be used as target values for branches; how can these values be specified?
    - Only one value can be specified at once.
    - Multiple values can be enumerated.
    - Can a value-range be specified for shorter description?

These are also very general aspects. By examining the relevant features of the programming languages listed as examples, those features of the given language will be presented which differ from that of its already discussed ancestors. Elements typical for the given language will be introduced, but aspects, which are not applicable, or which have already been described in detail at the ancestor of that language, will not be discussed.

3.9.1 Branching in COBOL

Branches in COBOL can be coded with the *IF–THEN–ELSE* statement. For this structure consider the following example.

```
IF ⟨condition⟩ THEN
      ⟨statement1⟩
      ⟨further statements⟩
   ELSE
      ⟨statement2⟩
      ⟨further statements⟩
      ⟨last statement⟩.
```

The meaning of the statement is the usual: if the condition after the *IF* gets fulfilled, the statements behind the *THEN* get executed until the *ELSE*; if the condition is false, the statements behind the *ELSE* get executed (if this *ELSE* branch exists, it may be omitted).

The end of sentence period is there only after the last closing statement of the whole *IF* statement. Other statements within it must not be closed by a period (otherwise the COBOL compiler will interpret this as the end of the IF statement at the wrong statement). The *ELSE* part can be omitted, but if specified, it cannot be completely empty. To have it explicitly do nothing, the *NEXT SENTENCE* can be used as an empty statement.

3.9.2 Conditional statement in Pascal

Pascal offers two types of conditional statements: one is the traditional **if**–**then**–**else** statement, the other is the multiway branching depending on the value of a discrete type (the **case** statement).

The general form of the **if**–**then**–**else** structure is the following:

if ⟨*logical expression*⟩
then ⟨*statement*$_1$⟩ **else** ⟨*statement*$_2$⟩*;*

The meaning of the statement is clear: if the value of the logical expression after the **if** is *true*, the statement following the **then** gets executed, otherwise the statement following the **else** gets executed. Only one statement can be specified after the **then** and the **else**. If multiple statements should go there, they must be enclosed with a **begin**–**end** block statement. The whole **else** branch with the ⟨*statement*$_2$⟩ part can be omitted. To understand the statement separator role of the semicolon consider following two statements:

if ⟨*condition*⟩ **then** ⟨*statement*⟩*;*
if ⟨*condition*⟩ **then** *;* ⟨*statement*⟩

The meaning of the first is obvious: if the condition after the **if** holds (it is true), the statement after the **then** gets executed. The semicolon at the end of the line separates the **if** structure from the next (in this case empty) statement. In the second example there is an empty statement after the **then**, which is separated by the semicolon from the next statement, which will always be executed regardless of the value of the condition after the **if**.

Now let us move on to the *dangling else* problem, which implies an inaccuracy in the definition of Pascal (but also in C and C++). It concerns the nested **if** statements, where it is not always clear to which statement the **else** branch actually belongs. Consider the following example:

```
if (a <= b) then
   if (a < b) then Write("a < b")
   else Write("a = b");
```

In this example, the question is: to which **if** statement does the given **else** branch belong. Indentation suggests that it belongs to the inner one, but the language does not define this. As a result, there is only one sure solution: every statement, even the empty one, should be packaged into a **begin**–**end** block, so that the **else** branch of the inner and of the outer **if** statements may get obvious. In the above example and everywhere in Pascal, the rule applies, that the **else** part always belong to the most inner **if** statement (the same is true for C and C++).

3.9.3 Multiway branching in Pascal

Multiway branching in Pascal is implemented by the **case** statement. The general form of the statement:

```
case ⟨discrete expression⟩ of
   ⟨selector_1⟩: ⟨statement_1⟩;
     ...
   ⟨selector_n⟩: ⟨statement_n⟩;
   else ⟨statement⟩
end
```

The working of the statement is clear: first the expression after the **case** gets evaluated, then execution continues at the selector with the same value as evaluated. If the value of the expression after the **case** does not match any of the selector values listed, control is transferred to the statement after the **else** (if this branch is omitted, control is transferred to the next statement skipping all the branches). The discrete (integer or enumeration) type of the expression after the **case** can be *integer*, *char*, *boolean* or any enumeration or range type. As selector values for the alternative branches enumeration and also subdomains may get specified. For example `23..44` is a valid selector for a **case** branch.

As for reasons of efficiency many compilers allocate a jumping table for the whole possible value range in the executable code, and compress the size of the table by using relative relocation addresses for jumping stored as short words, this also severely limits the size of the **case** structure. The aforementioned Pascal-P4 for example did not support **case** branching based on integer values because of this limiting relative jumping on 16 bits. A more concrete and narrower value range had to be specified there.

3.9.4 Safe branching: innovations of Modula-3

In the Pascal language, many errors derived from the carelessness of the programmers as they forget to introduce **begin**–**end** blocks when expanding **if** statements or **while**/**for** loops with new statements. To eliminate errors of such origins, Modula-3 changed these structures: on the one hand, it introduced the mandatory closing **END** statement for these structures, on the other hand, it permitted the use of multiple statements (separated with semicolon) in these control structures without framing them with **BEGIN**–**END** statements. As a consequence, conditional branching has now the following form:

IF ⟨*logical expression*⟩
    THEN ⟨$statement_1$⟩; ⟨$statement_2$⟩; ⟨$statement_3$⟩; ...
    ELSE ⟨$statement_{E1}$⟩; ⟨$statement_{E2}$⟩; ⟨$statement_{E3}$⟩; ...
END;

The **BEGIN**–**END** block statements can fully be omitted here. For the sake of simplicity in nested branching, Modula-3 has also introduced the **ELSIF** keyword. For example, the following statement structure would become unreadable if nested. This could easily happen for multiple branching conditions:

IF ⟨$logical\ expression_1$⟩
    THEN ⟨$statements_1$⟩
    ELSE IF ⟨$logical\ expression_2$⟩
            THEN ⟨$statements_2$⟩
            ELSE ⟨$statements_3$⟩
        END
END

In Modula-3 this can be done in a simpler way with the **ELSIF** keyword instead of the **ELSE IF**:

IF ⟨$logical\ expression_1$⟩ **THEN** ⟨$statements_1$⟩
ELSIF ⟨$logical\ expression_2$⟩ **THEN** ⟨$statements_2$⟩
ELSE ⟨$statements_3$⟩
END

The general form is the following:

IF ⟨$logical\ expression_1$⟩ **THEN** ⟨$statement_1$⟩; ⟨$statement_2$⟩; ...
ELSIF ⟨$logical\ expression_{A1}$⟩ **THEN** ⟨$statement_{A1}$⟩; ⟨$statement_{A2}$⟩; ...
ELSIF ⟨$logical\ expression_{B1}$⟩ **THEN** ⟨$statement_{B1}$⟩; ⟨$statement_{B2}$⟩; ...
..
ELSE ⟨$statement_{E1}$⟩; ⟨$statement_{E2}$⟩; ⟨$statement_{E3}$⟩; ...
END;

There can be any number of **ELSIF** branches. As a general rule, if multiple branches are true in this structure at the same time, the first appearing true

branch will be executed. Please note that many researchers try to weaken this requirement by allowing the execution of any randomly chosen true branch instead of the first one. With this the program could have some degree of non-determinism, or by parallel execution all the true branches could start to execute in parallel. However, the Modula-3 language does not support these. We have mentioned them only to point at a research topic worth pursuing. With this step – introducing a closing element (in this case the **END** keyword) for each of the structured statements, including the **IF** as well – the designers of the language have also solved the problem of dangling ELSE.

3.9.5 Safe CASE in Modula-3

The **CASE** statement also allows the dropping of the **BEGIN**–**END** framing statements. The general form of the **CASE** gets modified as shown in the following example:

CASE ⟨*discrete expression*⟩ **OF**
    ⟨*selector*$_1$⟩ => ⟨*statements*$_1$⟩;
      ...
  | ⟨*selector*$_n$⟩ => ⟨*statements*$_n$⟩;
  | **ELSE** *statements;*
END

The selector parts (⟨*selector*$_1$⟩...⟨*selector*$_n$⟩) can have one or multiple constant values, or a value range. The vertical line can appear also on the beginning of the first branch, like at others, but causes no change in execution.

The selector values need not cover all the possible values of the discrete expression. If there is no matching selector to the actual value of the discrete expression, and also the **ELSE** branch is omitted, a runtime error will occur.

3.9.6 Branch structures in C

In C branching can be implemented with the **if**–*then*–**else** structure. The general form of this is the following:

if *(*⟨*expression*⟩*)* ⟨*statement*⟩*;*
/* or */
if *(*⟨*expression*⟩*)* ⟨*statement*$_1$⟩*;* **else** ⟨*statement*$_2$⟩*;*

If the value of the expression in parentheses after the **if** is *true* (that is not zero), then ⟨*statement*$_1$⟩ gets executed. If there is an **else** branch, and the value of the expression is *false* (that is zero) ⟨*statement*$_2$⟩ gets executed. There can only be one statement for each of the branches. To specify multiple statements at once, those must be enclosed in curly brackets defining a block statement.

3.9.7 Multiway branching in C

Multiway branching in C is implemented by the **switch** statement:

```
switch (⟨discrete type expression⟩) {
    case ⟨value_1⟩: ⟨statements_1⟩; /* break; */
    ...
    case ⟨value_n⟩: ⟨statements_n⟩; /* break; */
    default: ⟨statements⟩;
}
```

The expression in parentheses after the **switch** gets evaluated, and the system looks for the label following the **case** with the same computed value to transfer control to (if there is no such label, execution continues on the **default** branch). Please note that execution *falls through* from every **case** branch to the following, unless it encounters somewhere a **break** statement, which causes it to leave the **switch** block and continue at the statement after that. This can cause many program errors. This is why there is a **break** statement in every branch – commented out – in the above template. That is how a statement sequence could belong to multiple **case** branches, as shown in the example below.

```
switch (⟨discrete type expression⟩) {
    case ⟨value_1⟩:
    case ⟨value_2⟩: ⟨statements_2⟩; break;
    case ⟨value_3⟩: ⟨statements_3⟩; break;
}
```

In this example if the value of the discrete type expression is $\langle value_1 \rangle$, or $\langle value_2 \rangle$, then the $\langle statements_2 \rangle$ part gets executed, and reaching the **break** statement causes the execution to leave the **switch** block and continue after it. If the value of the discrete type expression is $\langle value_3 \rangle$, execution is transferred to $\langle statements_3 \rangle$. The last **break** statement is unnecessary but allowed, as there are no more branches, so execution cannot ”fall through” (nevertheless, specifying a **break** there is a safer programming style, as by adding a new branch after the last one, the separating **break** would not be forgotten since it is already there).

3.9.8 Multiway branching in C#

In C# the **switch** (multiway branching) statement was modified in a way to support the more frequent case: the C# compiler gives an error if a **case** branch is not closed by a **break** statement. The reason for this is that in most cases execution should not ”fall through” to the next branch, and *this is characteristic*. That is why the compiler requires all branches to be closed by a **break** statement. If there is the intention to continue execution with the next **case** branch, the **goto** statement can be used (which was directly introduced for this purpose, whilst

from Java, the closest language, this was left out). The **goto** statement can be used like in C, but there are several major restrictions:

1. No jumping inside a block is allowed (like into the body of a loop from outside of the loop);
2. No jumps between classes.

The designers of C# find the only place for **goto** as legitimate: to ensure this "fall through" behavior (this requires no new labels, as those of the **case** branches can be used as **goto** targets). The use of **goto** is allowed also in other structures, but structured programming offers every feature needed not to use it anywhere else, except for the above mentioned case for the **switch** statement.

3.9.9 Conditional statement in Smalltalk

In Smalltalk the conditional statement is a message sent to the logically `true` and `false` values (as objects) resulting from logical expressions. These object can handle the `ifTrue:` and `ifFalse:` messages.

```
(a > 5) ifTrue: [output value:'value of a is greater than 5'].
```

The above statement evaluates the (`a > 5`) logical expression, which can result in a logical true or false value. The result object receives the `ifTrue:` message with a block as parameter, and will execute this block if its value is true, calling the procedure declared before and showing the message `value of a is greater than 5` on the screen. The opposite of `ifTrue:` is `ifFalse:`. These can be used in Smalltalk to implement the traditional `if-then-else` structure. Consider the following example:

```
(a > 5) ifTrue: [output value:'value of a is greater than 5']
     ifFalse: [output value:'value of a is NOT greater than 5'].
```

3.10 Loops

Regarding loops in programming languages, the following aspects will be examined:

- Is there a way to implement loops with unknown iteration count?
  - If yes, is there a pre- and post-test loop?
  - Does the exit condition have to be of logical type, or can it also be of any other type?
- Does the loop body have to be in a separate block, or in a separate procedure? (See the paragraphs in COBOL).
- Is there a way to implement loops with known iteration count?

  - If yes, what type can be the loop variable?
  - What expressions can be used to specify the initial value, the limit and the increment for the loop variable?
  - When will the iteration count be determined?
  - Can the value of the loop variable be modified within the loop, and if yes, what are the consequences for the loop?
  - What is the scope of the loop variable, is its value defined after exiting the loop?
  - How often are the value of the limit and the increment evaluated?
- Is there a general loop in the language (like in Ada the **loop**–**end loop** structure)?
- Are there statements to leave the loop, which can be used anywhere within the body of the loop? (Like **break** or **exit**.)
- Is there a way to force the next loop iteration? (Like, for example, in C **continue**.)
- Are traversing (or iterator) loops supported? (Like, for example, in C# the **foreach** structure.)

These are also very general aspects. By examining the relevant features of the programming languages listed as examples, those features of the given language will be presented which differ from that of its already discussed ancestors. Elements typical for the given language will be introduced, but those will be not discussed which are not applicable, or which have already been described in detail at the ancestor of that language.

3.10.1 Loops in COBOL

To call subprograms, the *PERFORM* statement can be used. It has numerous forms, and loops can also be implemented with these statements. To call paragraphs the *GOTO* statement can also be used (this is unconditional transfer of control, as described earlier: the COBOL language supports this usage). The simplest form of *PERFORM* is the following:

```
PERFORM paragraphname.
```

This executes the statements in the given paragraph, and after its last statement, control is transferred back to the next statement after the *PERFORM*.

To execute a sequence of paragraphs, the following form can be used:

```
PERFORM paragraphname1 THRU paragraphname2.
```

This transfers control to the paragraph named `paragraphname1`, and runs all the paragraphs up to and including `paragraphname2`. Obviously the `paragraphname2`

paragraph must be after the `paragraphname1` (any number of other paragraphs can reside between them).

The *PERFORM* statement can be used for looping. The simplest form of a *PERFORM* loop with fixed number of iterations is the following:

```
PERFORM paragraphname n TIMES.
```

Above `n` is a numerical value. The loop executes the given paragraph `n` times.

To implement loops with exit conditions the *PERFORM* statement can also be used. The form for this is the following:

```
PERFORM paragraphname UNTIL condition.
```

Its meaning: execute the paragraph with the given name until the condition behind the *UNTIL* gets true.

For example, the next code snippet reads in data in the `READ1` paragraph until (signaling valid data) the `R-FLAG` is set to zero:

```
MOVE 1 TO R-FLAG.
PERFORM READ1 UNTIL R-FLAG = ZERO.
```

The form of a counting loop in COBOL is the following:

```
PERFORM paragraphname VARYING loop variable
      FROM value BY increment UNTIL condition.
```

Its meaning: start the loop variable with the value specified after the *FROM*, execute the paragraph with the given name until the condition behind the *UNTIL* gets true. After every iteration the value of the loop variable gets updated with the given increment.

As shown above, the COBOL language requires *the loop body to be within a paragraph*; its name will be specified after *PERFORM*.

3.10.2 Loops in Pascal

The standard Pascal language offers numerous forms of loops for programmers. The **while** is the pretest loop statement. The general form of the **while** is the following:

while ⟨*logical expression*⟩ **do** ⟨*statement*⟩

This has the meaning: as long as the logical expression in the **while** is true, execute the statement after the **do**. After the **do** only one statement can be specified; to use multiple statements, they must be enclosed in a **begin**–**end** block. The following will cause an infinite (never ending) loop:

while *true* **do** ⟨*statement*⟩

true in Pascal is the logical true constant value: an always true expression.

The posttest loop variations in Pascal are the **repeat**–**until** pair. The general form of this is the following:

repeat
⟨*statement*$_1$⟩ ; ⟨*statement*$_2$⟩ ; . . . ; ⟨*statement*$_n$⟩
until ⟨*logical expression*⟩;

The loop body[12] is between the **repeat**–**until**, which is a sequence of statements to be executed repeatedly. This statement first executes the loop body, and then, if the value of the logical expression after **until** is false, the loop body executes again and the exit condition gets evaluated again. This repeats until the logical expression after the **until** returns a logical *true* value. With this loop statement an infinite loop can be implemented like this:

repeat
⟨*statement*$_1$⟩ ; ⟨*statement*$_2$⟩ ; . . . ; ⟨*statement*$_n$⟩
until *false;*

The original form of Pascal designed by Wirth had only the above mentioned two forms for loops with unknown iteration count. Numerous compilers support the so called **do**–**loop** loop, which can have multiple statements between the **do** and the **loop** statements. To leave the loop the **exit** statement can be used. Note that without the original standardization, this statement can have different formats for different compilers.

With regards to the so called *counting loops* with known iteration count (in advance), in Pascal the general form of the counting loop is the following:

for ⟨*loop variable*⟩ := ⟨*value*$_1$⟩ **to** ⟨*value*$_2$⟩ **do** ⟨*statement*⟩;

or

for ⟨*loop variable*⟩ := ⟨*value*$_2$⟩ **downto** ⟨*value*$_1$⟩ **do** ⟨*statement*⟩;

where ⟨*value*$_1$⟩≤⟨*value*$_2$⟩ must hold. In the first case, by entering the loop, the loop variable starts with ⟨*value*$_1$⟩, the one statement loop body after the **do** (if necessary, this can be a block statement) gets executed, the value of the loop variable gets incremented by one, and all this repeats until this value exceeds the ⟨*value*$_2$⟩ specified after the **to**. The second form essentially works the same way: here the loop variable starts from ⟨*value*$_2$⟩ and counts down (with a step of −1) repeating until reaching ⟨*value*$_1$⟩ (more precisely: until stepping bellow it).

Counting loops in Pascal have a number of shortcomings, which got improved (or differently solved) in the subsequent languages:

- The loop variable can be changed only by +1 (using **to**) or by −1 (using **downto**).
- The value of the loop variable after the loop is undefined (the actual value depends on the compiler, and the code it has produced). This is

[12] The code to be executed in a loop.

a consequence of the scope of the loop variable extending to the whole block declaring it, and it is not restricted to the loop only.
- Changing the loop variable is not allowed, but this is not checked by the compiler.
- The limit for the loop variable gets evaluated only once, before entering the loop, not before each iteration. This helps to produce efficient code, since the number of iterations is known in advance (and will not be changed by modifying the value of the loop variable). However, this can be a disadvantage for programmers routinely working in other languages, as if they continue to modify the value of the loop variable from within the loop, it will definitely be a problem.

3.10.3 Modula-3: safe loops

The **WHILE** loop structure known from Pascal has also changed as it is shown at the branches: there is no need to frame multiple statements for the loop body with the **BEGIN**–**END** pair. It is enough to write the loop body into the structured loop construct and close it with an **END** tag at the end. The **REPEAT**–**UNTIL** loop can be used the same way as in Pascal.

The form of the counting **FOR** loop has been simplified by leaving out the framing **BEGIN**–**END** pair, and has obtained greater expressive power as the increment may be any number (thus, there is no need for negative and positive increments to have different statements, like **to**, or **downto** in Pascal).

The form of the **FOR** loop in Modula-3 is the following:

FOR ⟨*loop variable*⟩ := ⟨$value_1$⟩ **TO** ⟨$value_2$⟩ **BY** ⟨*step*⟩
DO
  ⟨$statement_1$⟩; ⟨$statement_2$⟩; . . . ⟨$statement_n$⟩
END

The significant change is the appearance of the **BY** part, where the increment can be specified (a value of zero results in an endless loop). This together with the increment can be omitted, and the default value of +1 will be used. The designers of the Modula-3 also made an effort to resolve the problem of the loop variable present in the Pascal language:

- The loop variable within the **FOR** statement is visible only in the loop body, and local to it.
- The declaration of the loop variable happens with its occurrence (naming) in the **FOR** statement.
- After the end of the loop the loop variable ceases to exist, so its value cannot be referenced outside the loop.
- The loop variable is not object to change within the loop. Assignment to it causes compilation error.

3.10.4 Loop–end–exit loops

Modula-3 standardized the **DO**–**LOOP**–**EXIT** loop briefly mentioned with reference to Pascal.

The general form of the loop is shown in the following example.

LOOP
⟨*statements*⟩
. . . **EXIT** . . .
⟨*statements*⟩
END

Statements between the **LOOP** and the **END** are running in an endless loop. The loop can be left with the **EXIT** statement. The **EXIT** causes the loop to immediately terminate, and execution continues after the **END**. With **EXIT** every loop can be left (even **WHILE** loops), but to comply with the principles of structured programming, it is recommended to limit the use of the **EXIT** statement only for the **LOOP**–**END** loops. The **EXIT** statements usually occur in **IF** statements within **LOOP**–**END** loops: this controls the exit by satisfying certain conditions. It is possible to use **LOOP**–**END** loops without **EXIT** statements; this will cause an infinite loop. Infinite loops usually do not make much sense, but in some cases (such as in a communication program where the task is to receive and forward communication packages without stopping) this could have its own meaning. These loops will run until the program has been forced to terminate from the outside e.g. due to the shutdown of the computer.

3.10.5 Features of the Ada language

Similar control structures, but with different syntax were introduced into Ada too, thus extending the loops of Pascal. Since, however, these are primarily syntactical innovations, they will not be discussed in detail here.

Here our focus will be on novelties reflecting better design solutions, and having more expressive power.

The counting **for** loop plays a similar role in Ada, as in the earlier described programming languages. Ada defines the characteristics of the counting loop very thoroughly:

- The loop variable is a local *constant* of the loop;
- This variable is declared in the loop statement, with the most appropriate and narrowest type according to the value range of the loop;
- The loop variable (and a value range covered) must be of discrete type;
- The value range to cover by the loop will be computed only once, before entering the loop;
- Aa an obvious consequence of the above, there is no sense in talking about the defined value of the loop variable after leaving the loop, since it is

not visible there. If there is a variable with the same name declared in the containing block of the loop, the loop variable hides it (to access the outer variable, its qualified name with the name of the declaring block must be used, but bear in mind that it has nothing to do with the loop).

The general form of the counting loop is the following:

for ⟨*loop variable*⟩ **in** ⟨*value range*⟩ **loop**
  ⟨*loop body statements*⟩
end loop;

As shown, the loop variable has to cover a value range. This can be the range of the whole value set (referenced in Ada as ⟨*type name*⟩'*range*) of existing data types (such as integers, or an enumeration type), or just a subset of a value range: for the range *1–N* (where *N* is a positive integer) specifying the *1..N* value range to cover it. With the **reverse** keyword the iteration direction on the given value range can be reversed. This has the following form:

for ⟨*loop variable*⟩ **in reverse** ⟨*value range*⟩ **loop**
  ⟨*loop body statements*⟩
end loop;

The stepping value of the counting loop is +1, for **reverse** loops it is −1.

3.10.6 Repeating structures in C and Java

In C, as much as in Pascal there exists a pretest loop. Its form is the following:

while (⟨*arithmetical-logical expression*⟩) ⟨*statement*⟩;

The statement part given in the loop will be executed, as long as the expression in parentheses after the **while** is true (has a not zero value).

The posttest loop in C looks like the following:

do {
  ⟨*statements*⟩
} **while** (⟨*condition*⟩);

The statements in the loop will be executed: at least once, then the loop body will be executed as long as the expression in parentheses after the **while** is true (has a nonzero value). The C language also supports the counting loop:

for (⟨*initializaton*⟩; ⟨*condition*⟩; ⟨*loop variable update*⟩)
  ⟨*statement*⟩;

This is equivalent to the following **while** loop:[13]

```
⟨initialization⟩;
while (⟨condition⟩) {
  ⟨statement⟩;
  ⟨loop variable update⟩;
}
```

In the **for** statement the 3 elements separated by semicolons can each be an expression. <Initialization> is usually an assignment, <condition> must hold for staying in the loops, <loop variable update> can be an arbitrary expression: usually here there is an expression to increase or decrease the loop variable, but any kind of expression is allowed.

Regarding the **for** type loop, Java allows the loop variable to be declared in the initialization part, having its scope and lifetime limited to the loop. So the scope of the loop variable is well specified, its value after leaving the loop is not defined (since that variable is no longer visible). Consider the following example:

```
for (int i = 0; i < 100; i++) System.out.println(i);
```

Conversely, if the loop variable was defined outside the loop, its value can be accessed after exiting the loop. This is illustrated in the following example:

```
int i;
for (i = 0; i < 100; i++) System.out.println(i);
/* value of i after the loop is 100 */
```

3.10.7 Novelties of the C# language

Control structures of the C# language are comparable in many ways with those of Java; C# has also taken over many features form the C language.

The C# language also supports another loop type besides the loops described earlier, the so called *iterator loop* (also called traversing loop). This allows organizing a counting loop with a loop variable, where it iterates over the values of a set which implements the *IEnumerable* interface (for more on interfaces, see Section 10.7.6., for more on the *IEnumerable* interface, see the official description of the C# language and its standard library). The **foreach** loop statement serves this purpose. An example for its usage is the following:

```
int[] Integers = { 1, 2, 3, 5 };
foreach (int x in Integers) {
    Console.WriteLine(x);
}
```

[13] The only exception is when the program executes a **continue** statement in the *statement* part before evaluating the expression. If so, the **while** loop simply jumps to the condition, the **for** loop updates the loop variable first, and then checks the condition.

In the above statement the value of the *x* variable iterates over all the elements of the array named *Integers* and outputs their value (thus, the program writes the 1, 2, 3, 5 numbers in separate lines). The value of the *x* variable is declared in the beginning of the loop, and cannot be modified within the loop. If the elements of the array should be modified, the regular **for** loop must be used. This type of loop is called *iterator loop* or traversing loop.

Arrays implement the *IEnumerable* interface, they have a *GetEnumerator* method returning an instance implementing the *IEnumerator* interface. Through this interface the elements of the set to be traversed can be accessed. The interface specifies the implementation of three methods – namely the *MoveNext*, *Current*, *Reset* methods. These methods enumerate the elements in the set. Arrays automatically implement this interface, but similar classes can also be made for implementation. The following example enumerates all the even numbers.

```
public class EvenGenerator : IEnumerable {
    int max;                    // upper limit
    public EvenGenerator(int max) { this.max = max; }
    public IEnumerator GetEnumerator() {
        return new EvenPointer(max);
    }
    public class EvenPointer : IEnumerator {
        int max;    int current = -2;
        internal EvenPointer(int max) { this.max = max; }
        public Object Current { get { return current; } }
        public void Reset() { current = -2 };
        public bool MoveNext() {
            if (current > max) return false; // No more below max.
            current = current + 2;
            return true;          // Next even number ready.
        }
    }
}
```

Even numbers in a given range (in our case, in the range from 0 to 100) can be traversed by the following **foreach** statement:

```
foreach (int value in new EvenGenerator(100)) {
   Console.WriteLine(value);
}
```

By generating an equivalent **while** form, the compiler automatically creates the required method calls (first the generator gets initialized, then as long the *MoveNext* method returns a logical true value, the variable named value gets its current value from the return value of the *Current* method, which can be used anywhere within the loop body).

In this way a loop can be easily written which iterates, for example, over all the prime numbers.

3.10.8 Iterators

The above *EvenGenerator* example in C# demonstrates the essence of iterator loops. In this section we elaborate on the concept of iterators in more detail.

Iterators and the CLU

The notion of the iterator was introduced already in the CLU language. Its essence was the same as what we have shown in our C# example: the computed value set of a function had to be traversed in a program. But the implementation slightly differs now from that of in CLU. The iterator is implemented as a standalone function. This function, like the solution in C#, delivers the next value – with the ksyield statement designed specifically for this purpose (its role can be compared to `return`). A complete example will be omitted because of the very unique notation system of the CLU.

The following example code snippet shows the structure of an iterator in CLU.

```
all_primes = iter () yields ( int ) % iterator function declaration
  p : int           % p is an integer variable
  yield(2)          % returning first prime
  while true do
    % computing next prime number
    % (stored in p)
    yield (p)       % returning next prime
  end
end all_primes
```

Following this, in the next statement, the loop variable `i` will iterate over all the prime numbers:

```
for i : int in all_primes() do ....
```

The iterator as a control structure abstraction

As shown above, some programming languages support some form of the iterator loop statement. These languages allow some kind of generation and processing of data elements by the iterator loop. For this we need the generator of the first, and for every next element. Using these, the iterator loop statement can traverse and process the data elements of an arbitrary data structure.

This is about general algorithm-classes: iterators build bridges between algorithms and data structures containing data elements (so called containers[14]). The iterator algorithm must fulfill a well defined task: it must support traversing the elements of data structures in some order, without requiring detailed knowledge about its structure. All it needs to know about the traversed data structure is how to get the data elements from it. The developers of the data structures must only know how – through which interface – the iterators will ask for the next needed data element.

The notion algorithm-class may remind us of classes in object-oriented languages which are central tools for abstract data types and data abstraction. The designers of the C++, C#, Java and other programming languages have also studied this technique for algorithm abstraction, and have worked out an easily applicable method for generalizing algorithms traversing the content of abstract data structures, without introducing new language elements.[15] These are called traversing (or iterator) algorithms. To this end, the languages mentioned above offer the so called iterator classes specified as part of the language, but implemented in a standard library.

The standard library of the C++ language, the STL has the richest and best developed iterator program structures. C# and Java also contain such elements. In Java first the *java.util.Enumeration* interface was introduced to support iterators, then with the collection framework it was enriched with a variety of possibilities, which were previously only accessible for C++ programmers. Based on the collection framework of Java, the developers of the C# language worked out a framework with similar expression power.

Using well the possibility of operator overloading in C++, the designers of the standard library STL for C++ managed to give the programmers support for processing complex abstract data types in the same natural way, as in the case of basic data types. For example, the `++` operator with integer numbers means incrementing by one: the value of an integer typed variable will be changed to the next integer number. In the case of a list, the `++` operator could mean the selection of the next list element within a list traversing algorithm. In languages where there is no operator overloading, these steps can be done by calling the appropriate methods. This results sometimes in a less readable code than the solution with operator overloading.

Consider the following iterator, which computes the sum of the elements in a data structure. The iterator is defined as a C++ template; the exact meaning of

[14] The container terminology is mainly characteristic of the C++ language, as for Java environments and in case of the C#, the notion collection is widespread to name the same feature.

[15] This approach is based only on the existing statements of the language.

totaling and the implementation of the operations executed in each step depend on the data type, which is specified as the parameter of the template.

```
template<class C> typename
C::value_type total(const C& c) { // c is a collection of the elements.
    typename C::value_type sum=0; // Initialize sum with 0.
    typename C::const_iterator iter=c.begin(); // Start from beginning.
    while (iter != c.end()) {     // until the end is reached.
        sum = sum + *iter;        // Add actual element to the sum.
        ++iter;
    }
    return sum;
}
```

The `*` operator is used to dereference the iterator (by accessing its actual value), and the `++` operator increments the iterator: this will step to the next element of the collection passed as the parameter to the *total* method. The successive order is determined by the developer of the *C* collection. The *value_type* denotes the type of data to be summarized,[16] which is usually a template parameter. The *const_iterator* type declares an iterator, which does not allow the modification of the content of the traversed collection. It has the methods *begin* and *end* shown in the example, which return the first and the last element according to the order of the underlying traversal. For more details about the specialties of the C++ language, templates and operators, please refer to Chapter 11 and 5.

The STL distinguishes 5 main types of iterators (apart from the simple iterator, which only refers to a single element, without any other next elements). The simplest form of iterators are the input and output iterators: these are forward moving and allow to traverse the collection in one pass. Through input iterators the referenced element of the collection (its value) cannot be changed; output iterators support modifications (changing the actual value or expanding the collection), but referencing the actual value is not possible. In practice, this distinction is based on which operators have to be defined for each iterator type. For output iterators the operators `*p=` (assignment) and `++` must be implemented, whereas for input operators `=*p` (referencing the actual value), `->`, `++`, `==` and `!=` must be implemented.

A special form of the input and output iterators are the forward iterators. These allow both read and write access and traverse forward the elements of the collection. Bidirectional iterators support also backward moving within the elements of the traversed collection by defining the `--` operator.

[16] This notion comes from the STL.

The most special form of the iterators is the random access iterator, which allows relative addressing,[17] and the measuring of distance between elements.

The implementation of such iterators will not be discussed here. The specification of a standard library usually does not deal with the implementation details of the given library element. Specification only sets the interface to call from the outside. It is the task of the programmer to choose the best feature to implement an iterator. For example an iterator over all possible permutations of the elements of a set can be implemented as a sequential subprogram, but in environments offering features for parallel programming it would be a legitimate option to implement this as a program thread running in parallel (see Chapter 13).

3.10.9 Loop statement in Smalltalk

In Smalltalk the loop statement can be similarly handled as has already been shown with regard to branching (there is a pretest **while** type loop statement, and an array iterator loop). The only salient difference is that the loop condition must be evaluated by a block, which will be called after each iteration. For sending this block – which returns a logical value – the `whileTrue:` message is used, whereas for specifying the loop body block as a parameter, the traditional **while** type structure is used.

Consider the following example:

```
|declaredVariable|
...
declaredVariable := 1.
[declaredVariable < 10] whileTrue: [
   Transcript showCR: declaredVariable.
   declaredVariable := declaredVariable + 1.
]
```

The condition is between square, not round brackets. This is because a block must be specified there, not a logical value. The Smalltalk virtual machine evaluates the given block before every loop iteration, then it sends the `whileTrue:` message to the resulting logical value with the embedded block as parameter (which now outputs the value of the variable named `declaredVariable`, and increases it by one).

This program writes the numbers from 1 to 9 to the screen.

The other array iterator is based on the idea that every array can receive a `do:` message with a block as its parameter. That block must accept one parameter

[17] The pair of this iterator type on elemental data types may be found in pointer arithmetic. It is not a coincidence that for these iterators all the operations of pointer arithmetic must be also implemented.

(the actual array element will be passed there). With this message the block passed to the `do:` will be executed for every element of the given array.

```
|anArray|
...
anArray := #('one' '2' 3 4 5 '6' 7.0).
anArray do: [ :elem |
   Transcript ShowCR:elem
].
```

The above example iterates over the elements of the `anArray` array and writes them to the screen. As shown above, the array elements can be of any type. In one array string and numerical elements, or any other values may coexist. The reason for this is that every possible value is an object, and the array stores references to these objects, which are for every data type the same (but the types of the referenced values can differ).

3.11 Self-invoking code (recursion)

For the sake of completeness about control structures the self-invoking code, or recursion must be mentioned as a feature of control management (as this falls under the topic of procedure and function calling, for more details see Chapter 7). The possibility and the functioning of the self-invoking code is based on the fact that the name of the procedure (or function) is visible within its body, so it can invoke itself. When a procedure of function invokes itself directly or indirectly (through another procedure), it is called self-invoking code or recursion.

The Pascal language supports self-invoking code. In COBOL and in the so called second generation languages like FORTRAN, self-invoking is not possible.

At designing self-invoking algorithms, it is important to rule out cases when a procedure of function calls itself infinitely. This can be achieved by observing two basic rules at designing self-invoking algorithms:

- Every self-invoking algorithm must contain a non-recursive alternative. While it may be empty, usually it contains the solution for the *base case* of the problem.
- In the case of recursive alternatives, care must be taken that the calling chain *eventually reaches* the non-recursive alternative (the base case).

Consider the classic example, the recursive definition of the factorial function:

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

The simple case here is to compute the factorial of 0, which is 1 by definition. The other alternative describes the factorial computing for positive numbers.

The task (computing the factorial of the number n) is reduced to the factorial computation for the one less number $(n-1)$, which must be multiplied by n. Based on the above definition of the factorial it can be seen that the algorithm will terminate for every positive n: as the task is always to compute the factorial for lesser numbers, 0 will be reached definitely, for which the result is defined non-recursively: value 1.

The two characteristic cases of applying self-invoking algorithms are the following:

- The solution of the problem can be achieved by using the same algorithm on partial problems and using these partial results to compute the result;
- A data structure containing a self reference must be processed. For example, take the tree data structure. In the case of a tree data structure, a simple case is to process the leaves, and the processing of the whole tree can be divided into smaller task, e.g. to process the root and the subtrees.

The lack of self-invoking algorithms in the early languages caused no problems as every recursive algorithm can be transformed to non-recursive (iterative) one by using a loop and with the help of a stack data structure. The execution stack of the recursive program is simulated with the auxiliary stack (pushing into the parameters of the self invocations, storing the subtasks to be performed).

3.12 Summary

In this chapter we have discussed the basic formal and informal tools of algorithm descriptions, from low level (assembly) languages – with an example of the control structures – to high level programming languages. By focusing on commonly used programming languages, we have examined some specific control structures. That is, we have discussed COBOL, Pascal and the C language in great detail, and the control structures of Modula-3, Ada, Java, and the C# languages were also dealt with.

The progress has been evident – to manage control, the features are improving fast.

The control structures have been mostly investigated usually on the basis of some shared features, which have been described in detail.

Obviously, apart from the programming languages discussed here, there are many more languages. These have been developed for specific purposes and are used by a specific (usually small) target group only. Their control structures are often the extensions of the ones presented here.

We have given two application areas as an example: database management and realtime applications. Out of the database management languages, it is, for example, the SuperNOVA which fits the transaction based aspects of databases in terms of its control structures: execution of subprograms can be successful or unsuccessful. In the case of a successful execution, it is likely that the transaction

containing modifications made in the database are final (that is, the data have been modified in the database). By contrast, in the case of an unsuccessful execution, designers assumed that the transaction also gets rolled back. Thus, if an error has occurred in the subprogram, the non `STATIC` global variables get their initial values restored back, and with signaling the unsuccessful condition, the execution of the subprogram terminates immediately. This is the main characteristic nature of other so called fourth generation languages (4GL).

The other area is that of realtime applications: in a realtime environment, the execution of the program is not necessarily correct only because it delivers correct results. There are also time limits, which must be considered by the program. *The execution of a realtime program is correct if it delivers correct results before a set time limit.* In these languages there are much simpler (a narrower set) control structures than shown in this chapter, based on which the compiler is able to produce correct code by proper scheduling of the statements; else if the compiler detects that there is no such scheduling which could deliver correct results in time, the compilation can fail.

3.13 Exercises

Exercise 3.1. Implement a program to compute the n^{th} Fibonacci number using the LMC assembly language introduced in this chapter.

Exercise 3.2. Examine the redundancy of the control structures discussed in this chapter. Which control structures can be omitted without decreasing the expressive power of the language?

Exercise 3.3. What is your opinion about the application of self-modifying code?

Exercise 3.4. Write a self-invoking function to define all possible permutations of an integer array.

Exercise 3.5. Write an iterative (non-recursive) function to define all possible permutations of an integer array.

Exercise 3.6. What is the output of the following C# code snippet?

```
int i = 2;
switch (i) {
  case 0:
    goto case 2;
  case 1:
    goto case 2;
  case 2:
    System.Console.WriteLine("i < 3");
    break;
  case 3:
    System.Console.WriteLine("i = 3");
    break;
  default:
    System.Console.WriteLine("i > 3");
    break;
}
```

Exercise 3.7. Imagine the C-based control program of an automated missile defense and retaliatory strike system. Why would this code earn its naming, "the last C bug of the world"?

```
#include <iostream>
using namespace std;

void launchMissiles() {
    cout << "This is the end of the world!" << "\n";
}
int getRadarStatus() {
    int i;
    cout<< "Reading the status of the radar: ";
    cin >> i;
    return i;
}
int main() {
    int status;
    while (true) {
        status = getRadarStatus ();
        if (status = 1)// enemy attacks!
            launchMissiles();
        cout << "The status value is: " << status << "\n";
    }
    return 0;
}
```

3.14 Useful tips

Tip 3.1. The sequence of Fibonacci numbers is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0$ and $F_1 = 1$ by definition. This definition is recursive, but can be used in an iterative way by practically enumerating all the Fibonacci numbers until n, where only the two last result numbers are required to compute the next one.

Tip 3.2. Think of the multiway branching, how it could be substituted by multiple normal branchings. Normal branching may also be implemented with loops, leaving the loop body after only one execution. But loops may also be considered as branching and jumping back to the same code. Examine how even the three loop types (pre- and posttest, counting) are interchangeable.

Tip 3.3. As we have mentioned, the `ALTER` statement in COBOL supports the making of self-modifying programs. However, this feature is not widespread, nor fully supported. The main reason for this is that to implement self-modifying code, the so called code segment storing the executable form of the program must be modified. Its memory location is managed by most operating systems as read only, partly because of security reasons,[18] and partly because of code segment sharing: if multiple instances of a program are active, at runtime they can use the code segment shared together, as only data segments storing their data must be allocated separately and managed for every program.

Tip 3.4. The permutation of an array is an arrangement of the content of the array into a particular order. The number of permutations of n array elements is $n!$. The solution can be based on the well-known algorithm of R. Sedgewick see [Sed77]: each array element will be exchanged to the end and the others will be recursively permuted.

Tip 3.5. See e.g. the book of Donald E. Knuth: The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations [Knu05].

Tip 3.6. Consider the actual selector value of the switch statement to find the case which is executed.

Tip 3.7. Examine the fatal call of launchMissiles()! Should not it be called only if the radar returns an appropriate status?

[18] Security reasons here imply the protection of the code segment, which is not to be overwritten, not even accidentally, because this is often caused by memory management errors. With this solution, modern operating systems support debugging.

3.15 Solutions

Solution 3.1.

```
            INP          ; 00 901 input n
            BRZ end      ; 01 703 while n > 0
            BRP loop     ; 02 811
end         LDA act      ; 03 509 result is act
            OUT          ; 04 902
            HLT          ; 05 000
one         DAT 1        ; 06 001 decrement
n           DAT          ; 07     loop counter
prev        DAT 0        ; 08 000 previous value
act         DAT 1        ; 09 001 actual value
next        DAT          ; 10     next value
loop        STO n        ; 11 307 store loop counter
            LDA prev     ; 12 508 compute next value = prev + act
            ADD act      ; 13 109
            STO next     ; 14 310
            LDA act      ; 15 509 shift act to prev
            STO prev     ; 16 308
            LDA next     ; 17 510 shift next to act
            STO act      ; 18 309
            LDA n        ; 19 507 decrement loop counter
            SUB one      ; 20 206
            BRZ end      ; 21 703
            BRA loop     ; 22 611
```

Solution 3.2. The multiway branching structure can be substituted by multiple branchings checking always for the next branch. The loop condition may be used to implement branching, if the loop body is left after only one execution. If the starting of a loop body can be marked and jumped to, then a conditional jump back to the loop start, or jumping out of the loop body can replace the loop statement. Counting loops are easily implemented with normal loops and assignments using the loop variable, changing its value properly and checking the desired boundaries. A pretest loop can be substituted by a conditional execution of a posttest loop

`while` (`condition`) `do` something

is equivalent with

`if` (`condition`) `do` (`do` something `until` !`condition`)

and vice versa by executing the loop body additionally before a pretest loop:

`do` something `until condition`

is equivalent with

`do` something; `while` (!`condition`) `do` something

Solution 3.3. Computers of von Neumann architecture store data and code in the memory, allowing the access and handling of instruction codes as data. This is the basis for self-modifying code in assembly level languages, as running code can access and modify itself within the memory. This, of course, must be done extremely carefully, since the slightest error may render the code not (deterministic) runnable. That is why most operating systems try to prevent any write access and modifications to memory areas storing program code. Nevertheless self-modifying code can be used for various purposes, such as optimization of the code (like so called HotSpot optimizers), patching of subroutine (pointer) address

calling, usually as performed at load/initialization time of dynamic libraries, or else on each invocation, patching the subroutine's internal references to its parameters so as to use their actual addresses. It can also be used for hiding (obfuscating) of code to prevent reverse engineering (by use of a disassembler or debugger) or to evade detection by virus/spyware scanning software and the like, and for compressing code to be decompressed and executed at runtime, e.g., when memory or disk space is limited. These techniques are used mainly by low/machine level languages, but in high level languages interpreted code may be also supported "on the fly" mimicking some degree of self modification. For example debugging and profiling on the Java virtual machine platform is also implemented by code injection, that is by modifying the compiled bytecode of every high level statement to include additional instructions during execution.

Solution 3.4. The solution is based on the well-known algorithm of Robert Sedgewick. It is written in Ada. It stores the result in a text file.

```
with Text_IO; use Text_IO;
procedure Perm_Probe_Recursive is
package Int_IO is new Integer_IO(Integer);    use Int_IO;

   N : constant Integer := 3; -- the size of the vector
   type Vect is array (1 .. N) of Integer;
   F : File_type;

   procedure Swap(V:in out Vect; I,J:Integer) is
   --swaps the V(I) and V(J) in V
      Temp: Integer;
   begin
      Temp:=V(I);
      V(I):=V(J);
      V(J):= Temp;
   end;

   procedure Put_Vect(V:Vect) is
   begin
      for I IN 1..V'Length loop
         Put (F, V(I)); --puts into the file
         Put (V(I)); -- puts on the screen too
      end loop;
      New_Line;
      New_Line (f);
   end;

 procedure Perm(V:in out Vect; N:Integer) is
-- gives all permutations of the  values in V recursively
   begin
      IF N=1 THEN
         Put_Vect(V); --we are ready
         RETURN;
      end IF;
      for I IN 1..N-1 loop
         Swap(V, I, N);
         Perm(V, N-1);
         Swap(V, I, N); --back, for the next trial
      end loop;
      Perm(V, N-1);
   end Perm;

   Myvect: Vect:=(4,5,6); --the actual vector
```

```
begin
   Create(F, Name=>"perms2.txt");
   Perm(Myvect, Vect'Length);
   Close (f);
end;
```

The permutations from the file perms2.txt:

```
5          6          4
6          5          4
6          4          5
4          6          5
5          4          6
4          5          6
```

Solution 3.5. A possible solution is written in Ada, it gives all permutations of values in an integer array and prints the result to a text file. The parameters of the procedure are: the input vector and the name of the result file. The file-handler procedures are local in the permutation procedure. One can change the size of the vector, and the elements in it, the actual values are only demonstrations of the call.

```
with Text_IO; use Text_IO;
procedure Perm_Probe_Iterative is

   package Int_IO is new Integer_IO(Integer);    use Int_IO;

   N : constant Integer := 3; -- the size of the vector
   type Vect is array (1 .. N) of Integer;

   procedure Permutations ( V  : Vect;  File_Name : String) is
      -- all permutations of the elements in V vector will be
     -- in the text file File_Name
    procedure File_Get ( F : File_Type;  Tmp  :  out Vect ) is
    -- reads the values of the next line in the file F to Tmp vector.
      I : Integer;
     begin
        I:=1;
      while not End_Of_Line(F) loop
         Get(F,Tmp(I));
         I:=I+1;
      end loop;
      Skip_Line(F);
   end File_Get;

   procedure File_Put ( F : File_Type; Tmp :  Vect;  Size : Integer) is
  --puts the values of the Tmp vector
  --from the 1..Size interval to the file F
   begin
      for I IN 1..Size loop
         Put(F,Tmp(I));
      end loop;
      New_Line(F);
   end File_Put;

   procedure File_Copy (F_In, F_Out:File_Type) is
  --makes a copy of the file F_In to file F_out
      I : Integer;
   begin
      while not End_Of_File(F_In) loop
         while not End_Of_Line (F_In) loop
            Get(F_In, I);
```

```
         put(f_out,I);
      end loop;
      Skip_Line(F_In);
      new_line(f_out);
   end loop;
 end File_copy;

   Temp1, Temp2 : Vect;
   F1,  F2 : File_Type;
   Act_Size,  Act_Pos  : Integer;
   Temp_Inp : Boolean;

begin
   Act_Size:=1;
   Act_Pos:=1;
   Create (F1, Name=>File_Name);
   Create(F2, Name=>"temporary.txt");
   Put(F2,V(1));
      --first element of the vector is written into temporary file
   Temp_Inp:=True;

   for I in 2..N loop -- iterates on the indexes of the vector
      if Temp_Inp then
      -- this is the case when F2 will be the input file
      -- and F1 the output
         Reset(F2, In_File);
         Delete(F1);
         Create (F1, Name=>File_Name);
         while not End_Of_File(F2) loop
            File_Get(F2, Temp1); --next line from the file
            while Act_Pos <= (Act_Size+1) loop
               if Act_Pos = 1 then
                  Temp2(1):=V(I);
                  Temp2(2..Act_Size+1):=Temp1(1..Act_Size);
                  Act_Pos:=Act_Pos+1;
                  File_Put(F1, Temp2, Act_Size+1);
               else
                  Temp2(1..Act_Pos-1):=Temp1(1..Act_Pos-1);
                  Temp2(Act_Pos):=V(I);
               Temp2(Act_Pos+1..Act_Size+1):=Temp1(Act_Pos..Act_Size);
                  Act_Pos:=Act_Pos+1;
                  File_Put(F1, Temp2, Act_Size+1);
               end if;
            end loop;
            Act_Pos:=1;
       end loop;
       Act_Size:=Act_Size+1;
   else
   --this is the case when F2 will be the output file and
   --F1 the input
       Reset(F1, In_File);
       Delete(F2);
      Create(F2, Name=>"temporary.txt");
      while not End_Of_File(F1) loop
         File_Get(F1, Temp1);
         while Act_Pos<= (Act_Size+1) loop
            if Act_Pos=1 then
                  Temp2(1):=V(I);
                  Temp2(2..Act_Size+1):=Temp1(1..Act_Size);
                  Act_Pos:=Act_Pos+1;
                  File_Put(F2, Temp2, Act_Size+1);
             else
                  Temp2(1..Act_Pos-1):=Temp1(1..Act_Pos-1);
                  Temp2(Act_Pos):=V(I);
               Temp2(Act_Pos+1..Act_Size+1):=Temp1(Act_Pos..Act_Size);
```

```
                    Act_Pos:=Act_Pos+1;
                    File_Put(F2, Temp2, Act_Size+1);
                end if;
             end loop;
             Act_Pos:=1;
             end loop;
             Act_Size:=Act_Size+1;
          end if;
          Temp_Inp:= not Temp_Inp; --swaps the input and output files
       end loop;

        if Temp_Inp then --the result is in the temporary
           Reset(F2, In_File); Reset(F1, Out_File);
           File_Copy(F2,F1);
         end if;
       Close(F1);
      Close(F2);
    end Permutations;

     V : Vect := ( 5, 6, 7); --actual trial

  begin
     Permutations(V, "perms.txt");
  end;
```

The permutations from the file perms.txt:

```
7         6         5
6         7         5
6         5         7
7         5         6
5         7         6
5         6         7
```

Solution 3.6. The result is: `i < 3`

Solution 3.7. This is a typical C bug related to the assignment expression support of the language. As the conditional statement receives an assignment (=) instead of an equality comparison (==), and the value of this assignment (*=1*) is treated as a boolean true expression, the conditional statement will always be executed regardless of the value of the status variable. A good programming style is to use for comparisons with constant values a reverse order, always putting the constant to the left side, so if the comparison operator turns into an assignment, the compiler will give an error, since constants are no L-values and can not be assigned to. Additionally, it is always a good idea to turn up the warning level of compilers, because this common error can usually intercepted in this way. So, having the comparison fixed, the following version of the main function will probably guard the world better:

```
int main() {
    int status;
    while (true) {
        status = getRadarStatus ();
        if (1 == status);// enemy attacks!
            launchMissiles();
        cout << "The status value is: " << status << "\n";
    }
    return 0;
}
```

Or NOT?!?

4 Scope and lifespan

Whichever programming language and environment we use, whichever paradigms they adhere to, whichever possibilities and features they offer, the scope and lifespan of the variables, the functions and the types are important notions.
The scope of an identifier is the part of the source code from where a language element (an object, a function, a type, etc.) is accessible with the given name.
The lifespan of an object is the part of the program's running time between the creation and the disposal of the given object, that is, when the object is live, is usable.
Scope and lifespan are related concepts, but their meanings do not overlap.

The principle of information hiding (so that the parts optimally connect) is essential in bigger projects, which involves the close cooperation of a number of programmers. The writer of a part should know about the part written by the other programmers only to the necessary extent: interface. There should be no unnecessary or accidental dependence between the parts as it encumbers human work, causes complications and possibly errors. When working on a program in collaboration with others, it would be annoying to find out that the name i (our favorite one for a loop index) cannot be used anymore, because another programmer was faster and already used it inside another module for some other purpose. Our own program can become unfamiliar after some while, so we can become "another programmer" in this sense.

The principle of information hiding is supported to different extent with different degrees of sophistication by the different programming languages.

A program uses different elements, e.g. types to describe common data structure and behavior, functions to encapsulate and reuse execution, and "objects" – things stored in the memory. These elements may have zero or more names: identifiers which can be used to refer to them. Usually such elements have names which are directly used to access them, but it is also possible that an object has no name at all and is accessible only indirectly (that is, with the help of some language construction, like pointers), or it can be accessible in both ways. Examples for only indirectly accessible objects are dynamically (that is, in runtime) allocated objects and elements of arrays. Dynamically allocated objects are usually accessed through pointers or references, an element of an array is usually accessed with an expression combined from the array name and the actual index.

Identifiers must follow specific syntactical rules, the exact details depend on the language, environment, and on compilation flags. In most programming languages, however, it is safe to use alphanumeric characters, usually started by a letter and continued by letters or numbers. In many programming languages identifiers are case sensitive, and the number of significant characters in an

identifier may be limited. Thus, it is always always useful to check for the exact rules of identifiers in the specific programming language.

Identifiers are important resources for the programmer. A well-chosen identifier helps to understand the role of the programming element, like an interface function or a type name. A wrong identifier can be misleading and may deceive the reader of the code. Therefore for those identifiers which are accessible in many places of the program it is worth choosing longer, and telling names. Only for identifiers used locally do we tend to choose short names. A typical loop variable for example is called i – its usage context explains its role. A possible rule of thumb can be the broader the scope, the longer the name.

Identifiers can be reused, i.e. the same identifier may refer to different elements of the program in different parts of the source code. The part of the source where from an identifier can refer to a particular element of the program is called the *scope* of the identifier. The scope – or visibility – rules of an identifier are language specific; however, modern programming languages share many common patterns when defining scope rules.

During program execution we use different memory locations to store our objects. These memory locations are not necessarily needed for the whole execution time and modern programming languages can reuse unused memory parts. The allocation of the memory should precede the use of the object and we we must not refer to the memory after we have abandoned it. There is a specific time interval of the program execution when we rightfully refer to that memory area as the storage place of a certain object. This interval is called the *life* of the object. Referring to a memory area out of the object's life time may lead to invalid results or even run-time errors.

4.1 The types of memory storage

Programming languages typically define some abstract model to describe the proper behavior of the language. The description includes the memory storage model, i.e. how objects are mapped to the physical implementation, and how long they are accessible for use. The memory storage model is an important aspect of the scope and life of the objects.

4.1.1 The static memory

Static memory is sized and allocated at compile time, when also an initial value (defined by the programmer or default) is assigned. This memory will be available during the whole execution of the program.

The main advantage of the static memory is its simplicity. Static memory usually requires no runtime maintenance (object construction and destruction in object-oriented languages may be exceptions). However, this simplicity is also a drawback. Static memory is allocated for the whole execution time, even if its

contents are used only in a fraction of the execution – this is rather uneconomical. Besides this, they have certain drawbacks in a multithreaded environment as well – as they are shared between threads their access must be protected.

Static memory is primarily used for global variables – thus, sharing information between modules or subprograms during most of the execution time.

The executable code of the program may also be considered a static value, even if it is not manipulated in the same way as the data. The environment (the hardware and the operating system) might support the separation of the read-only and the read-write parts. It is of advantage to store the code of the program in a read-only part, so that an already running copy of a program may share it with a newly started instance, albeit at the price of preventing the self-modification of the code.

4.1.2 The automatic memory

We often use objects with a restricted lifetime: a loop variable is usually required only during the execution of the loop; math computations apply variables to store the temporary results before further use, etc. It would be a serious waste of resources to use static memory for such purposes. Instead, block-oriented languages use *automatic* memory to reserve storage when entering a block and keep that storage valid until leaving the block. The name automatic – or auto in short – comes from the fact that no programmer action is required to mark the beginning and the end of the life of such objects.

Automatic storage is usually implemented by a *stack*, a LIFO (last-in, first-out) data structure. When entering a block, the stack pointer identifying the top of the stack is incremented by the size of the required automatic variables - thus "allocating" memory. When leaving the block, the stack pointer is set back to the initial value to "free" the automatic storage. This position is also stored in the stack. As usually no other action happens simultaneously automatic variables are uninitialized – they may contain the "garbage" of the earlier content of the memory. In some object-oriented languages, however, constructor and destructor calls may be associated with these actions.

In many cases the function call mechanism are also implemented by using this stack. If recursion (that is, the ability of a function to directly or indirectly call itself) is not supported by a language and an environment, maximum one copy of a function may be active at any time. This may it is possible to store the data necessary for the calling of the function in the static, (that is, at compile time) preallocated memory. If, however, recursion is to be supported, as is generally the case, the data necessary at runtime for the calling of the function (parameter values, return address, register values) are stored in the stack. The stack area corresponding to a function call is called an *activation record*. As functions call each other their activation records build up on top of each other.

In multithreaded execution environments, each thread has its own stack, meaning, automatic storage variables are thread locals.

It is especially dangerous to refer to an object with automatic life after its lifetime – i.e. after leaving the block. The memory address will be still valid (therefore, no run-time error will occur), but it may happen that the value of that memory area could be already modified – perhaps an other block is using that activation record and our value has been modified by then.

4.1.3 Dynamic memory

There are situations when neither the static nor the automatic storage suits our purpose. This is the a case is when we need a storage with a shorter lifetime which still has to leave the block of creation, or when we do not know the size of the object in compilation time. In these situations we use *dynamic* memory.

In the case of dynamic memory the programmer is fully responsible for the lifetime control of the object. Dynamic memory handling is either supported directly by language features (e.g. the **new** operator of C++ and Java, and *delete* in C++), or indirectly by (standard) library functions (e.g. C's *malloc* and *free*). The allocation of such memory is based on the explicit request of the programmer: they typically use a **new** expression or initiate a specific function call. Allocation happens in a data structure called *heap* or *free memory*. The allocation process looks up an unused continuous area, administers its use and returns a pointer to it. The deallocation process takes the pointer to an allocated area on the heap and marks it free with some additional actions (like concatenating free neighboring areas).

In some implementations frequent allocations and deallocations may fragment the heap. Moving allocated fragments would invalidate pointers or references to the area, thus it is impossible in many languages. In the *managed heap* of the .NET system, however, allocated fragments can be moved and references are updated. A Java Virtual Machine can be implemented in many ways, and memory handling and garbage collection are eminent subjects to steady research, development and refinements. Oracle's HotSpot VM includes several generational garbage collectors, which rank the objects into different generations and store them in several heap areas and move them around. The gory details actually do not have to do with the core language features and the themes discussed here. Due to them being mainly implementation details, we, as programmers are not affected by them and we are protected from their complexity. However, when we want to fine-tune the JVM to improve performance and to eliminate possible bottlenecks, and pondering about picking a garbage collector and customizing its parameters, we still need to deal with them.

While heap allocation actions are controlled by the programmer, deallocation may happen either upon the programmer's request (by *free* in C, *delete* in C++), or can be automatically controlled by the *garbage collector* (ADA, Java, C#).

Heap allocation and deallocation are expensive operations in execution time and in possible lock conflicts. Therefore, heap allocations should be minimized in performance critical programs.

Although dynamic memory is quite flexible, the handling of dynamical data may lead to some errors still:

- memory is allocated but never eventually freed (memory leakage);
- several attempts are performed to free the same memory area;
- a pointer or reference is held to an already freed and possibly (for other purposes already) reused memory;
- everything is done correctly but the memory still gets exhausted.

Dynamic memory handling is not supported by every language and every environment. For example COBOL was not designed with it in mind, and even if some versions and implementations support it, it is a foreign feature to its logic and its culture.

4.1.4 A simple example

Let us consider a simple and admittedly contrived example to compute the third member of the Fibonacci series in a highly inefficient way, which however illustrates recursion and variables in different storage areas. The Fibonacci series was invented by Leonardo Fibonacci in 1202. For purposes of demonstration let us assume the rabbits are immortal (at least in the interval being discussed), and a pair of rabbits produces (the female gives birth to) a pair every month after the age of one month. From these assumptions it follows that the number of rabbit pairs in a given month is the sum of the number of the rabbits in the previous month plus the number of the newly born pairs, with the latter being equal to the ones living two months ago. If $F(n)$ denotes the number of the rabbit pairs after n months, then $F(n) = F(n-1)+F(n-2)$. An interesting series (occurring at several places in mathematics including the analysis of algorithms [Knu87]) is defined this way.

The series is defined in itself: a value can be computed from the previous ones. A widely used representation of this idea is computing the *fib* function in a recursive way. The following example is written in C++, but it is in a procedural, – in fact – C style. We chose the language C++ rather than C, because of the relatively easy programming of the output due to *std::cout*.

```
#include <iostream>
int ct;
int fib (int what) {
    int ret = 1;
    ct++;
    std::cout<<ct<<" fib ++ "<<what<<" "<<(void *)&ret<<std::endl;
    if (what > 1) ret = fib (what - 1) + fib (what - 2);
    std::cout<<ct<<" fib - "<<what<<"="<<ret<<std::endl;
    return ret;
}
```

```
int main () {
    int x = fib (3);
    std::cout«"the result is "«x«" in "«ct«" step(s)."«std::endl;
    return 0;
}
```

In a given environment the following output is generated:

```
1 fib ++ 3 0xbffff708
2 fib ++ 2 0xbffff6d8
3 fib ++ 1 0xbffff6a8
3 fib -- 1=1
4 fib ++ 0 0xbffff6a8
4 fib -- 0=1
4 fib -- 2=2
5 fib ++ 1 0xbffff6d8
5 fib -- 1=1
5 fib -- 3=3
the result is 3 in 5 step(s).
```

In order to compute *fib(3)*, we need the values of *fib(2)* and *fib(1)*, so there are several active calls to *fib* with different parameters, and local *ret* variables. The address of *ret* is output so as to demonstrate it. The calls of the function *fib* are counted in the global variable *ct*.

4.2 Scope

The scope of an identifier is that part of the source code, where the denoted element (variable, process, type etc.) can be accessed with the given name.

Scopes include the following areas:

- the entire program;
- a compilation unit (see below);
- a subprogram;
- a block of code;
- a type (a class) or a namespace.

Different scopes may of course, overlap: a given point of the program can (and often does) belong to more scopes. Scopes usually contain each other: a point in a function belongs to the scope of every containing block, to that of the function, and to the global one; in object-oriented environments the scope of a type (a class) can be part of those of others due to inheritance.

```
extern int x;                    /* global x is defined somewhere else */
#ifdef __cplusplus
namespace N {
    int x;
    void f();
}
#endif
int foo(int y) {
    if (y == 0) return x;        /* global x. */
    {
        int x=2;
#ifdef __cplusplus
        if (y > 0) return ::x;   /* C++: global x */
        N::x++;                  /* C++: x from namespace N. */
#endif
        if (y < 0) return x;     /* 2 */
        {
            int x = 3;
            return x;            /* 3 */
        }
    }
}
#ifdef __cplusplus
void N::f() {
    x++;                         /* denotes N::x. */
}
#endif
```

The resolution of a name, that is, the search for the element denoted by a name (identifier) is performed among the scopes beginning at the narrowest proceeding towards the broadest one. So a declaration of a name in a closer scope hides the names declared in a further, wider scope.

However, names can be overloaded, that is, it is possible for a name to denote several different things at the same time. Obviously, this only applies when the denoted language elements have features that set them apart, apart from the name itself. In languages which support static (compile-time) type checking, functions may be distinguished by the signature, that is, the number and types of the arguments, so functions with the same name but with different signatures can be handled:

```
void print(int i);
void print(char *s);
void print(char c);
```

ANSI C performs type checking, but does not support the name overloading

in the above sense. During the resolution of an overloaded function name ADA takes the return type into consideration, but C++, C#, and Java do not.

C++ supports the scope operator ⟨*<name space name>*⟩*::*⟨*<object>*⟩ or ⟨*<class name>*⟩*::*⟨*<object>*⟩. We have seen the example *std::out* denoting the *out* object (handling the standard output) of the namespace *std* of the standard library objects and functions. Special version of scope operator with empty left operands *::x* denotes the global *x* variable.

4.2.1 Global scope

Global variables are accessible from every point of the program and reside in static storage. They enable easy communication among different parts (modules, functions) but can cause difficult dependencies among them. If a function modifies global variables, these changes are called the function's side effect. A function can be considered as an encapsulation, a "short hand" for an arbitrarily long and complicated sequence of instructions. Side effects contradict this to some extent.

In C and C++ the global variables are the ones defined outside of every function, namespace or class, and are not marked with the **static** keyword. In FORTRAN the directive *COMMON* can be used to define global memory areas which can be broken into variables in different ways in the different modules.

4.2.2 Compilation unit as a scope

It is important to break programs of at least middle size into independently compilable parts, so that after a change the whole program need not be compiled. Different languages support this with the introduction of different units: partitions, modules, compilation units, library modules, submodules. In large projects it is an explicitly stated design goal to brake programs down into such smaller units[Lak96].

A compilation unit (if the programming language supports it) is the set of the functions and variables put into the same source file and compiled together. (Java does not support this notion: the source of a public class or interface can be only in the source file named after it, and the byte codes of the classes are written into their own **.class** files, even those of the anonymous inner classes.) It is language-dependent how the functions and variables in a compilation unit are accessible from outside.

The variables and functions in the compilation unit deemed only locally accessible, that is, accessible only within the compilation unit, may be put into its scope.

Some languages (Pascal, Ada, C and C++) distinguish syntactically between declarative and implementing parts, others including Java do not. While in the language family of Pascal (Pascal, Ada) and in Java the compiler takes the

necessary information from the object or byte codes of the referred units, in the languages C and C++ special source (header) files serve this purpose.

In C and in its successor C++, the keyword **static** denotes which variables declared outside every function make up the scope of the compilation unit. The storage class will be static, and the scope will be confined to the compilation unit. This heritage from C is a way of modularization. It is a rather unfavorable situation as it ties the question of accessibility (design level) to the question what is put together into the same compilation unit (implementational level).

C++ introduces the notion of the unnamed namespace to define a scope local to a compilation unit. Its usage is equivalent but still it is preferred to the static variables.

The variables in the compilation unit serve as storages for the state of the module (that is, the set of functions in the compilation unit), since the functions share the variables preserving their values across the function calls. The names of the static variables and functions cannot be seen from outside the compilation unit, meaning they are reusable.

4.2.3 Functions and code blocks as scope

Static or automatic variables can have a function (method, subprogram) or in the case of block structured languages a code block as their scope. (The body of a function can be regarded as a code block anyway.) These variables are called local to the function or to the code block. Variables declared in a block are accessible from further blocks inside. From a block of course the locally declared and the global variables are also accessible, and the ones declared in the containing blocks and in the containing function, compilation unit too.

The static variables declared inside a function can be used as storage retaining the value across function calls with the restricted scope of the code block or function. A similar role is played in ALGOL 60 and SIMULA 67 by the local variables marked with the keyword *own*.

If not declared as **static**, the variables declared in a function or a code block become automatic, that is, they are put into the storage automatically allocated at entering and destroyed and possibly reallocated at leaving. The parameters of a function also belong to its scope.

4.2.4 A type as scope

Types and classes are scopes as well. Here the keyword **static** has a special meaning: static members belong to the class and not to the instances. Static data members are of static storage. Instance (non-static) functions access both static and instance functions and data members simply by the name, whereas static functions access only the static ones.

In C++ and in Java both functions and data members can have access modifiers:

- Private data members and functions can only be accessed from within the same class;
- Protected data members and functions can be accessed from within deriving types;
- Public data members and functions can be accessed from everywhere.

Java also applies the default semi-public accessibility without a keyword, meaning accessibility from the same package.

In C++ the private data members and functions of a type can however be accessed from a type declared to be its **friend**. In Java there is no friendship: private members are accessed only from the given class and its internal classes (besides reflection magic).

Inheritance has consequences regarding the scope: the functions of a derived type access both the data members and the functions of the base type as if they had been declared in the derived type, with the exception of the private ones.

4.3 Lifespan

One of the most basic properties of a variable (a memory object) in any programming language is its *lifespan*. The lifespan of an object is the part of the program's running time between the creation and the disposal of the given object, that is, when the memory allocated to the object stores its value. Lifespan is an important notion in traditional (procedural, structured) and in object-oriented languages as well.

4.3.1 Creation and destruction of objects

When an object is created, as a first step, memory is allocated to the object. Then this memory area must be initialized: it must be prepared for use, that is, the object's invariants must be set, the object must be given a consistent internal state. In object-based languages there is a special function for this latter purpose, i.e. the constructor belonging to the given type, whose name is usually identical to that of the type. Due to overloading there can be several constructors with different parameter lists. Initializing functions may, of course, be used in procedural languages: the API may require that a structure be initialized before the first use, but if there is no language-imposed guarantee, the compiler cannot enforce this, so erroneous usage is possible. Object-based languages, however, guarantee that an object is created only in a controlled manner, that is, essentially with the assistance of an appropriate constructor.

Allocating memory is the duty of the runtime environment, searching for and calling the appropriate constructor is up to the compiler, and the correct coding of constructor (and the destructor) is the responsibility of the programmer.

The constructor is neutral with regard to the storage type: it is not aware where the memory area to be initialized was allocated. A constructor is used

to create an instance of a given (exactly known) type. In an object-oriented environment (that is, supporting inheritance) it is possible that only a parent type (implementing some given interface) is interesting, not the exact type of the instance. So some object (of the ”factory” type) might be able to provide an instance of the given interface, which might actually be of an inheriting, derived type. The very method of the factory, however, cannot be storage-type neutral: it must create the object in order to able to return a reference or a pointer to it.

A typical example for this situation is provided by the class *DriverManager* in the package *java.sql* of the Java JDBC API. Its method

public static *Connection getConnection(String url)* **throws** *SQLException*

returns an object implementing the interface *Connection*, if it finds a registered driver class which is willing and able to provide one. If a non-static factory method is used, it can be polymorphic for more complicated design patterns.

Constructors are special ones among the methods due to their relation to memory allocation. Therefore in C++, where there generally exist pointers to the methods, there are none of the constructors.

When an object is destroyed, and if external resources (files, locks, descriptors, handles, dynamic memory) have to be freed, first some house-keeping activities might be necessary. This is followed by the freeing of the memory allocated to the object. The latter is the chore of the runtime environment, but the former is the duty of a special method pertinent to the type, the so called destructor. In C++ its name is that of the type prefixed with a tilde (\ ) to allude to its being complementary to the constructor.

The Java language was designed with garbage collection in mind, therefore there is no destructor mechanism, but a *finalize* mechanism. The Java virtual machine stores the objects – that is, those of class (non-primitive) types – on a *garbage collected heap*, even if the reference variable is static. Memory eligible to *garbage collection* (that is, not reachable through any active path of reference chains) is detected, than finalized (that is, the *finalize* method is run) and freed typically in a low priority thread. The *finalize* method can be used to free resources before the object is lost and its memory is freed. Java does not define, however, when the virtual machine will perform garbage collection, and finalizing. This is in contrast to the C++ language, where the destructor is immediately executed at the end of the object's life span. As a long waited enhancement, in its version 1.7, Java introduced the try-with-resources statement, which guarantees that for every resource declared within the statement, the appropriate *close* method will be executed upon leaving the statement.

4.3.2 Static

Static variables are usually created at program start and persist while the program runs. The order of the creation of the variables inside a module (compilation

unit) follows naturally from their declaration order.

In Java, variables of primitive types and object references can be static, but the objects themselves are created only dynamically on the heap. In Java the variable called *staticObject* of the following class is assigned a value in the static block running at class load time:

```
public class ClassWithStaticObject {
    static int staticObject[] = { 123, 456 };
}
```

In C++ static variables declared within a compilation unit are initialized in the declaration order and destructed in the opposite order at the end of the program. Many C++ programmers (mis-)translate this as "constructors run at the very beginning of the *main* function and destructors run at the end of *main*". This is a serious (and potentially) dangerous oversimplification.

Firstly, local static variables have static storage, but their constructor runs only when the program execution reaches their declaration for the first time. This can happen well after *main* started. It is even possible that the control *never* reaches the declaration – meaning, the object will never be constructed. Such a non-constructed object naturally will never execute its destructor, which reveals that the C++ language should register the construction of local statics under runtime.

Secondly, there are serious problems with the initialization order of global static variables. Although the order of construction of static variables declared within a compilation unit is well-defined, the C++ standard says nothing about the order *between* compilation units. Unfortunately, many C++ programmers ignore this problem, naïvely thinking that static objects are not accessible before the *main* function. However, it may happen (and normally *does* happen) that one of the static object's constructor tries to access an uninitialized static object from an other compilation unit. This can lead to nasty, hard to debug errors.

This static initialization problem can be avoided using the *singleton* pattern: we encapsulate our statics into dynamically created objects.

4.3.3 Automatic

Automatic variables are stored on the stack along with the data needed for the runtime support of procedure calling in an automatically allocated area – hence the name. At leaving the function, this area is automatically freed. In most cases it is dangerous if a variable containing a pointer to some area has a broader scope than that of the area it points to. In the following example, the function returns a pointer to an area which is freed at the moment of the function's returning and just waits to be overwritten:

```
/**** BAD CODE! ****/
char *gettime()
{
    char wtime[24];
    /* fill wtime */
    return wtime;
}
```

C++ keeps track of the automatic variables and calls the destructors (in the opposite order of creation) if control is about to leave a function or a block for any reason like reaching a **return** statement, falling through the closing brace or due to an exception being thrown.

4.3.4 Dynamic

There are languages where the programmer can control the allocation of memory. The lifespans of these dynamical memory areas stretch either until the programmer explicitly frees them (irrespective of the overall structure of the program) or until all references go away and the areas became unreachable. This latter requires support from the runtime environment: "garbage collection". Some languages and environments were designed to include garbage collection (SIMULA 67, Ada, Java, C#), others either do not have it at all or have it as an optional feature, not as part of the specification. Garbage collection not only requires runtime efforts, but it can pose a principal problem. If the language supports pointers which can be manipulated, the value of a pointer to dynamically allocated memory can be hidden in the program. The following code snippet illustrates a situation when the address of a newly allocated piece of memory (originally stored in the variable *pt*) is no longer available in our program, and thus the garbage collection takes this memory as unreachable and as eligible to garbage collection, nevertheless the pointer is clearly reproducible from the values in *pointer0* and *pointer1*:

```
char *pt = new char();
char pointer0[sizeof(pt)];
char pointer1[sizeof(pt)];
*(char **)pointer0 = pt;   // Bitwise copy of pt.
*(char **)pointer1 = pt;   // Bitwise copy of pt.
pointer0[0] = 0xab;
pointer1[1] = 0xcd;
pt = 0;
```

Garbage collection frees the programmer from the burden of freeing the memory by detecting unreachable objects. However, keeping references to an object does prevent it from becoming unreachable, that it, eligible to garbage collection. Unintentional references can be kept in callback handlers, queues etc.

under the hood. This innocently looking code runs to memory exhaustion (to an *OutOfMemoryError*) for example in SUN's JVM v1.4 (but fortunately no longer in v1.5):

```
public class a {
  public static void main( String args[] ) {
    while ( true ) { Thread t = new Thread(); }
  }
}
```

In the infinite loop *Thread* objects are just created (but not started) and the variable holding the reference is reused, and the Threads become seemingly unreachable, so in theory they could be garbage collected and the memory reclaimed. But the particular *Thread* implementation seems to keep track of the unstarted threads obviously retaining references to them, hindering their becoming unreachable and garbage collected. Similar unintentional retention of references can occur in purely user-supplied code as well.

If a reachable reference exists to an object, this makes it reachable and not eligible to garbage collection. The so called *weak references* (as opposed to the common, strong ones) yield a weaker level of reachability not preventing the object's becoming unreachable in the common sense and garbage collected, in which case, of course, the encapsulated strong reference cannot be used any longer. For example, Java (since v1.2) defines three levels of weak reachability listed in diminishing order of strength: *soft*, *weak*, and *phantom*.

4.4 Examples

Usage of a static buffer

Let us assume an operating system storing the creation and last access dates of the files in some machine- but not user-friendly form, the number of the elapsed seconds since some given starting date. For this purpose let *time_t* denote the integral type.[1] Obviously a function will be necessary to convert a given *time_t* value into some human-readable form. Let us call this function *ctime*. When using this function, the question arising who will allocate the memory for the human-readable form. The caller cannot know how much space will be required, so the simple solution is the called function allocating it in a static area. The C standard library *ctime* function does exactly this: it returns a pointer to its static buffer. The function declaration as seen in the header file *time.h*:

[1] An example of it is the UNIX operating system, where the starting time, the so called Epoch, is 1. January 1970. The readers might remember the "Y2K bug" from year 2000. A similar critical situation will arrive on 29. January 2038, when the maximal signed 32 bit integer values will reach their upper limits in UNIX systems. Hopefully, many of our readers will experience the "Epoch 0x7fffffff bug".

```
char * ctime(const time_t *tp);
```

This is the simplest solution but it comes at a price. If we want to print the time before and after a long-running portion of code, the following "solution" will not work. A hint: *time(NULL)* returns a *time_t* value representing the current time.

```
char *before;   char *after;   time_t bt, at;
bt = time(NULL);
before = ctime(&bt);     /* 1 */
long_running_function();
at = time(NULL);
after = ctime(&at);      /* 2 */
printf("%s %s\n",before,after);
```

This "solution" does not work, because the second call to *ctime* reuses the same buffer, so the value stored during the first call is overwritten. Copying this value before the second call would alleviate the problem.

The usage of static buffers is more problematic in *multithreaded* environments as a function might be called by several threads at the same time. Therefore, the usage of static buffers is better avoided. A possible solution is the caller' supplying the necessary buffer and taking care of the allocation and freeing:

```
void ctime1(char *buffer, const time_t *tp);
```

Or we may want to pass along the buffer size:

```
void ctime2(char *buffer, int bufflen, const time_t *tp);
```

The most comfortable solution would be the object-oriented approach, provided it is supported by the given environment.

Resource management through objects

Object-based management of resources is well supported by the constructor-destructor mechanism of C++. This is the idiomatic (that is, best fitting the language C++) way of creating and freeing of resources. Let us consider this example:

```
void bar();
void foo() {
    char *pt = new char[1024];
    bar();
    delete[] pt;
}
```

It seems to be correct: at the beginning of the process, a char array is allocated, and before returning it is carefully freed. (The pair of brackets `[]` after **delete** denotes our intention to free an array, not a single object.) What happens

however if the called function throws an exception? The flow of control leaves the function *foo* without the array's being freed. Since *pt* is the only pointer to this memory area, it cannot be freed any longer. The unexpected exception disturbs our program's behavior: it is not *exception-safe*. We can patch our program by catching the exception:

```
void bar();
void foo() {
    char *pt = new char[1024];
    try {
        bar();
        delete[] pt;
    } catch (...) {
        delete[] pt;
        throw;
    }
}
```

A similar example in Java is more elegant due to the try-finally pair of blocks. Code written in the *finally* block is always executed, regardless of whether the exception has been thrown.

```
void foo() {
    try {
        get_resource();          // allocation of the resource
        bar();                   // may throw exception
    } finally {
        release_resource();      // releasing of the resource
    }
}
```

An even better solution of the C++ example is to encapsulate the resource into a single object, making use of the automatic insertion of the destructor calls

for the automatic objects falling out of scope:

```
class mypuffer {
    char *pt;
    mypuffer(const mypuffer&);          // forbid copying
    mypuffer& operator=(const mypuffer&); // forbid assigning
public:
    mypuffer(int size) { pt = new char[size]; }
    ~mypuffer() { delete [] pt; }
};
void bar();
void foo() {
   mypuffer a(1024);
   bar();
}
```

This technique is called *Resource Allocation Is Initialization* or *RAII*.

Fortunately, we do not have to reinvent the wheel. In C++ a whole branch of *smart pointers* of the standard library (e.g. the class templates *unique_ptr*, *shared_ptr* and others) serves the purpose of the exception-safe handling of automatic pointers. A smart pointer wraps a raw pointer, and it can be used similarly to pointers. When destroyed, it also destroys the object pointed to.

The *auto_ptr* defined in the earlier C++ standard has a number of issues, thus its usage is better avoided.

4.5 Summary

Scope and lifespan are two related but not overlapping major concepts in programming languages. The scope of an identifier defines the section of the source code from where a named language element (an object, a function, a type, etc.) is accessible using that identifier. Scope categories exist from local scope to class, namespace, compilation unit, and all program wide visibility with many variations in different programming languages. As well-chosen names are essential resources, programmers should select the adequate scope category to manage the program's identifiers and to help the compiler detecting possible errors.

The lifespan (or simply: life) of an object is the part of the program's running time between the creation and the disposal of the given object. Lifespan categories are typically determined by the storage types used by the programming language. Most programming languages use automatic, static, and dynamic storage. Automatic objects are constructed when the program control enters the block where they have been declared and are disposed of when the control leaves the block. Static storage (with global, namespace or even local scope) is allocated at the beginning of the program (although details vary in different languages)

and remains available during the whole run of the program. In case of dynamic storage, allocation and deallocation are under the control of the programmer, although proper resource handling is sometimes supported by either a garbage collection mechanism or by language elements, like smart pointers. Modern object-oriented languages support the object's creation and disposal with user defined constructors and destructors.

4.6 Exercises

Exercise 4.1. The languages mentioned in this chapter are compiled ones. Consider interpreted (script) languages, for example AWK, Perl, JavaScript, or Unix-shell (sh, ksh, csh, bash, zsh and so on) and examine to what extent the contents of the chapter apply to them. Check if, for example, there are non-global variables in them at all.

Exercise 4.2. In the example on page 139find a way for regaining the value of the pointer *pt* in the code example. Rewrite the example to avoid the use of casting.

Exercise 4.3. Write the *ctime1* and *ctime2* functions alluded to in the *ctime* example in Section 4.4. Give an object-based solution too.

Exercise 4.4. Rewrite the "resource allocation and freeing" example from Section 4.4 to include not only one, but three resource-objects in C++, and in Java before and after version 1.7, that is, without or with a **try**-*with*-*resources* statement.

Exercise 4.5. Locate the code block initializing the static variables in the example *ClassWithStaticObject* in Section 4.3.2.

```
javac ClassWithStaticObject.java
javap -private -c ClassWithStaticObject
```

Exercise 4.6. Test the following C language example with various numbers of command line parameters and discuss the output:

```
#include <stdio.h>
int foo (int i)
{
    if (i <= 0) {
        /* deliberately with no initial value */
        int j;
        return j;
    }
    return foo(i - 1);
}

int main (int argc, char *argv[])
{
    printf ("%3d %8x\n", argc, foo (argc));
    return 0;
}
```

4.7 Useful Tips

Tip 4.1. We will give a solution by analyzing the JavaScript language.

Tip 4.2. There are three different solutions to this problem.

- Using cast as above;
- Copying raw bytes using the standard library *memcopy*function;
- Using union.

Tip 4.3. In the object-oriented version you can implement the buffer as the member variable of the class.

Tip 4.4. Do not forget to forbid the copy of RAII object in the C++ solution. In java use *AutoCloseable* interface.

Tip 4.5. Compile the class and then use the Java class file disassembler tool *javap* with the *-private* option.

Tip 4.6. Recall the different memory models and initialization strategies.

4.8 Solutions

Solution 4.1. We will give an exemplary analysis of the JavaScript language. Let us image we want to do some elementary ASCII art with Javascript functions in an HTML page:

```
<html>
<head>
<script>
function doit() {
    document.frm.f.value = makeIt();
}
function makeIt() {
    ret = "";
    for(i=0;i<10;i++) {
        ret += makeOne(i) + "\r\n";
    }
    return ret;
}

function makeOne(ind) {
    w = "";
    for(i=0;i<=ind;i++) w += i;
        return w;
}
</script>
</head>
<body>
<form name="frm" onsubmit="return false;">
<textarea name="f" rows="20" cols="20">
</textarea>
<button onclick="doit()">doit!</button>
</form>
</body>
```

The idea is to ”compute” the multiline text by concatenating its lines, but the outcome is not exactly what we expect:

```
0
012
01234
0123456
012345678
```

The point is that in Javascript a variable, even if declared or assigned a value inside a function, like the loop variables i in the functions *makeIt* and *makeOne*, is a global one unless declared with the *var* keyword. So in our case we use the same global i in these functions as loop variable, so these functions interfere with each other through the side effect conveyed by this coupling variable. There is a general principle to put the variables in the narrowest possible scope in order to avoid situations like this and also to avoid littering a broader scope with unneeded names. Having violated this principle, we introduced the bug.

The remedy can be to fix the loops like this:

```
for(var i=0;i<10;i++) ...
```

and

```
for(var i=0;i<=ind;i++) ...
```

so the loop variables are local ones and our ASCII art unfolds itself in its full beauty:

```
0
01
012
```

```
0123
01234
012345
0123456
01234567
012345678
0123456789
```

Solution 4.2. This file contains a solution for casting, copy memory and using union.

```
#include <stdio.h>
#include <string.h>

void foo(void); // test funcion with casting
void bar(void); // test funcion using memcpy
void baz(void); // test funcion using union

int main(int argc,char *argv[] )
{
    foo();
    bar();
    baz();
}
void foo() // with casting as in the book
{
    char *pt = new char();
    char pointer0[sizeof(pt)];
    char pointer1[sizeof(pt)];
    *(char **)pointer0 = pt; // Bitwise copy of pt.
    *(char **)pointer1 = pt; // Bitwise copy of pt.

    printf("%d\r\n",__LINE__);
    printf("%p\r\n",pt);
    pointer0[0] = 0xab;
    pointer1[1] = 0xcd;

    pt = 0;

    printf("%p\r\n",pt);
    pointer0[0] = pointer1[0];
    pt = *(char **)pointer0;
    printf("%p\r\n",pt);
#ifdef FREE
    delete pt;
#endif
}
void bar() // using memcpy
{
    char *pt = new char();
    char pointer0[sizeof(pt)];
    char pointer1[sizeof(pt)];
    printf("%d\r\n",__LINE__);

    memcpy(pointer0,&pt,sizeof(pt));
    memcpy(pointer1,&pt,sizeof(pt));
    printf("%p\r\n",pt);
    pointer0[0] = 0xab;
    pointer1[1] = 0xcd;
    pt = 0;

    printf("%p\r\n",pt);
    pointer0[0] = pointer1[0];
    memcpy(&pt,pointer0,sizeof(pt));
    printf("%p\r\n",pt);
```

```
#ifdef FREE
    delete pt;
#endif
}

void baz() // using union
{
    char *pt = new char();
    union
    {
        char *pt0;
        char pointer0[sizeof(pt)];
    };
    union
    {
        char *pt1;
        char pointer1[sizeof(pt)];
    };

    printf("%d\r\n",__LINE__);
    printf("%p\r\n",pt);
    pt0 = pt;
    pt1 = pt;

    pointer0[0] = 0xab;
    pointer1[1] = 0xcd;
    pt = 0;

    printf("%p\r\n",pt);
    pointer0[0] = pointer1[0];
    pt = pt0;
    printf("%p\r\n",pt);
#ifdef FREE
    delete pt;
#endif
}


/* compilation and results:

$ g++ -dumpversion
3.4.6

$ g++ *.cpp

$ ./a.out
19
0x99e8008
(nil)
0x99e8008
37
0x99e8018
(nil)
0x99e8018
63
0x99e8028
(nil)
0x99e8028

$ g++ -DFREE *.cpp

$ ./a.out
19
0x83b6008
(nil)
```

```
0x83b6008
37
0x83b6008
(nil)
0x83b6008
63
0x83b6008
(nil)
0x83b6008

$ g++ -dumpversion
4.1.2

$ g++ a.cpp
$ ./a.out
19
0xe843010
(nil)
0xe843010
37
0xe843030
(nil)
0xe843030
63
0xe843050
(nil)
0xe843050

$ g++ -DFREE a.cpp

$ ./a.out
19
0x1b79c010
(nil)
0x1b79c010
37
0x1b79c010
(nil)
0x1b79c010
63
0x1b79c010
(nil)
0x1b79c010


//// ------------

bcc a.cpp
Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
a.cpp:
Warning W8057 a.cpp 11: Parameter 'argc' is never used in function
    main(int,char * *)
Warning W8057 a.cpp 11: Parameter 'argv' is never used in function
    main(int,char * *)
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

a.exe
19
00902C88
00000000
00902C88
37
00902C98
00000000
00902C98
```

```
63
00902CA8
00000000
00902CA8

bcc -DFREE a.cpp

Borland C++ 5.5.1 for Win32 Copyright (c) 1993, 2000 Borland
a.cpp:
Warning W8057 a.cpp 11: Parameter 'argc' is never used in function
    main(int,char * *)
Warning W8057 a.cpp 11: Parameter 'argv' is never used in function
    main(int,char * *)
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

a.exe
19
00902C88
00000000
00902C88
37
00902C88
00000000
00902C88
63
00902C88
00000000
00902C88

*/
```

Solution 4.3. The C-like solution is as follows:

- ```
  /*
  We make use of the standard C functions in (time.h)
  in particular of this:
  size_t strftime (char* ptr, size_t maxsize, const char* format,
         const struct tm* timeptr );
  */
  #include <stdio.h>
  #include <time.h>
  void long_running_function(void);
  void foo(void);
  void bar(void);
  void baz(void);
  void ctime1(char *buffer, const time_t *tp) {
      struct tm * p = localtime(tp);
      strftime(buffer,24,"%a %b %d %H:%M:%S %Y",p);
  }
  void ctime2(char *buffer, int bufflen,const time_t *tp) {
      struct tm * p = localtime(tp);
      strftime(buffer,bufflen,"%a %b %d %H:%M:%S %Y",p);
  }
  int main(int argc,char *argv[])
  {
      foo();
      bar();
      baz();
  }
  void foo(void)
  {
      char *before; char *after; time_t bt, at;
      bt = time(NULL);
      before = ctime(&bt); /* 1 */
      long_running_function();
  ```
  ```

```
    at = time(NULL);
    after = ctime(&at); /* 2 */
    printf ("%s %s\n",before,after);
}
void bar(void)
{
    char before[32];
    char after[32];
    time_t bt, at;
    bt = time(NULL);
    ctime1(before,&bt); /* 1 */
    long_running_function();
    at = time(NULL);
    ctime1(after,&at); /* 2 */
    printf ("%s %s\n",before,after);
}
void baz(void)
{
    char before[20]; // deliberately too small in order to demonstrate
    char after[20];  // the usefulness of the size parameter.
    time_t bt, at;
    bt = time(NULL);
    ctime2(before,sizeof(before),&bt); /* 1 */
    long_running_function();
    at = time(NULL);
    ctime2(after,sizeof(after),&at); /* 2 */
    printf ("%s %s\n",before,after);
}
void long_running_function(void) {
    clock_t when = clock() + CLOCKS_PER_SEC; // a second
    while( when >= clock());
}
```

The object-oriented solution in C++ contains:

- The class definition:

```
#include <time.h>

class Ctime
{
private:
   char buffer[25];
public:
   Ctime(const time_t *tp);
   const char *getBuffer();
};
```

- The class implementation:

```
#include "Ctime.h"
#include <time.h>

Ctime::Ctime(const time_t *tp)
{
    struct tm * p = localtime(tp);
    strftime(buffer,sizeof(buffer),"%a %b %d %H:%M:%S %Y",p);
}

const char * Ctime::getBuffer()
{
    return buffer;
```

```
}
```

- The test program:

```
#include "Ctime.h"
#include <time.h>
#include <stdio.h>

void long_running_function(void);

int main(int argc,char *argv[])
{
    char before[32];
    char after[32];
    time_t bt, at;
    bt = time(NULL);

    Ctime ct1(&bt);
    long_running_function();
    at = time(NULL);
    Ctime ct2(&at);
    printf("%s %s\n",ct1.getBuffer(),ct2.getBuffer());
}

void long_running_function(void)
{
    clock_t when = clock() + CLOCKS_PER_SEC; // a second
    while( when >= clock());
}
```

Solution 4.4. The C++ solution without RAII:

```
void bar();
void foo()
{
    char *pt = new char[1024];
    char *pt1 = new char[1024];
    char *pt2 = new char[1024];

    try
    {
        bar();
        delete[ ] pt;
        delete[ ] pt1;
        delete[ ] pt2;
    }
    catch (...)
    {
        delete[ ] pt2;
        delete[ ] pt1;
        delete[ ] pt;
        throw;
    }
}
```

The C++ solution with RAII:

```
class mypuffer
{
    char *pt;
    mypuffer(const mypuffer&);             // forbid copy
    mypuffer& operator=(const mypuffer&);  // forbid assignment
public:
    mypuffer(int size) { pt = new char[size]; }
    ~mypuffer() { delete [ ] pt; }
};
void bar();
void foo()
{
    mypuffer a(1024);
    mypuffer b(1024);
    mypuffer c(1024);
    bar();
}
```

The java solution before version 1.7

```
/*
An exception can be thrown during the allocation of a resource
We declared random checked exceptions from the java.lang package
*/
public class ex4a
{
    // this is a method that must be called on the object
    // before it gets garbage collected
    public void close()
    {
    }

    // in a production code you might want to be more specific
    // about the possible Exceptions
    void doit()  throws Exception
    {
    }

    // resource allocation
    ex4a getResource1() throws IllegalAccessException
    {
        return new ex4a();
    }

    ex4a getResource2() throws ClassNotFoundException
    {
        return new ex4a();
    }

    ex4a getResource3() throws InstantiationException
    {
        return new ex4a();
    }

    // in a production code you might want to be more specific
    // about the possible Exceptions
    public void foo() throws Exception
    {
        ex4a res1 = null;
        ex4a res2 = null;
        ex4a res3 = null;

        try
        {
            res1 = getResource1();
```

```
            res2 = getResource2();
            res3 = getResource3();
            doit(); // do something here potentially throwing an exception
        }
        finally
        {
            if ( res3 != null )
            {
                res3.close();
            }
            if ( res2 != null )
            {
                res2.close();
            }
            if ( res1 != null )
            {
                res1.close();
            }
        }
    }
}
```

The java solution using version 1.7 features

```
public class ex4b implements AutoCloseable {

    public void close() throws Exception{}

    public void doit( ) throws Exception{}

    ex4b getResource1() throws IllegalAccessException {
        return new ex4b();
    }

    ex4b getResource2() throws ClassNotFoundException {
        return new ex4b();
    }

    ex4b getResource3() throws InstantiationException {
        return new ex4b();
    }

    void foo() throws Exception {
        try (
            ex4b  res1   = getResource1();
            ex4b  res2   = getResource2();
            ex4b  res3   = getResource3();
        )
        {
            doit( ); // do something here potentially throwing an exception
        }
    }
}
```

Solution 4.5.

```
public class ClassWithStaticObject {
    static int staticObject[] = { 123, 456 };
}

javap -private -c ClassWithStaticObject

Compiled from "ClassWithStaticObject.java"
public class ClassWithStaticObject extends java.lang.Object{
static int[] staticObject;
```

```
public ClassWithStaticObject();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>":()V
   4: return

static {}; // static class initializer block
  Code:
   0: iconst_2
   1: newarray int
   3: dup
   4: iconst_0
   5: bipush 123
   7: iastore
   8: dup
   9: iconst_1
   10: sipush 456
   13: iastore
   14: putstatic #2; //Field staticObject:[I
   17: return
}
```

Solution 4.6.

```
#include <stdio.h>

int foo (int i)
{
    if (i <= 0)
    {
        /* deliberately with no initial value */
        int j;
        return j;
    }
    return foo(i - 1);
}
int main (int argc, char *argv[])
{
    printf ("%3d %8x\n", argc, foo (argc));
    return 0;
}
```

A possible output looks like the following:

```
$ ./a.out
  1 bffa4270

$ ./a.out 0
  2        0

$ ./a.out 0 1
  3        0

$ ./a.out 0 1 2
  4   5b4e70

$ ./a.out 0 1 2 3
  5   2ac6a4

$ ./a.out 0 1 2 3 4
  6   11ca08

$ ./a.out 0 1 2 3 4 5
  7   e98aa7

$ ./a.out 0 1 2 3 4 5 6
  8   5b8573
```

The core of the example is this function:

```
int foo (int i)
{
    if (i <= 0) {
        /* deliberately with no initial value */
        int j;
        return j;
    }
    return foo(i - 1);
}
```

It calls recursively itself diminishing the parameter with each iteration, and returning a random value from the stack when bottoming out. Depending on the argument, these random values will be different, due to their being grabbed from different (uninitialized) locations of the stack.

5 Data types

Data type is a central notion of modern programming methodologies. Data types are the abstraction that enable programmers to formulate their algorithms using close to real life entities instead of sequences of bits. Programming languages provide a rich set of built-in data types. This chapter provides an overview of these types and their categorization, then it takes a look at the rules of forming and evaluating expressions.

Any problem that we try to solve using a computer is somehow related to the "real world" around us. However, the programs that we create to solve these problems operate within computers. Therefore, one of the most important steps of solving a problem is modeling the objects of the real world within a computer. Then, we can solve the problem using the computer on these model objects, and finally we can project the result to the real world, and interpret it there.

The world around us is extremely complex. Modeling it in full detail would require a system of the same size as the world. When modeling real life objects we use *abstraction*, i.e. we only include those attributes of an object in our model which we deem relevant for the solution of the problem. This way we create groups of objects that behave similarly from the problem's point of view. This similarity is expressed by *data types*.

Programming languages have built-in data types, which model some abstract notions of the real world such as numbers, letters, text etc. These objects frequently appear in all sorts of problems, building them into the programming languages enable reusing them and greatly simplify the programming task. In this chapter, we will study these built-in data types. Most programming languages provide means to create new data types as well. The construction methods of these new data types are discussed in Chapter 6.

5.1 What is a data type?

Before starting the categorization of data types, we need to define what a data type is. The introduction has demonstrated that a data type is a very complex notion, but the informal description provided there is not suitable for deeper study. To tackle the complexity, we will use multiple definitions – from different points of view – which together will help us to get a better understanding of data types.

5.1.1 The programming language perspective

Data type: an abstraction which collects the common attributes of certain objects of the program. These attributes are *encoding*, *size*, *structure* and on a higher level, *semantics*. Semantics is defined by the type-value set and the operations (or methods) of the type.

This definition describes data types from the point of view of programming languages. The language (i.e. the compiler or interpreter) needs to know how the type is represented in memory (encoding), how much memory an object of the type occupies (size). In case of composite types, the language needs to know the internal structure of the type to make its components accessible. And finally, it needs to know the semantics, at least on a lower level – e.g. for the integer numbers, the programming language provides the basic operations.

Naturally, these four aspect appear on the programmers' side as well, but with very different emphasis.

5.1.2 The programmers' perspective

We will use a more formal description to define data types from the programmers' point of view. This definition separates the description of external behavior (type specification) from the actual implementation. The model and the methodology used in this chapter is described in details in [Fot83].

Type specification

The system $\mathcal{T}_S = (T, I_S, \mathbb{F})$ is called *type specification* if

- T denotes an arbitrary base set,
- $I_S : T \to \mathbb{L}$[1] is the invariant of the specification, $T_S = [I_S] = \{t \in T | I_S(t) = true\}$ is called the *type-value set*,
- $\mathbb{F} = \{F_1, F_2, \ldots, F_n\}$ is a set of problems – called *operations* or *methods* of the type – where T_S is part of the *state space* of F_i, i.e. which operate on T_S.

Type specification answers the question "what". It describes what the elements of the type are and what its operations do. Type specification provides the programmer with all the information required to use the type.

From the above mentioned four attributes, type specification contains semantics only. The base set T is a set of real life objects relevant to the current problem. The invariant of the specification (I_S) can further limit this set as, for instance the limitations of the computer prevent us from representing the whole set. For example, if T is the set of natural numbers, the invariant of the specification might narrow it to a finite interval (e.g. $0 \ldots 2^{32} - 1$), because the finite memory of a computer cannot represent indefinitely large numbers. If T

[1] $\mathbb{L}$ is the set of boolean values i.e. $\mathbb{L} = \{true,\ false\}$

is the set of real numbers, the limitation is stricter, not even an interval. The invariant of the specification selects a subset of the base set called the *type-value set*. This is typically finite, though sometimes it can be numerable, e.g. when we don not want to limit the size of the files in the specification of some file type. The limitations that are represented by the invariant of the specification are important for the user of the type. Therefore, these limitations must be part of the type specification, that is, they cannot be left for the implementation.

The third component of the system describes the relevant behavior of the objects of the type, the so called *operations*. Important that this description is declarative, it specifies *what* the operations do but leaves the *how* unspecified.

Notice that the valid values (type-value set) and the permitted operations on those values are specified together. For the built-in types, separating these two is not even possible, but this unity should be maintained in user-defined types as well.

Type realization

The system $\mathcal{T} = (\rho, I, \mathbb{S})$ is called *type* if

- $\rho \subseteq E^* \times T_S$ is the representation function (relation), where T_S is the type-value set and E is some elementary type-value set.
- $I : E^* \to \mathbb{L}$ is the invariant of the type
- $\mathbb{S} = \{S_1, S_2, \ldots, S_n\}$ are programs, for which E^* is in the state space.

This system describes the realization of the type. E is an elementary type-value set and E^* is the set of finite sequences created from the elements of E. E is a type-value set which is already available in our system. It can be an already implemented user-defined type or a built-in type – simple or composite. At the lowest level, E is the $\{0, 1\}$ set of bits as all data types are represented as sequences of bits. However, most of the time, we use higher-level data types when defining the representation of a new data type.

Representation function

The first component of the system is ρ, the representation function. Formally it is a relation because to the elements that do not satisfy the invariant of the type it can assign anything. However, in practice, it is often a function or partial function. It assigns to a sequence of elementary values the type-value they represent. The definition implies that not all sequences of elementary values represent a type-value, and some type-values can have multiple representations. Let us suppose, for example, that we need to represent a subset of rational numbers. The representation we choose is a sequence of two integer numbers, where the first number is the numerator the second one is the denominator of the rational number. In this representation, we do not assign any rational number to a sequence where the second element is zero. On the other hand, all (kp, kq)

pairs (where $k, q \neq 0$) will represent the same p/q rational number (where p and q are relative primes).

Invariant of type

The third element of the system (I) is the invariant of the type. This property describes which sequences of elementary values represent a type-value. Typically not all sequences are valid, they have to satisfy certain internal constraints. In the example above we have seen that only those pairs of integer numbers were valid representation of a rational number where the second number – the denominator – was non-zero. We may further restrain the representation by demanding that the numerator and the denominator are relative prime or that the denominator must always be positive.

Operations

The third component of the system is the set of type operations. These operations are programs which state-spaces include the sequences of elementary values.

There is a close relationship between the invariant of the type and these type operations. If a program has an outbound parameter of a sequence of elementary values, the program – in its postcondition – must ensure that the resulting sequence satisfies the invariant. On the other hand, if the program has an inbound parameter of a sequence of elementary values, the program may require in its precondition that the sequence satisfies the invariant. Clearly, the implementation of the program depends both on the representation function and the invariant of the type. Often these two together are referred to as the *representation of the type*.

Types and specifications

We will now determine when a type is adequate to a type specification. The formal definition is the following:

The $\mathcal{T} = (\rho, I, \mathbb{S})$ type *is adequate to* the $\mathcal{T}_S = (T, I_S, \mathbb{F})$ type specification if:

- $\rho([I]) = T_S$, and
- $\forall F \in \mathbb{F} : \exists S \in \mathbb{S} : S$ is a solution of F through ρ.

The first condition means that all sequences of elementary values which satisfy the invariant of the type represents a valid type-value, i.e. one that satisfies the invariant of the specification. Additionally, there is a valid representation for all type-values in the set T_S.

The second condition is that for each specified operation there is a program in the type which implements it. The ”solution through ρ” means that the input data, which is given in the state space of F is transformed to state space of the

solution program S using the inverse relation of ρ. There we execute S and then the result is mapped back to the state space of F using ρ. If the result satifies F, then S is a solution to F through ρ, that is, it is a valid implementation of the specified operation using the given representation. In other words, a program S is a solution through ρ to F if it modifies the sequences of elementary values in such a way that the interpretation of the results matches the expected result specified in the problem.

Notice that many types can be adequate to the same type specification, which may differ in the chosen representation, in the invariant of the type – see the example above – or in the implementation of operations. Our mathematical model only handles the functional correspondence between the type and the specification. Which realization of a type is appropriate in a particular application may depend on non-functional aspects such as execution time, memory footprint, etc.

5.1.3 Type systems of programming languages

The objects our programs operate on have a type, which determines their behavior, the set of operations on them. Essentially, a type is a set of semantic rules, which specify what operations are *permitted* on the bit sequence representing the object. If we perform an illegal operation on an object, it is a clear semantic error in our program. One of the most important goals of programming languages is to help the programmer write correct programs, to help discovering semantic errors early, preferably in *compilation time*.[2] One of the means of performing semantic verifications in the programs is the type system and *typedness* of programming languages [CW85].

Typedness means that programming languages assign a type to the entities (objects) used in the programs. All constants, operators, variables or functions have a type (in some languages other constructs can have a type as well). The type of expressions is determined using a *type inference* system. In some languages – for example Ada or Pascal – the type of symbols are specified in their declaration and the compiler verifies that the definition and the usage of these symbols is consistent. In other languages – for example in ML – instead of using explicit declarations, the compiler determines the type from the expressions as long as its possible and as long as consistency can be maintained.

In *statically typed programming languages* the type of all expression can be determined at compilation time. While static typedness is a useful propery, it imposes too strict requirements against languages. Fortunately, these requirements can be loosened a bit while maintaining consistency. A programming language is called *strongly typed* if it guarantees that the type of all expressions is consistent, even if the exact type of the expression cannot be determined at compilation time. Clearly, all statically typed languages are strongly typed. However, there

[2] Compilation time is a loose term in this chapter. It is used for interpreted languages as well. It refers to the time of syntactic verification of the program.

are strongly typed languages which are not statically typed. Languages which support inheritance based subtype polymorphism cannot be statically typed though many of them – for example C++, Java or Eiffel – are strongly typed. Different variants of polymorphism are discussed further in details in Section 11.1 and Chapter 10.

In other languages the consistency of expressions can only be determined at runtime. Therefore, evaluating such expressions might cause runtime errors. Such languages are Smalltak, dBase, or many scripting languages. These langauges are called *weakly typed*.

5.1.4 Type conversions

When developing an application, we often need to change the type of some data. This operation is called *type conversion*, *typecast* or *coercion*. Depending on the programming language and the actual types, the conversion can happen automatically (coercion) or it can be explicit. There are different variants of conversion which are further described in the following sections.

Changing representation

Type conversion can serve multiple purposes. One possible purpose is to change the data representation. As we have seen in previous sections, an object can have numerous different representations in memory. When solving a concrete problem, we try to choose the most suitable representation, the one that serves our goals the best. However, some objects may have multiple different representation in our program. The simplest examples are numbers. Most modern programming languages offer some integer data types using two's complement representation and some real data types using floating point data representation. This means that for integer numbers, which are also real numbers, we have two distinct representations in these languages. Though the represented "real world" object is the same, the available set of operations depends on the chosen representation. Therefore, to perform some operations we might need to convert the data from one representation to the other.

In some programming languages – such as Ada – these conversions must always be explicitly denoted. In other languages, some of these conversions can take place automatically. In such languages, when evaluating the $2.2 + 1$ numeric expression, the representation of the integer number 1 is automatically converted to floating point representation and the addition is performed on floating point values. This conversion is called *widening* because we change from a narrower type-value set of integer numbers to a wider set, the set of real numbers. Widening conversion happens when a 16-bit integer value is converted to 32-bit or a **float** value to **double**. Many languages allow automatic widening conversions because it is safe, they can be performed without data loss. This is also the reason why the integer value 1 gets converted in the expression above.

Converting a floating point value (2.2) to integer has the risk of loosing some precision, because it is a conversion to a more limited type-value set; it is a *narrowing conversion*.

The example below is a syntactically correct code snippet in C, C++, and Java as well.

```
double x;
int i;
x = 2.2 + 1;
i = 2.2 + 1;
```

When calculating the value of x, in all three languages a widening conversion is performed automatically. However, the calculation of i requires two conversions. The first is a widening conversion required to evaluate the expression. As the type of the expression will be **double**, before the assignment to i, another conversion is required. This is a narrowing conversion from the floating point expression type to the integer type of the variable. This conversion is not always safe to perform. In the example, we loose precision when the original value of the expression 3.2 is converted to the integer 3. In language C this narrowing conversion is automatic. In C++ the conversion is automatic, but the compiler warns of the potential data loss during compilation. However, in Java the second assignment is illegal and it results in a compilation error.

String conversions

Many programming language offers means to convert various objects to String and convert Strings to objects. Though they change the representation of the object, they are different from the representation changing conversions described above. String conversion does not only change the representation, they actually change the represented ”physical” object. When the number 42 is converted to a string, the result – at least in most programming languages – is not a different representation of the number 42 but a string of characters – a piece of text – which consists of the characters ‘4’ and ‘2’. Not only does the representation change but the represented object as well. Depending on the type and language, the conversion may be irreversible. Examples for such string conversions are the *Image* attribute of Ada or the *toString()* method of Java objects. In Ada the conversion to String is always explicit while Java permits automatic conversion as long as the result in not ambiguous.

Changing the interpretation

The conversions that change the representation of an object transform the bit sequence that describes the object in memory. A different variant of type conversions leaves the actual bit sequence intact while changes its *interpretation*. While the former type of conversions results in some runtime operation, interpretation

changing conversions are purely compile time constructs. Their purpose is to "calm down" or "work around" the type verification system of the compiler.

In object-oriented languages, the conversions between a subclass and a superclass is an interesting mixture of representation- and interpretation changing conversions. In languages like Java, where objects are stored by reference, the conversions between subclass and superclass are possible in both directions, provided that the dynamic type of the object (see Chapter 10) permits it. In practice it is an interpretation changing conversion as the bit sequences representing the object – both the reference and the referenced memory area – remain unchanged. However, if the objects are stored by value – for example, in C++ or the expanded objects in Eiffel –, then the conversion is only allowed from subclass to the superclass, and it involves changing the representation because the data members introduced in the subclass are truncated. Here the effect of narrowing and widening conversions is modified as well. Converting from subclass to superclass is a widening conversion, but it may result in data loss. The conversion itself is safe, but the narrowing conversion is not possible later.

Type conversions in the languages

C

In C the *type cast* operator can be used for both the representation changing conversions between the different numeric types – though denoting these explicitly is not required – and for the interpretation changing conversions between the various pointer types and the **int** type. Any pointer can be converted to any other pointer or **int** as this only involves changing the interpretation of the representing bit sequence. However, they cannot be converted to floating points types because it would involve changing the bit sequence as well. C also permits interpretation changing conversions between the constant and variable versions of a type as it leaves the representing bit sequences intact.

C++

In C++, for backwards compatibility, the type cast operator of C is available but its usage is not recommended. Type casts in C are inherently dangerous, they are mostly interpretation changing conversions, which effectively disable the rather limited type safety of C. Using them requires careful consideration. Instead of this single multi-purpose cast operation, C++ has introduced a number of different type cast variants which are tailored for the different use cases of type conversion. This separation enables selecting the conversion operator which fits the required purpose the best.

- The **static_cast** operator can be used for conversions between related types, e.g. the pointer types of classes within the same type hierarchy, between numeric types or between integer and enumeration types.

- The **dynamic_cast** operator is also used for conversions between pointer types of classes within the same type hierarchy. However, at runtime when converting from the superclass to the subclass this operator verifies that the dynamic type of the referenced object also permits the conversion (see Chapter 10).
- The **const_cast** operator is used to convert a constant object to a non-constant object of the same type.
- The **reinterpret_cast** operator can be used to convert between types where the only relationship between the types is that they are represented by bit sequences of the same length. This corresponds to the original type cast operation of C. Important to note that while the other three cast variants are portable, the usage of **reinterpret_cast** may result in platform or implementation dependent code.

Java

In Java the conversion between scalar types is representation changing. In the case of widening conversions, it is automatic, while in the case of narrowing conversions, it needs to be explicitly denoted. Between reference types the conversions are limited. From subclass to superclass the conversion is automatic. Conversion from a class to the interfaces it implements is also automatic. The reverse conversions are possible but they include runtime type checking. During the check the Java Virtual Machine verifies that the dynamic type of the referenced object is assignable to the type it is being converted to. These conversions need to be explicitly denoted in the code and if the type checking fails, a runtime exception will get thrown.

Since Java 5.0 there is also a conversion between scalar types and their corresponding Java wrapper type. This is an automatic representation changing conversion called *autoboxing*.

Eiffel

In Eiffel, variables can store objects either by reference or by value. When storing objects by reference, the variable only contains a pointer to the area in memory where the attributes of the object are stored. When the object is stored by value, the variable actually contains the attributes of the object. These objects are called *expanded* in Eiffel. The rules of type conversions are essentially the same as described in the introduction, but their effect depends on how the object is stored. Conversion from subclass to superclass is automatic, but in the case of expanded objects, it results in data loss; data members – called features in Eiffel – introduced by the subclass are truncated. Conversion from superclass to subclass for expanded objects is not permitted. For objects stored by reference, it is possible to convert from the superclass to subclass provided that the dynamic type of the object is really assignable to the subclass. This is done by using the

reverse assignment attempt operator. If the dynamic type of the object is not assignable to the left-hand side of the assignment, the value of the variable will be *void*.

CLU

In CLU, there is a very special interpretation changing conversion. When creating a new abstract data type, the type system of CLU differentiates the abstract type from the concrete type used for its representation, and mandates an explicit type conversion between them. For further details see Section 9.5.3.

Ada

Finally the conversion between the base type and the derived types in Ada is an interpretation changing type of a conversion as the derived type in Ada has inherited the representation of the base type. Using conversion it is possible to create connection between the derived types of the same base type (for further details see Section 5.8.1).

5.2 Taxonomy of types

This section will provide a classification of types based on their structural properties.

On the highest level of the taxonomy there are two classes. *Primitive types*, which are logically atomic, and have no identifiable parts, and *composite types*, which are constructed from other, already existing types. This chapter will focus on the former, that is the primitive types. Composite types are discussed in Chapter 6.

Primitive types can be divided into two classes. *Pointer types* are the abstractions of memory addresses, and are discussed in details in Section 5.6; *scalar types*, on the other hand, represent simple atomic values or quantities.

Scalar types can be grouped into *real* and *discrete* types. The former category contains the various representations of real numbers. The two most common variants are *floating point* and *fixed point* representations. Discrete types can be categorized as *enumeration* and *integer* types. Figure 5.1 gives an overview of this type taxonomy.

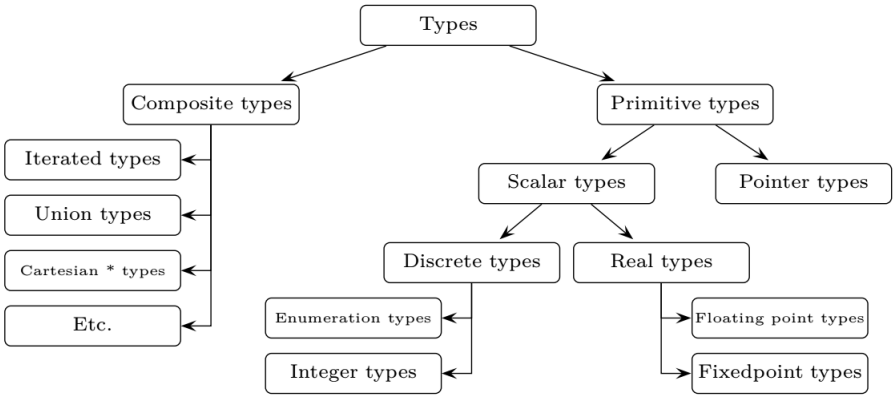


Figure 5.1: Type taxonomy

5.2.1 Type classes

Type classes defined in the taxonomy are not just of theoretical relevance. They often manifest themselves in the programming languages, and as they belong to a type class, it determines the usage of the type. Common constraints are that the loop variable of a **for** loop must be of a discrete type, or that arrays can only be indexed by discrete, or sometimes by integer types only. Type class can also determine the set of operations for the given type. Availability of some operations might be determined based in their belonging (or not) to the corresponding type class. When a new type is defined within the class, these operations become available automatically.

5.2.2 Attributes in Ada

Type classes are very important in Ada. Each type class has a set of operations, which are implicitly defined for each type in that type class. Part of these operations are operators, e.g. in Ada each scalar type is ordered and relational operators ($<$, $<=$, $=$, etc.) are implicitly defined for them. The other types of operations are the so called *attributes*. Attributes are special type class specific operations which are of three different kinds:

- There are attributes which are simple properties of the type. For example, for a scalar type S the attributes *S'First* and *S'Last* are the smallest and largest value of the type.
- Other attributes are operations of the type. These attributes have an object of the corresponding type in their signature, either as the type of an argument or as the type of their return value. Typical examples are the conversion operations. For any discrete type T the attributes *T'Pos* and *T'Val* are defined. These attributes are functions. *T'Pos* maps elements

of T to their ordinal number, while *T'Val* is an inverse operation which returns the type-value for the specified ordinal number.

- The third group of attributes is an interesting mixture of the previous two. Mixture, because they operate on objects of the type, but they return information about the type itself. For example, the attribute *'First* is defined for array types as well. For an array type A, *A'First* returns the lowest value of the index interval of the type. However, in Ada it is possible to define array types with indefinite index boundaries. For these types, the actual index interval is defined when a concrete instance – an array object – is created (see Section 6.6.4). The *'First* attribute is applicable to these array types and the objects as well, but in the case of such types, the object itself is used as qualifier.

The following example is the definition of the enumeration type *Dwarf*:

type *Dwarf* **is** *(Bashful, Doc, Dopey, Grumpy, Happy, Sleepy, Sneezy);*

We have defined *Dwarf* as an enumeration type; therefore, a number of operations are defined for it implicitly.

As it is a discrete type, it can be converted to integer type (to ordinal numbers) and back using the attributes *Dwarf'Pos* and *Dwarf'Val*.

As discrete types are scalars as well, an ordering is defined on the type *Dwarf*, and the relational operators (<, <=, etc.) may be used. It is possible to create intervals (*range*), and using the **in** and **not in** operators, we can decide if an element is part of an interval or not . Additionally, there are fourteen other attributes defined for it – specific to the scalar type class –, which enable, among others, getting the lowest and highest value of the type, conversions to and from a string, and finding the dwarf following another according to the ordering.

5.3 Scalar type class

Scalar types are the simplest types of programming languages. type-values are simple, they have no defined internal structure. Numerous types belong to this type class, from integer type to enumerations or real numbers. Therefore, the set of operations of these types vary on a wide spectrum. Nevertheless, there are some common properties for scalar types which can be studied for the whole type class.

5.3.1 Representation

The scalar type class is present in almost all programming languages in some form. Even object-oriented languages tend to differentiate scalar types from other classes.

One of the reasons for the special handling of scalar types is that these types are often not implemented by the language itself but at least partly they are

realized in hardware. All CPUs used today realize the integer type using two's complement representation, typically on 16, 32 or 64 bits. Most generic purpose CPUs also support real numbers by using floating point representation. Therefore, the most efficient way of realizing these types in programming languages is to reuse the support built directly into the CPU. Additionally, these are the most widely used types in programming solutions. Thus, their efficient realization is crucial in all practical applications.

As a result, programming languages usually handle scalar types very similarly in terms of representation and set of operations too. Even in those languages where complex types are stored by reference to the corresponding object in memory – such languages are, for example, Eiffel, Java or CLU –, scalar types are stored by value. In Eiffel these types are called *expanded types*. Some exceptions are Smalltalk or JavaScript where variables have no type and all objects, including scalar values, are stored by reference.

5.3.2 Operations

The most important common property of scalar types is that they are totally ordered. Therefore, the common operations are related to ordering:

- Relational operators ($<, \leq, >, \geq, =, \neq$) are usually defined;
- It is common to have operations or built in constants to determine the smallest and greatest element of a scalar type.

5.3.3 Scalar types in Ada

In Ada-95 scalar types have the following operations:

- Relational operators: $<$, $<=$, $>$, $>=$. $=$, $/=$
- Interval definition: *range* ⟨*lower bound*⟩..⟨*upper bound*⟩.
- Checking if a given value falls in an interval: **in**, **not in**

Additionally, for a scalar subtype S (see Section 5.8.1 for details) the following attributes are defined:

- *S'First* and *S'Last* denote the lowest and highest value of the subtype respectively.
- *S'Range* is the type-value set (*range S'First .. S'Last*).
- *S'Base* is the base type of the subtype S.
- *S'Min* and *S'Max* are functions with two arguments of type S. The functions return the minimum and maximum of the specified values respectively.

- *S'Pred* and *S'Succ* are unary functions which return the values preceeding and succeeding their arguments. The operations are not cyclic, both *S'Pred(S'First))* and *S'Succ(S'Last))* results a *CONSTRAINT_ERROR* exception. Associating these operations with the scalar type class is unusual. Real numbers also belong to scalar type class and a real value has no successor or predecessor, at least in mathematical sense. However, as we can represent a finite subset of real numbers only – both using fixed point and floating point representations –, it is possible to determine for each real value the previous or next "representable" real value. However, these values are representation dependent and their mathematical meaning is unclear.
- *S'Image* and *S'Wide_Image* unary functions, can convert a value of type *S* to a string. The function *S'Image* produces 8 bit (ASCII) character string while the other output of the other function is a 16 bit (ISO 10646) character string.
- *S'Value S'Wide_Value* unary functions convert a string to the corresponding values in *S*. These functions are the inverse operations of the respective *Image* attributes.
- The *S'Width* and *S'Wide_Width* attributes specify the length of string produced by the corresponding *Image* attributes.

5.4 Discrete type class

There are two kinds of discrete types: enumerations and integer types. Almost all programming languages have at least a few discrete types, though not all of them supports enumerations. In some languages such as Pascal this type class is called *ordinal types*. A very important property of this type class is that usually only discrete types can be used for indexing arrays and as the loop variable in **for**-like loops. Typical operations of the type class involve the following:

- Calculating the successor and predecessor of a scalar value. For integer types this means incrementing or decrementing the value by one but the operations can also be defined for enumerations to return the next or previous element according to their ordering. Programming languages differ on the basis whether they allow the values to wrap around.
- Converting to integer (ordinal number). Though this is meaningful for enums only, sometimes it is allowed for the whole discrete type class.

A good example for integer conversion operators is Ada, where these operators are attributes defined for the discrete type class. For a discrete type *D*

- the attribute *D'Pos* is a unary function mapping the type *D* to integer numbers. For integer types this function is identity. However, for enumerations this function returns the ordinal number of the specified value, i.e. the index of the value in the order of enumeration. Indexing starts at 0.
- The attribute *D'Val* is the inverse function of *D'Pos*, it determines the type-value for the specified ordinal number.

An example for using these attributes is handling generic arrays. In Ada, arrays can be indexed with any discrete type. Using the attributes above, we can create generic subprograms for arrays, e.g. a generic binary search, which can convert the index values to integer when determining the midpoint of an index interval.

5.4.1 Enumerations

The type-value set of enumeration types is defined simply by listing its elements. These types are usually represented by mapping the values to integers in the order of listing. Therefore, typically they have some operation to map to and from integer values. Enumerations are both a type class and a type construction technique. It is not listed among type constructs for two reasons. Firstly, they differ from the other constructions because they do not use already existing types for creating a new type. Secondly, programming languages which support this construction method often contain some enumerations built in.

Pascal-type languages usually support enumerations. The following example is the Pascal version of the previously defined Ada type *Dwarf*:

```
type Dwarf = (Bashful, Doc, Dopey, Grumpy, Happy, Sleepy, Sneezy);
```

On the other hand, many languages have no support for enumerations as they can relatively easily be implemented by using some integer type and defining constant values for the type-values. C and C++ even provide some syntactic support for this. The **enum** structure is a simplified method for defining integer constants. By default, the construct assigns successive integer values starting at 0, unless otherwise specified. In the example below the days of week will get the values from 0 to 6 while the special *NoSuchDay* element will get the value 999.

```
typedef enum { Monday, Tuesday, Wednesday, Thursday,
       Friday, Saturday, Sunday, NoSuchDay = 999
} DaysOfWeek;
```

The defined type *DaysOfWeek* can be used in type declarations of variables. However, if the application uses multiple enumerations – for example, we have a *Month* type as well –, nothing prevents us from assigning the value *Thursday* to a variable of type *Month*. Actually, any integer value can be assigned to them – we can just as well calculate *Tuesday + January* because all enumerations in these languages are essentially integer types, the **enum** construct is just syntactic sugar to simplify an often recurring task.

The **unique** declarations of Eiffel work similarly:

Red, Yellow, Green: INTEGER **is unique**;

CLU has a better solution for replacing enumerations. The language offers different union constructs (see Section 6.4.3). By using the construct *oneof* we can create a data type which behaves very similar to enumerations. When using this *labeled union* construct for enumeration types, all information is carried by the label itself. Therefore, it is useful to use the simplest type of the language, the *null* type, as the component type for all components. The only valid value of this type is *nil*, which is used to initialize the components. Though the result is a bit inconvenient to use, it provides type safety – elements of an enumeration cannot be assigned to a variable of another enumeration type.

```
Colors = oneof [ red, yellow, green : null ]
trafficlight : Colors := Colors$make_green(nil)
tagcase trafficlight
  tag red :
    trafficlight := Colors$make_green(nil)
  tag yellow :
    trafficlight := Colors$make_red(nil)
  tag green :
    trafficlight := Colors$make_yellow(nil)
end
if Colors$is_red(trafficlight) then
  Car$stop(my_car)
end
```

In the first versions of Java, there was no enumeration construct. Similarly to C or C++, integer or string variables and the corresponding constants are used to represent enumerations. However, the need of type safety has eventually lead to the development of the *type safe enum* pattern described in [Blo01]. This construct is a bit similar to the solution used in CLU: it uses named instances of a **final** class with a private constructor only.

```
public final class Color {
    static private int counter = 0;
    private final String name;
    private final int index;
    private Color(String name) {
        this.name = name;
        this.index = Color.counter++;
    }
    static public final Color RED = new Color("Red");
    static public final Color YELLOW = new Color("Yellow");
    static public final Color GREEN = new Color("Green");
    static private final Color[] elements = {RED, GREEN, YELLOW}
    public String toString() { return name; }
    public Color next() {
        return elements[(index + 1) % elements.length];
    }
}
```

Luckily, the developers of the language have also recognized this need, and in the 5.0 version of the language the *enum* type construct was introduced. Enumerations in Java are classes, their values are objects, stored by reference. Each enumeration implicitly extends the base class *java.lang.Enum*, and as there is no multiple inheritance in Java, they cannot extend other classes. However, they can implement interfaces, they can have their own methods, etc. The construct ensures that each named value is unique. This ensures that – unlike for other classes – the `==` operator can be used to check equality of enumeration values. Another difference is that enumerations can be used as labels in **switch**-es:

```
public enum Color { RED, YELLOW, GREEN;
    public Color next() {
        switch (this) {
            case RED: return GREEN;
            case GREEN: return YELLOW;
            case YELLOW: return RED;
        }
    }
}
```

5.4.2 Integer types

Integer types are present in almost all programming languages. Their representation and set of operations are very similar too, because in practice they use the integer arithmetics built into the CPU of the computer. As integer types are the most frequently used data types in programs, programming languages put big

emphasis on efficient implementation. This efficiency often leads to interesting compromise solutions in the types.

Type-value set

Programming languages usually support multiple different integer types. The differences between these types are usually the number of bits used to represent the values – which directly determines the length of the interval that can be represented. Other differences concern the questions whether the interval is symmetric around zero,[3] or whether it contains non-negative values only. In other words, whether the type is *signed* or *unsigned*.

Most programming languages follow these patterns. Even the number of bits used in the representation are similar, the typical values being 8, 16, 32 and 64. Sometimes the specification of the programming language does not explicitly define the number of bits to be used, but specifies a lower bound only (i.e. an integer type I must have at least 32-bit precision). However, these lower limits are typically the numbers specified above, and implementations of these languages rarely choose different values. To provide optimum performance, programming languages want to base their integer types on the integer arithmetics of the underlying CPU. Allowing for flexibility is an old practice dating back to times when CPU architectures were less uniform.

There are some programming languages – typically scripting languages – which use very different integer arithmetics. For example, numbers in JavaScript or long integers of Python have a virtually unlimited type-value set. Supporting such arithmetics usually comes with a performance penalty and it is rarely necessary.

Operations

Integer types have operations of three categories. Firstly, the potential special operations inherited from the type class, such as conversion operations, operations to query the type-value set, etc. Secondly, the usual mathematical operations: addition, subtraction, multiplication, division, sometimes exponentiation with positive exponent, etc. And finally the bit-level operations such as left- and right shift, bitwise *and*, *or*, *exclusive or*, and *negation* operations. Bitwise operations are again an indicator of performance awareness. These operations are very rarely used in mathematics, their exact semantics depends on the details of the used representation. This contradicts the usual separation of type specification and implementation where representation is part of implementation. Nevertheless, for certain performance critical applications, we use our intricate knowledge of the representation and make use of these "strange" operations. The processors

[3] In the most commonly used two's complement representation the interval is not symmetric. The type-values set of a k-bit two's complement integer type is the $[-2^{k-1} \dots 2^{k-1} - 1]$ interval.

of modern computers usually have built-in realization of these operations, often they are the fastest operations of the processor.

There are two operations which may produce results out of the type-value set and thus require further attention. They are division and exponentiation with negative exponent. For both of them, the result can be a non-integer real number. Instead of division, most languages provide *integer division* where the result of the division a/b is the integer part of the quotient, e.g. $5/2 = 2$. By contrast, in languages which have an exponentiation operator, the usage of negative exponent is typically not allowed.

The other mathematical operations may have a result outside of the type-value set too, but these results are integer numbers, meaning they just exceed the limitations of the used representation. This is called a numeric *overflow*. Some languages (for example Ada) do check these cases at runtime and generate some runtime error, e.g. an exception. However, most languages simply ignore the overflowing bit, and the outcome of the operation will be just as many of the least significant bits of the calculated results as what fits into the representation. In the case of a k-bit integer type, this corresponds to the rules of modular arithmetics modulo 2^k. In the case of signed integer types – using two's complement representation – the results can, however, be strange. Consider the following two 8-bit signed integer values: 96 and 32, in 8-bit two's complement binary representation 00110000 and 00010000. The sum of these numbers (128, or binary 10000000) exceeds the highest positive element of the type value set, which is 127. If the language does not support runtime range checks, no runtime error will be generated, but the sum of two positive numbers will be negative, -128 which is the interpretation of 10000000 in 8-bit two's complement representation.

Integer types in Pascal

The type set of Pascal is implementation dependent. For example Turbo Pascal has five integer types: the 8-bit signed *ShortInt*, the 8-bit unsigned *Byte*, the 16-bit signed *Integer*, the 16-bit unsigned *Word* and the 32-bit signed *LongInt*, while Freepascal extends this set with five more integer types: the unsigned 32-bit *Cardinal*, the signed 32-bit *Longint*, another unsigned 32-but type called *Longword*, the signed 64-bit *In64* and the unsigned 64-bit *QWord* types. The language is configurable to support runtime range check, an option which can be enabled or disabled using a compilation option. When enabled, the compiler adds extra code to the binary program which generates runtime errors on arithmetic overflows.

The set of operations is the following:

- Addition ($+$), subtraction ($-$) and multiplication ($*$) as usual;
- Two variants of division. The regular division, denoted by $/$, has an unusual result, that is it returns a real value. The **div** operator can be used for integer division. The residue is calculated using the **mod** operator;

- Simple bit arithmetics: left and right shift (**shl**, **shr**), the bitwise negation (**not**), and (**and**), or (**or**) and exclusive or (**xor**) operations;
- Some more complex mathematical operators, such as the absolute value (*Abs*) and square (*Sqr*) calculation.

The `int` type of CLU

CLU has only one integer type called `int`. The type-value set is implementation dependent. The set of operations is mostly the usual, but their notation is different from other languages. In CLU even the basic arithmetic operations are denoted using prefix notation and the English name of the operators. For example the addition $a+b$ in CLU is written as `Int$add(a,b)`. To simplify writing arithmetic expressions, CLU provides syntactic sugar[4] for most operations. The operations are the following:

- Basic arithmetic operations: addition (`add`, `+`), subtraction (`sub`, `-`) multiplication (`mul`, `*`), integer division (`div`, `/`) and residue (`mod`);
- Some more complex mathematical operations: exponentiation with non-negative exponent (`power`), absolute value (`abs`), and maximum (`max`) or minimum (`min`) of two values;
- Relational operators: less than (`lt`, `<`), greater than (`gt`, `>`), less than or equal to (`le`, `<=`), greater than or equal to (`ge`, `>=`) and equality (`equals`, `=`);
- Operators for converting from (`parse`) and to (`unparse`) string;
- Interesting additions are the *iterators*, which can iterate over the elements of an integer interval, potentially with a specified step. The two default iterators of the type are `from_to` and `from_to_by`.

Integer type class in Ada

Ada further divides the integer type class into two parts, *signed integers* and *modulo types*. For the former, the type-value set is an interval of integer numbers , while for the latter, it is the set of residue classes modulo m, where m is defined in the type declaration. The difference between the two types is that in the modulo type the residue classes are represented by the values $0 \dots m-1$ and the operations are cyclic ($m-1+1=0$). In other words, there is no range check. If the modulus (m) is some integer power of 2, then the type corresponds to the unsigned integer types of other languages. The type-value set of signed integer types can contain negative values as well, and there is a

[4] For some operations, in addition to the default prefix form, CLU defines an infix variant as well. This is called *syntactic sugar* because it is just a simplified notation for the other, which the compiler automatically substitutes with the prefix form (for details see Section 5.7.1).

range check in each operation. If the result is not in the specified type-value set, a *CONSTRAINT_ERROR* exception is raised.

```
-- 8-bit unsigned (0..255) value:
type Byte is mod 256;

-- Altitude above sea level (in meter):
Type Altitude is range -15000..9000;
```

Though it seem we can create an arbitrary number of distinct integer types in Ada, in reality all of them are derived from a special *root_integer* type.[5] In other languages, where integer types often have an implementation specific value set, when porting an application to a new platform, we may encounter hard to discover errors caused by changes in the used representation, e.g. in C, the type *int* has a platform specific precision. On 16-bit platforms it is 16 bits, on 32-bit platforms it is 32 bits, etc. When porting an application across such platforms, it changes the value set of the type *int*. In Ada, developers are strongly encouraged to define their own integer types, which makes their application more portable. The application has to explicitly declare what type-value set they expect for the given type. The semantics of the type declaration is not platform specific. Therefore, porting the application cannot change the type-value set.

Nevertheless, there are some built-in integer types in the language which are directly usable. All implementation must support the *Integer* type. Its type-value set must include at least the $(-2^{15}+1)\ldots(2^{15}-1)$ interval. Additionally, the specification allows other built-in integer types such as *Short_Short_Integer*, *Short_Integer*, *Long_Integer*, *Long_Long_Integer*. It does not specify their exact value set, it only mandates that the relationship of their respective type value sets must correspond to what their names imply. Additionally it mandates that the value set of the *Long_Integer* type must include at least the $(-2^{31}+1)\ldots(2^{31}-1)$ interval.

Apart from the above mentioned differences in the semantics of the arithmetic operators, the set of available operations is identical in both integer type classes. As integer types are discrete types, and therefore, they are scalar types, all operations defined in those classes are available. Additionally, integer types have the following operations:

- Usual basic arithmetic operations: addition ($+$), subtraction ($-$), multiplication ($*$) and division ($/$);
- Two different operators for calculating residue: Firstly, the operator **mod**, for which $a = b \cdot n + (a \,\mathbf{mod}\, b)$ for some signed integer n and the sign of $(a \,\mathbf{mod}\, b)$ is the sign of b. Secondly the operator **rem**, for which $a = (a/b) \cdot b + (a \,\mathbf{rem}\, b)$ and the sign of $(a \,\mathbf{rem}\, b)$ is the sign of a (see Table 5.1 for examples);

[5] The *root_integer* type is special because it cannot be used in variable declarations. This type is only used for deriving integer types.

- Absolute value (**abs**) and exponentiation (**) with non-negative exponent.

A	B	A/B	$(A \textbf{ mod } B)$	$(A \textbf{ rem } B)$
5	2	2	1	1
5	-2	-2	-1	1
-5	2	-2	1	-1
-5	-2	2	-1	-1

Table 5.1: Difference between the different residue operations of Ada integer types

Brief overview of some other languages

C and C++

In C and C++ the signed integer types are the 8-bit **char**, the 16-bit **short** the 32-bit **long** and the platform specific **int**. On 64-bit architectures an additional 64-bit integer type **long long** has also been introduced. Additionally, from each singed type the corresponding unsigned type can be created using the **unsigned** modifier. The operations are the usual ones, there is no overflow check.

Java

As Java applications are executed in the standard virtual machine, the Java language can very accurately specify the required representation of the various integer types. The integer types **byte**, **short**, **int**, **long** use signed 8, 16, 32 and 64 bit representations respectively. Additionally, there is an unsigned 16-bit **char** type as well. The set of operations is the usual, there is no overflow check at runtime.

Eiffel

Eiffel has adopted a rather unique solution by integrating scalar types, such as integers into the class hierarchy of the language. The class *INTEGER* is an expanded class, its objects are stored by value, which ensures efficiency. The class extends *COMPARABLE* and *NUMERIC* classes. From the former it has inherited relational operations, while from the latter it has inherited the usual arithmetic operations. Bit-wise operations are not supported directly on the *INTEGER* objects. However, there are operations for converting *INTEGER*s to bit sequences and vice versa. This approach gives a nice resolution for the previously mentioned contradiction between the need for efficiency and hiding representation.

Python

Python supports two integer types as part of the language. The representation of type `plain integer` is implementation dependent two's complement representation as supported by the CPU, but it has at least 32-bit precision. Its operations are the usual ones. However, the language has a more interesting second integer type called `long integer`. This type has a virtually unlimited type-value set, the representable maximum number is only bound by the memory limits. Such big number types are usually not built into other languages. They might have an implementation in some standard library like in the case of *BigNumber* class of Java .

PL/I

In PL/I, arithmetic types are specified along multiple dimensions. They have a *base representation*, which can be `BINARY` or `DECIMAL`, a *scale*, which can be `FIXED` or `FLOAT`, a *mode* of `REAL` or `COMPLEX` and a `PRECISION` which is the number of representable *digits* and a *scale factor* – number of decimal places – for fixed point types. Therefore, integer types of PL/I are `FIXED` types of `BINARY` or `DECIMAL` representation with a scale factor of 0. `BINARY` types use two's complement representation, while `DECIMAL` types are represented by BCD – Binary Coded Decimal – representation. In BCD representation the decimal digits of a number are stored as hexadecimal digits, e.g. the number 12345 is represented as $0x012345$ in BCD. BCD representation is hardly used at all for integer values. With regards to memory utilization, it is less economic as 4 bits (which can represent 16 different values) are used to represent a decimal digit (which has 10 different values), arithmetic operations are less efficient too. However, this representation is more accurate when used for representing real values. For example, the decimal value 0.2 is a repeating fraction of $0.00110011\ldots$ in binary form. As data representation is finite, there is always some loss of precision.

5.4.3 Outliers

There are two types in the discrete type class which do not have a fixed position in the provided taxonomy. These are *boolean* and *character* types, which are sometimes treated as enumerations, and sometimes as integers in the different languages.

Character type

Computers can work with numbers only. When computers need to deal with characters (text), they have to be represented as numbers. The mapping between numbers and characters is called the *character encoding*.

There may be more than one character type in a language. Most modern languages have built-in support for some character type which is compatible

with character sets that have more than 256 characters – e.g. the Universal Character Set, developed with Unicode (see Section 2.1.4 for details) – and support some 16+ bit character encoding, whereas they often retain support for the more traditional 8 bit character set as well. Such languages have two or more different character types.

The distinction between languages which treat character types as integer types and the ones that treat them as enumerations is very important. The languages where the character type is an integer type are agnostic to character encoding. The programs in that language do not manipulate text or characters but sequences of numbers which are later translated to pieces of text. The programs in these languages directly work on the representation of the character type, it is not hidden from them. Therefore, *the semantics of the programs are character encoding specific*. In languages where character types are enumerations, the type-values represent actual characters, the programs manipulate sequences of characters, meaning their semantics does not depend on the character encoding, which is completely hidden from them.

In C and C++ the character type is implemented by the integer type **char** which uses 8-bit two's complement representation. Its operations are the usual integer operations of the language. The languages do not specify the character encoding, the interpretation of the character values is application specific. The language hardly provides means for character manipulation as it would be representation dependent.

Java is an interesting hybrid between the two approaches. Characters belong to the integer types, **char** is 16-bit unsigned integer. However the language specifies that the type values correspond to the characters of Unicode/UCS Basic Multilingual Plane (BMP). The operations of the type are the same as for the other primitive integer types and the data representation is not hidden, but the interpretation of data is specified by the language. The wrapper class *Character* provides a number of useful operations for character manipulations.

In Pascal-type languages such as Ada, characters are enumerations. Operations are the usual for that type class. Ada supports two character types. The type *Character* has 256 elements, the type-values are the characters of ISO 10646 Basic Multilingual Plane (BMP) Row 00 (Latin-1), their ordering in the type corresponds to the characters order in Row 00. Similarly the elements of type *Wide_Character* correspond to the 65536 code points of ISO 10646 BMP. In this language the type-values represent the given characters, the underlying representation is completely hidden from the applications which operate on (sequences of) actual characters.

Boolean types

The other outlier type is the boolean type. Many languages (for example C) do not have such type at all but they use some other type – typically integer – for

representing boolean values. In these languages 0 represents the value *false* and non-zero values represent *true*.

In Delphi there are multiple boolean types, but in reality all of them are integer types. The type *Boolean* is represented on one byte, its permitted values are 0 and 1 which correspond to *false* and *true* values respectively. Additionally, the language defines the *Bytebool*, *Wordbool* and *Longbool* types which are 8, 16 and 32-bit integer types where the value zero represents false and the non-zero values represent true.

In other languages the boolean type belongs to enumerations. Its elements are *false* and *true*, typically in this order, i.e. false is less than true. Such languages include Pascal, Ada, CLU or Eiffel.

Be it an enumeration or an integer type, the set of operations of boolean types is wider than that of other members of its type class. Boolean types in each language support a set of boolean operations. Typical operations are the boolean *and* ($\wedge$), *or* ($\vee$), *exclusive or* ($\oplus$) and *negation* ($\neg$). These operators should not be mistaken with the corresponding bit-wise operators of integer types, especially in those languages where boolean types belong to the integer type class, or where there is no boolean type at all. In these languages both kinds of operators can be applied to objects representing boolean values, but the results of these operations can be very different.

When evaluating boolean expressions, often it is not necessary to evaluate both operands of a binary operator – the result of the expression can be determined by evaluating only one of them. For example the expression $A \wedge B$ is false, independently of B if A is false. Similarly $A \vee B$ is true if A is true, independently of B. For reasons of efficiency most programming languages do not evaluate the operand B if the value of the expression can be determined after evaluating A. If A and B are simple boolean values or variables, this *lazy* evaluation technique is beneficial. However, when evaluating B has some side effects, evaluating or not evaluating B makes a difference. For this reason, some programming languages have two variants of boolean operators *and* and *or*. The *lazy* versions of these operators do not evaluate the second operand if the value of the expressions is known after evaluating the first one. The *greedy* variants always evaluate both operands thereby ensuring that all side effects of the second evaluation take place as intended. Such lazy operators in Ada and Eiffel are called **and then** and **or else**, while in CLU they are `cand` and `cor`. The greedy variants in Ada and Eiffel are **and** and **or**, in CLU they are & and |.

Additionally, Eiffel has one more boolean operator for *logical implication* ($\rightarrow$). The expression $A \rightarrow B$ is false if A is true and B is false. Otherwise it is true. The implication operator is called **implies**.

5.5 Real type class

The real type class contains different realizations of real numbers. The cardinality of real numbers exceeds the cardinality of integers. Therefore, realization of real numbers is even more difficult. The most important difference between the types in real type class is the used data representation.

5.5.1 Type-value set

During the history of computer science many different solutions have been created for representing real numbers. Two major directions are fixed point representation and floating point representation. Neither of these are clearly superior to the other, that is both approaches have advantages and disadvantages. Therefore, though not with equal importance, both approaches are in use.

Fixed point representation

One of the simplest mapping of real values to integers is rounding. This is the intuition behind *fixed point* number representation, where real values are represented on a fixed length. This length is divided to a fixed length of integer part and a fraction. The advantage of this representation is that its precision is clearly determined. When developing some financial application, legal requirements may mandate a certain precision, for example that each value must be stored up to two-digits. Such requirements can be directly translated to fixed point representation.

However, this representation has a big disadvantage as well. It does not scale to numbers of different magnitude. If we need to represent big and small values at the same time, we cannot apply it. If many digits are used for the fraction, it is good for small numbers but in cases like that we do not have room for a big integer part. If the integer part is big, we can represent big values, but we do not have enough digits in the fraction part to represent small values. Additionally, the result of computations must also fall into the type-value set defined by the chosen precision. Consider, for example, that we try to compute the function $x \to 1/x$. If the chosen precision allows for large value of x, most likely it does not have enough digits in the fraction part to represent the small value of $1/x$ and vice versa.

These limitations make the use of fixed point representation in scientific and engineering applications almost impossible. Besides this, manual implementation of fixed point types is relatively easy. Consequently, most modern programming languages do not have a built-in fixed point real type. Whereas languages which have special support for financial applications do offer such a type.

Floating point representation

Floating point representation was created to overcome the limitations of fixed point representation. It is a computer realization of the *standard form*[6] of numbers. In standard form numbers are written in the form of $m \cdot 10^k$ where $1 \leq |m| < 10$ and k is an integer value.[7] The value m is called *significant digits* or *mantissa* of the number while k is the *exponent*. In floating point representation a binary version of the standard form is used: $m \cdot 2^k$ where $1 \leq |m| < 2$ with special encoding rules for the value 0.

Most modern programming languages support floating point representation of real values. Most generic purpose CPUs currently have built-in support for floating point arithmetics. The corresponding standard of IEEE (IEEE 754 – *IEEE Standard for Binary Floating-point Arithmetic*) was accepted in 1985, which unified the different floating point implementations that were in use previously. In addition to representation, the standard specifies the exact semantics of all operations and transformations. It defined five basic formats and some more so called extended formats. The basic formats are *half precision* (16 bit), *single precision* (32 bit), *double precision* (64 bit), *double extended* (80 bit) and *quadruple precision* (128 bit) representations. These formats differ only in the number of bits used to represent the different parts (mantissa, exponent) of the numbers. The two most typically supported formats are single and double precision floating points.

Single precision

Single precision floating point numbers are stored on 32 bits. The first bit denotes the sign (s) followed by the 8-bit exponent (k) and the 23-bit mantissa. As $1 \leq |m| < 2$, the first bit of m would always be 1, this bit is not stored, it is called the *hidden* or *implicit bit*. This way the precision of single precision floating point representation is 24 bits. Some bit sequences have special meaning:

- If $k = 255$ and $m \neq 0$, then the value is *not a number* (NaN).
  For example: 0 11111111 01000100010001000100010 $\rightarrow NaN$.
- If $s = 1$, $k = 255$ and $m = 0$, then the value is *negative infinity*.
  That is: 1 11111111 00000000000000000000000 $\rightarrow -\infty$.
- If $s = 0$, $k = 255$ and $m = 0$, then the value is *positive infinity*.
  Therefore: 0 11111111 00000000000000000000000 $\rightarrow +\infty$.
- If $0 < k < 255$, then the value is $(-1)^s \cdot 2^{k-127} \cdot (1.m)$, where $1.m$ denotes the binary number with integer part 1, and fraction part m.
  For example: 0 00000000 10000000000000000000000 $\rightarrow 2$
  or 1 10000001 10100000000000000000000 $\rightarrow -6.5$.

[6] Often called *scientific notation* as well.
[7] The value 0 cannot be represented in standard form.

- If $k = 0$ and $m \neq 0$, then the represented value is $(-1)^s \cdot s^{-126} \cdot (0.m)$. These are *non-normalized* values. For example:
  0 00000000 10000000000000000000000 $\rightarrow 2^{-127}$
  or 0 00000000 00000000000000000000001 $\rightarrow 2^{-149}$, which is the smallest possible value representable in this format.
- If $k = 0$ and $m = 0$, then the represented number – depending on the value of s – is $+0$ or -0.
  0 00000000 00000000000000000000000 $\rightarrow +0$
  and 1 00000000 00000000000000000000000 $\rightarrow -0$.

Double precision

The representation of double precision numbers is similar, but 64 bit are used. 1 bit is the sign, 11 bits represent the exponent and 52 bit is used for the mantissa. Similarly to single precision, there is a hidden bit which makes the precision 53 bits.

5.5.2 Operations

IEEE 754 also standardizes the set of operations for floating point types with their exact semantics, which includes the result of operations when at least one of the operands is a special value such as NaN, positive or negative infinity. The operations include the basic arithmetic operations $(+, -, *, /)$, exponentiation, square root, and different conversion operations. Due to standardization most modern general purpose processors have built-in support for these which makes their implementation very efficient.

Additionally, most modern languages offer some mathematical functions as part of standard libraries. These typically include logarithm and exponential functions, trigonometric functions and simple random number generators.

5.5.3 Programming languages

In this sections we will have a look at some concrete implementations and their unique features in various programming languages.

Pascal

Real types in Pascal use floating point representation. The type *Real* is 48-bit long with 1 bit sign, 8 bit exponent and 39 bit mantissa. Depending on implementation, other real types might be supported. For example in Turbo Pascal there are three other real types – *Single*, *Double* and *Extended* –, which correspond to single precision, double precision and double extended representations specified in IEEE 754.

Pascal allows the mixed use of integer and real types in expressions, but in these cases the result is always real. The operations of real types are:

- Arithmetic operations: addition ($+$), subtraction ($-$), multiplication ($*$) and division ($/$);
- Absolute value (*Abs*) and integer part (*Int*), the result of this operation is real;
- Square (*Sqr*) and square root (*Sqrt*);
- Natural based exponential (*Exp*) and logarithm (*Ln*) functions;
- Trigonometric functions: sine (*Sin*), cosine (*Cos*), etc.;
- Random number generator (*Random*);
- Conversion operators: truncate fraction (*Trunc*) and rounding (*Round*).

Real types in Ada

Ada supports both fixed and floating point representations. Similarly to integer types, we can create new real types. In the type declaration we have to specify the precision of the created type. Logically, all real types are derived from the abstract *root_real* type. All literals belong to the class of *root_real*, they are of type *universal_real*. This enables using the same set of literals for both floating and fixed point types. The types *root_real* and *universal_real* cannot be used in type declarations.

Fixed point types

In the declaration of fixed point types the absolute precision of the type is specified. If we want to represent numbers with k digit precision, the 10^{-k} value is called the *delta* of the type. As the language uses binary representation, the values of the type will not be the form of $n \cdot 10^{-k}$ but $m \cdot 2^{-p}$, where p is the smallest positive integer such that $2^{-p} \leq 10^{-k}$. The value 2^{-p} is called the *small value* of the type as this is the smallest representable value. This difference between the delta and the small value of the type can cause inaccuracies, e.g. in the example below the value of Y is different from what might be expected:

```
type Fixed is delta 0.01 range 0.00 .. 1.00;

X: Fixed := 0.05;      -- in reality X is 0.046
Y: Fixed := 5.0 * X; -- Y = 0.23
```

Floating point types

In the declaration of floating point types, first we need to specify the *relative precision* of the type, and then the number of *decimal digits* of the mantissa. For a floating point type with d decimal digit precision we need at least $b = \lceil d \cdot \ln(10)/\ln(2) + 1 \rceil$ bit long mantissa in binary representation. In this case the exponent is in the $-4 * b \ldots 4 * b$ range.

The precision of the built-in *Float* type is implementation dependent, but it should be at least 6 decimal digits. Similarly to integer types, implementations may support other built-in types (*Short_Short_Float*, *Short_Float*, *Long_Float*, *Long_Long_Float*) but the only requirement in the language is that their precision should be consistent with the naming of the types.

The set of operations for fixed and floating point types is similar. It includes basic arithmentic operations ($+$, $-$, $*$, $/$), exponentiation with integer exponent (**) and absolute value (**abs**). The difference between fixed and floating point types is attributes which can be used to obtain information about the precision of the types.

For a floating point type T the attribute T'**Digits** specifies the precision of the type in number of decimal digits.

Fixed point types introduce four new attributes. For a fixed point type S, the attributes S'**Delta** and *S'Small* are the delta and small value of the type. *S'Aft* is the number of decimal digits – digits *after* the decimal point – while *S'Fore* is the maximum number of digits in the integer part – i.e. *before* the decimal point – including the potential sign.

5.6 Pointer types

Pointer and reference types are the abstractions of addresses in the memory of the computer. there is no clear distinction between the two notions, but generally references are a safer variant of pointers. This safety means that a reference is guaranteed to point to an existing object, or if it is not, it has a special ”nowhere” value, which is clearly recognizable. Beyond this difference, the pointer and reference types show many similarities, which we will discuss together (the term pointer will be used to denote both). The term reference will be used for highlighting the occasional differences.

An important concept for understanding the usage of pointers is *reference level*. If we have an object, for example the number 42, its reference level is 0. The reference level of a variable v, which stores this object is 1. The reference level of a pointer that points to v is 2 etc. When a variable is used in an expression, its reference level is reduced by one, e.g. the expression *a := b + 1* means that the variable *a* should be assigned the *value stored in b* plus 1. The compiler automatically *dereferences* the variable *b*.

There is a wide spectrum of application of pointers. Often we do not even notice that we are using them. For instance, in many languages when we need to pass large and complex data structures as parameters to subprograms, the compiler does not copy the whole structure but simply passes a pointer to the structure as argument. Similarly, in object oriented languages to realize polymorphism and dynamic binding we have to access the objects via pointers (for details see Chapter 10). And often we apply pointers explicitly, for example when building dynamic data structures, or when we want to share access to an object.

5.6.1 Memory management

Before we elaborate on pointers, let us examine the memory management of a program. Based on their *lifetime* the variables of a program fall into three categories:

- *Static variables* are created upon the first evaluation of their declaration and their lifetimes span for the complete runtime of the application. Not all languages support the creation of static variables. These variables are often global variables of the application, though in some languages – for example C and C++ – subprograms can also have static variables, which preserve their value between invocations of the subprogram.
- *Automatic variables* are automatically created and destroyed during the runtime of the application. In languages which support block structures, the variables declared within blocks are such automatic variables. Every time the execution of the program enters a block, the variables declared there get created. As soon as the execution of the block is finished, these variables are destroyed. Therefore, they do not preserve their values between distinct executions of the block. Many modern languages allow subprograms to invoke themselves (*recursion*). In such cases multiple instances of the same variable can exist at the same time.
- The lifetime of *dynamic variables* is managed – at least partly – by the programmer. Dynamic variables are access via pointers. While in the case of static and automatic variables the compiler or runtime environment is responsible for allocating and releasing the required memory, dynamic variables are created by the developer using some *allocator*. The programmer is also responsible, to some extent, for destroying these variables when they are not needed anymore.

According to this categorization of variables, the memory used by the program is divided into three parts:

- The *static memory* is used to store the static variables. The size of this memory is determined at compilation time and the address of the variables residing here do not change during the execution.

- The *call stack* is the storage of automatic variables. When a block is entered during execution, automatic variables of the block are allocated at the top of the call stack. This solution enables having multiple instances of the same variable during recursive execution. When the execution of the block is finished, the automatic variables are removed from the stack.
- The third part of memory is the *dynamic storage area*. During program execution free memory blocks are stored here in a structure called *free list* or simply the *heap*. When the developer tries to create a new dynamic variable, the *allocator* selects a free memory block big enough to store the variable, removes it from the free list and returns its address to the developer. When the dynamic variable is destroyed, the memory block occupied by it is returned to the free list.

The lifetime of dynamic variables is controlled by the application developer. They are created using an allocator, which can obtain the required amount of free memory from the heap, and which can – in some languages – initialize the obtained memory block. Destroying dynamic variables can be explicit or implicit.

Explicit deallocators

In some languages the developer is fully responsible for destroying dynamic variables when they are not needed anymore. These languages provide some *explicit deallocator* operation. The developer is responsible for using this operation to free up the memory block occupied by a dynamic variable. Along similar lines, in some languages the deallocator is responsible for cleaning up the memory area before releasing it to the heap, e.g. in C++ it executes a designated method of the corresponding class called the *destructor*.

In these languages the developer has fine control over the memory usage. However, such freedom comes with big responsibility. If the developer forgets about releasing unused dynamic variables, the program can fill up the available memory, which is called *memory leak*. On the other hand, the developer also has to make sure that that the program does not attempt to access a memory block once it has been released. Trying to access dynamic variables which have once been destroyed may lead to runtime errors which are very difficult to find.

Garbage collection

In other languages, deallocation of unused memory blocks is not the developers' responsibility – it is handled by the runtime environment. In such languages for each dynamic variable a *reference counter* is maintained which indicates whether the corresponding variable is accessible from somewhere within the application. When a variable becomes inaccessible, it becomes eligible for *garbage collection*. The garbage collector is a mechanism in the runtime environment which is responsible for finding memory blocks that can be released and for

deallocating them according to the rules of the language. Depending on the runtime implementation, garbage collection can be triggered by a timer, a certain level of memory usage or the combination of both. Some languages allow the developer to explicitly trigger garbage collection as well.

The pointers in languages which support garbage collection are often called references. Their usage is safer, meaning it cannot happen that the developer forgets to release an unused variable or that he refers to a variable which has already been released. However, the developer has a lot looser control over the memory usage of his application. Another potential drawback of this approach is that garbage collection generates periodic peaks in the CPU usage of the application – it might even have to stop all other execution threads, which might have undesirable effect on the real time characteristics of the application. Garbage collection is a complex task, as it is not enough to release variables with reference counter 0, there might be circular references between variables. The garbage collection must be able to discover such blocks which have no external references anymore and release all objects in them simultaneously. Nevertheless the added safety and ease of use is a big motivation for using garbage collectors. The popularity of Java and C# contributed a lot to understanding the challenges of garbage collection and to the creation of better and better garbage collectors.

Memory management has fundamental influence on the usage of pointers and their available operations. Further details of memory management are provided in Chapter 4.

5.6.2 Type-value set

A pointer is an object which represents the location of another object in memory. The actual representation is implementation dependent but often it is an unsigned integer value which is the index (*address*) of the first byte of the referenced object in memory. Other representations are also possible. Disconnecting pointers from actual memory addresses might enable relocating objects at runtime for instance to enable defragmenting the memory.

Untyped pointers

In most languages pointers are usually *typed*, meaning they can refer to certain types of objects. Usually this typedness does not change the representation of the pointer, but it enables the compiler to verify the correct usage of the referenced objects. However, in some languages it is possible to define untyped pointers as well. Untyped pointers represent a memory address without restricting the type of object located at that position.

The usage of untyped pointers is rather limited. As the type of the referenced object is not determined, dereference is not permitted on them. Usually any pointer can be converted to untyped pointer automatically, while the opposite direction requires explicit – interpretation changing – conversion.

A typical use case for untyped pointers in languages which do not have a common superclass for all objects is to create containers which can store *any* kind of object. The following – C language – example is the implementation of a list type which can contain arbitrary objects.

```
struct __listelement_struct;
typedef struct __listelement_struct* List;
struct __listelement_struct {
    List     next;
    void* element;
};
List insert(List l, void* element) {
    List p = malloc(sizeof(struct __listelement_struct));
    if (p != NULL) {
        p->next    = l;
        p->element = element;
    }
    return p;
}
...
List  l = NULL;
int   i = 5;
char  text[] = "Hello world!";
l = insert(l, &i);
l = insert(l, &text);
```

As illustrated above, any kind of object can be inserted to the list. But in order to use the objects after having retrieved them from the list, we need to know their exact type. In C the **void*** pointers can be converted to any pointer type, but the developer needs some solution to determine the type of the retrieved objects to be able to use them. The language itself does not offer such mechanism.

In object-oriented languages where the language defines a common base class for all classes – for example in Java or Eiffel –, references of the common base class can serve as untyped pointers. Though not exactly untyped – its type is the common base class –, these references share the ability to point to any object in the language. However, a significant advantage is that objects of these languages typically carry information about their dynamic type, which enables us to safely perform the conversion when using such an "untyped" reference. Before introducing generics to Java, the collection framework of the language used *Object* references for the stored objects.

Pointers to nowhere

In all programming languages the type-value set of pointers contains a special value, the *pointer to nowhere*. This pointer is called *NULL* in C, *0* in C++, **null** in Ada and Java, `nil` in CLU, *void* in Eiffel, etc. It has many names, but its properties and function are the same. This pointer can be automatically converted to any pointer type; therefore, it can be assigned to any pointer variable. The value is almost always represented as the constant 0 bit sequence, which is not a valid address in any system. Dereferencing the pointer to nowhere always causes runtime error.

In many languages, especially the ones which provide references instead of pointers – e.g. Ada, Java, Eiffel, CLU – the default value of references is the pointer to nowhere. This means that uninitialized references can be clearly distinguished. In C or C++ uninitialized pointers have undefined values.

If a language supports garbage collection and its pointers or references by default are initialized to the pointer to nowhere, the pointers have a very useful invariant property. Their value is either the pointer to nowhere or they point to an existing object. This invariant greatly simplifies the development of sound applications.

Forward declaration

A very important property of pointer types is that their representation is independent from the referenced data type. We rely on this property to resolve the "chicken and egg" problem of creating linked data structures. In the example below – written in Ada – we describe the representation of linked list type. The *ListElement* type consists of an integer value – the data – and a reference pointing to the next element of the list. When declaring the type, we ran into a problem immediately. That is, in Ada reference type can only be declared for an existing type but components of a record type must also be of existing types. The solution to this contradiction is the *forward declaration* of the *ListElement* type. With forward declaration we let the compiler know that the type *ListElement* exists. As the representation of references does not depend on the details of the referenced type, the existence is enough to define the *ListElementAccess* type, which is a fully defined type at this point. Therefore, it can be used in the definition of *ListElement* as the type of one of its components.

```
-- Forward declaration of ListElement type
type ListElement;

-- We can now define the referece type to it
type ListElementAccess is access all ListElement;

-- And now we can define the ListElement type as well
type ListElement is record
    Data : Integer;
    Next : ListElementAccess;
end record;
```

5.6.3 Operations

Moving on to the operations of pointer types, some of the operations – e.g. assignment, or allocation – are present in all languages, others are very unique, language specific operations – e.g. the pointer arithmetics of C and C++.

Assignment

Assignment of pointers usually means copying the address of the referenced memory block. However, in languages which support garbage collection, such as Ada or Java, assignment includes the maintenance of reference counters as well. In the assignment $p := q$, if both pointer variables reference to existing objects, the reference counter of the object referenced by p needs to be reduced by one – p does not reference it anymore –, if its value drops to 0 it might also trigger garbage collection, depending on the implementation. At the same time the reference counter of the object referenced by q needs to be incremented by one as it is now referenced by p a well.

Generally, there are five ways of assigning a new value to a pointer variable:

- The pointer to nowhere is assigned to them. In many languages this is the default value of all pointers.
- The value of another pointer variable is assigned to them.
- The pointer to a newly allocated object is assigned to it using an allocator.
- The address of a static or automatic variable is assigned to it. However, not all languages support this option as it can be unsafe, especially in the case of automatic variables.
- Some languages allow assigning concrete memory addresses to pointers. For instance in C any integer value can be converted to a pointer.

Allocators

All languages that support dynamic variables must provide an *allocator* operation. This operation is responsible for obtaining a free memory block of sufficient size from the free list and returning its address. In many languages – in object-oriented languages such as Java and C++ in particular – the allocator is responsible for initializing the allocated memory block according to the type of the object to be stored there.

Deallocators

Not all languages have an operator for *deallocation*. Languages that use garbage collection typically do not provide explicit deallocators. However, some of these languages might provide some means for the developers to control deallocation.

It is quite common to provide a mechanism to explicitly trigger garbage collections, e.g. in Java we can use the *System.gc()* method.

Since the launching of Ada 95, Ada offers even deeper access. The developer can declare that he is taking full responsibility for maintaining the consistency of memory management related to a certain type by instantiating the *Ada.Unchecked_Deallocation* generic for that type. The instance of this generic is a deallocator usable for objects of the given type. It is important to note that the type is removed from the scope of garbage collection.

Referencing

Many languages allow assigning the address of a static or automatic variable to a pointer variable. The address of the variable is created using the *reference operator* of the language. Not all languages provide such mechanism, e.g. Ada 83, Eiffel or Java do not have such means. In these languages pointers can only reference dynamic variables.

In C and C++ the reference operator is *$&$*. It can be used without limitations on any objects.

The Ada 83 version of Ada did not allow the referencing of static and automatic objects, but this restriction was loosened in the Ada 95 revision. As the language is essentially garbage collection based, the references created using the reference operator also had to be integrated into this system for safety.

The challenge of introducing a reference operator is that the lifetime of automatic variables depends on the block structure of the language and the thread of execution. When the thread enters a block, the automatic variables of the block are created, and when it exits, the variables are destroyed.

Assume that we have a program which consists of blocks A and B, where B is embedded in A. In block A we declare a pointer IP and in block B we have an automatic variable I. We also have a statement in block B which assigns the

reference of *I* to *IP* using the reference operator. The following code snippet results in compilation error:

```
Type IntegerAccess is access all Integer;
IP : IntegerAccess;

Procedure B is
I : aliased Integer;
Begin
    -- IP is visible here as it is global to this block
    -- However, this assignment is invalid...
    IP := I'Access;
End B;

Begin
   B;
   -- ... because at this point IP would point to invalid memory area
   IP.all := 42;
End;
```

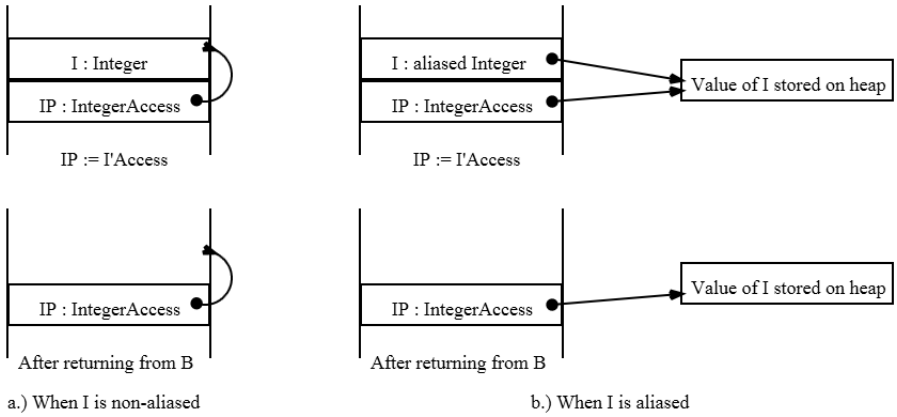


Figure 5.2: Difference between aliased and non-aliased variables in Ada

As shown in part a.) of Figure 5.2, when the execution exits *B*, the variable *I* is destroyed. However, the pointer *IP* still points to the same memory area. Destroying *I* would break the former invariant of references which guarantees, that they either point to a valid memory area or their value is **null**.

Therefore, Ada does not allow the use of the reference operator on *ordinary* automatic variables.

The solution of this problem in Ada 95 is that the developer needs to mark automatic variables with the keyword **aliased** in case they want to use referencing of that variable. The compiler will place these variables – even though they are

automatic – in the dynamic storage area and it will replace them with a pointer within the block. In other words, it converts these variables to dynamic ones, and creates an automatic variable which holds the only reference to them. If the referencing operator is not used, the automatic (reference) variable is destroyed which will drop the reference counter of the dynamic variable to 0, which will then be garbage collected. If the reference of the variable is assigned to some other pointer, the variable can live safely after leaving the block as it resides in the dynamic storage. See part b. of Figure 5.2. Below is the corrected example of using the reference operator (the '**Access** attribute) in Ada 95:

```
Type IntegerAccess is access all Integer;
IP : IntegerAccess;

Procedure B is
I : aliased Integer; -- Allocated I on heap, variable I is a reference.
Begin
    IP := I'Access;
End B;

Begin
   B;
   -- This is now valid, IP points to an address in the heap
   IP.all := 42;
End;
```

5.6.4 Dereference

One, if not the most important operation of pointers is *dereference*. Dereference operation reduces the reference level by one, i.e. it produces the referenced object based on its address.

In most languages dereference needs to be explicitly marked in code. For example in C and C++ the operator of dereference is the prefix `*`. If p is a pointer, `*p` denotes the referenced object. If p points to a structure (**struct**), or in the case of C++, it point to a class, which has a field f, the `(*p).f` reference to f can be shortened as `p->f`.

In object-oriented languages where objects are accessed through references – for example in Java, CLU or partly Eiffel – there is no need to explicitly mark dereference. The language defines for each operation whether they work on the reference or the referenced object. For example, if a and b are two object references in Java, the `a = b` assignment or the `a == b` equality operator are operations on the references. However, method invocation is always an operation on the referenced object; therefore, `a.equals(b)` will compare the objects and not their references.

Ada combines the two approaches. When it cannot be decided unambiguously on the basis of the context if the operation should be applied to the reference or the referenced object, dereference needs to be explicitly indicated. Dereference is indicated using the **.all** qualifier. Let us assume that *I* is an integer variable, *IP* and *JP* are two integer access – reference – variables. In the case of the *I:=IP* and *IP:=I* assignments or the *I = IP* equality check, the indication of dereference is not needed. In each case it follows from the context that the referenced object should be used, otherwise the operations would not be defined.

However, if we replace *I* with *JP* in the expressions, they will become ambiguous. The operations are defined at both reference levels – on references as well as on the referenced objects; therefore, we have to explicitly indicate dereference. The assignment *IP:=JP* is interpreted at the reference level; if we want to assign the referenced object, we have to use the *IP*.**all** := *JP*.**all** form.

Equality

Almost all languages provide means to check the equality of pointers. This equality is always interpreted at the reference level, meaning it is independent from the equality operation of the referenced objects, irrespective of whether they have such operations. Two pointers are equal if they point to exactly the same object, i.e. they contain the same memory address.

5.6.5 Pointers to subprograms

A special variants of pointers are the *pointers to subprograms*. The type-value sets of these types consist of the entry points of subprograms, rather than of the memory addresses of objects. The pointer to nowhere is element of these type-value sets as well.

Pointers to subprograms are always typed. The type of the referenced subprogram is the signature of the subprogram, which is the list of formal parameters and in case of functions the type of the returned value.

The set of operations for subprogram pointers is a lot more limited than for pointers of objects. There are no allocators or deallocators as subprograms cannot be dynamically created. Assignment and equality check is similar to object pointers. There are reference and dereference operators. The later in practice means the invocation of the subprogram.

C – and therefore, C++ as well – support subprogram pointers. The example below is a function which calculates the definite integral of a real function over a specified interval. The first parameter of the function is the integrand real function, which is followed by the end points of the interval and the number of integration points. Notice that neither reference, nor dereference need to be indicated explicitly, though both are allowed.

```
typedef double (*RealFunction)(double);
double Integral(RealFunction fn, double a, double b, int n) {
    double x     = a;
    double delta = (b - a) / (double)(n);
    double s     = 0.0;

    if (b <= a) return 0.0;
    while ((x + delta) <= b) {
        s += delta * (fn(x) + fn(x + delta)) / 2.0;
        x += delta;
    }
    return s;
}
double sininteg = Integral(sin, 0.0, 1.0, 1000);
```

In object-oriented languages, the importance of subprogram pointers is smaller. Subprogram pointer types can be substituted with an interface or abstract class which declares a single method, the subprogram to be passed. Then objects that implement the specified interface or abstract class can be created, which realize the required variant of the method. These objects typically do not have data members at all. They are called *function objects* or *functors*. Below is a Java implementation of the definite integral calculation. Notice the usage of an unnamed embedded class as functor.

```
public interface RealFunction {
   public double calc(double x);
}
public static double integral(RealFunction fn,double a,double b,int n) {
    double x     = a;
    double delta = (b - a) / (double)(n);
    double s     = 0.0;

    if (b <= a) return 0.0;
    while ((x + delta) <= b) {
        s += delta * (fn.calc(x) + fn.calc(x + delta)) / 2.0;
        x += delta;
    }
    return s;
}
...
double sininteg = integral( new RealFunction() {
   public double calc( double x ) {
      return Math.sin(x);
   }
}, 0.0, 1.0, 1000);
```

5.6.6 Language specialties

In this section we will examine some special, language specific solutions related to pointers.

Pointer arithmetics of C

One of the most interesting and probably most often used property of C is pointer arithmetics. By using pointer arithmetics we can use pointers very conveniently and in C, we may create close connection between arrays and pointers. Pointers in C are unsigned integers. Their value is the memory address – index – of the first byte of the referenced object.

C supports typed pointers. The type of pointers to a type T is $T*$. It has untyped pointers as well. The type of untyped pointers is **void***, which behaves as if it were a pointer to byte type (**unsigned char**) but dereference is not allowed for this type.

The pointer arithmetics of C extends the usual set of pointer operations in the following respects:

- The operator **sizeof** specifies the number of bytes used for the representation of a type T.
- Any pointer type can be converted automatically to **void***.
- The pointers of type **void*** can automatically be converted to any pointer type.
- Pointers can be converted to integers (**int** or **unsigned int**). This is an interpretation changing conversion. The conversion must be indicated explicitly.
- Similarly, integers can be converted to pointers of any type.
- Integers can be added to pointers. Let us assume that $T*\ p$ is a pointer and i is an integer. The type of the expression $p+i$ is $T*$. The addition modifies the numeric value of the pointer p with $i*$**sizeof**`(`T`)`. By using this property, we can iterate over the elements of an array by setting a pointer to the first element, and by always incrementing the pointer by 1 (`p++`), which will then point to the next element of the array. If the added integer is negative, the pointer will move in the opposite direction.
- Pointers of the same type can be subtracted from each other. The result is an integer, which, when added to the first one, gives the second pointer.
- Pointers can be indexed like arrays. Arrays and pointers in C are closely related. Arrays behave like pointers to the first element of the array. If t is an array, the reference to the ith element of the array (`t[i]`) is defined as `*(t+i)`. As the latter is a valid expression for any typed pointer p as well, C allows the usage of `p[i]` for the typed pointers too.

Pointer arithmetics makes pointers a very flexible and efficient tool in C, which has largely contributed to the popularity and success of the language.

Reference type in C++

In addition to pointers inherited from C, C++ has introduced reference types as well. For a type T, *$T&$* denotes the reference type to T. A reference is similar to a constant pointer. References must be initialized at declaration, and from that point on they behave like an alternative "name" – an alias – to the memory area used at initialization. For reference types there is no need to denote reference operation, any type T can automatically be converted to its reference type *$T&$*.

In the case of reference types, there is no need to explicitly indicate dereference either. As the reference itself is constant after initialization, all operations apply to the referenced object. The most important usage of reference types is in the formal parameter list of subprograms. By using reference types we can realize call-by-reference argument passing in C++. C only supports call-by-value. For out and in-out parameters[8] we have to use pointers in C. However, pointers are not guaranteed to be initialized, their usage is risky. References are always initialized, they offer a lot safer alternative in C++.

Pointers to constants

Let us assume that p is a pointer to an integer type. Using dereference, we can access the object it points to, and we can change the object, e.g. by adding another integer to it. Let us assume that c is an integer constant. If we could assing the reference of c to p, we could modify the value of c through p because at the point of dereference it is not possible to decide whether the referenced object is a constant or not. To prevent such errors, it is not possible to assing the reference of a constant object to a pointer.

To resolve this restriction, C and C++ have introduced the constant type **const** T for each type T. T and **const** T are distinct types; therefore, their pointer types are distinct as well. We get a compile time error if we try to modify the referenced object through a **const** $T*$ pointer. The conversion from T to **const** T is automatic; likewise $T*$ can automatically be converted to **const** $T*$. Therefore, a **const** $T*$ pointer can point to non-constant objects of type T. Such a pointer is a *constant view* of the variable object.

The most common usage of constant types – constant pointers in particular – is in the formal parameter list of subprograms. If the subprogram does not want to modify the value of the referenced object, it is a good practice to use the constant pointer type. This has two advantages. First, it makes the subprogram safer as it cannot change the value of the program accidentally – if it still changes the value, we get compilation a error. Second, the subprogram will be more widely applicable as it can be invoked with constant objects in

[8] Depending on the direction of communication, parameters of a subprogram can be *in*, *out* or *in-out*. For the latter two the subprogram can modify the values of the actual parameters. For further details, see Chapter 7

the actual parameters. The following example shows the implementation of the standard library function *strcpy*:

```
char* strcpy(char* dest, const char* src)
{
    char *d = dest;
    while (*src != '\0') *d++ = *src++;
    *d = *src;
    return dest;
}
...
char buffer1[100], buffer2[100];

strcpy(buffer1, "Hello world!");
strcpy(buffer2, buffer1);
```

Ada 95 distinguishes access types based on whether it is possible to modify the referenced object via the access. This distinction is made in the declaration of the access type. Only read only (*constant*) access types can reference constants.

```
type IntegerAccess is access all Integer;
type ConstantIntegerAccess is access constant Integer;

IP : IntegerAccess;
KP : ConstantIntegerAccess;
I  : aliased Integer := 21;
K : aliased constant Integer := 42;
begin
    IP := I'Access;  -- Correct.
    IP := IP + 1;    -- Correct.
    IP := K'Access; -- Compilation error!
    KP := K'Access; -- Correct.
    KP := KP + 1; -- Compilation error!
    KP := I'Access; -- Correct.
    KP := KP + 1; -- Compilation error! KP is constant view of I.
end;
```

Notice that updating the access *KP* is allowed as the access itself is not constant, only the object it points to.

5.7 Expressions

Expressions in programming languages are usually the function calls – including operators –, evaluations of variables and constant and combinations of these. Expressions are essential for implementing computations. In some languages,

such as C or C++, expressions are the simplest statements of a program. In ALGOL 68 all statements, including control statements, are considered *unit*s, i.e. expressions.

5.7.1 Structure of expressions

An important property of functions is *arity*, that is, the number of arguments the function accepts – functions of arity 1 are called *unary* functions, functions that accept two arguments are called *binary*, and in general, functions accepting k arguments are k-*ary*. In this section if we need to explicitly denote the arity of a function we will use a superscript like f^k. Constants can be treated like functions of arity 0. If we loosen our definition of functions a bit, variables and the control statements, blocks and other units of ALGOL 68 may also be treated as functions, and thus expressions are just function calls nested at arbitrary levels.

Operators are special functions. The primary difference is in the notation used to invoke these functions and the identifiers used to denote them. The set of operators in most languages is fixed. Even in languages which allow overloading of identifiers (see Section 7.6), overloading operators may not be supported. For example in Java, method identifiers of classes can be overloaded but operators cannot. Very few language allows creating new operators. One of these languages is Eiffel.

Operators are typically indentified by symbols (e.g. `+`, `-`, `/`, `*`, `++`, `<`, etc.) instead of names, though in some languages there are named operators as well – for example **new** or **delete** in C++. The other difference compared to ordinary functions is the form of applying operators. Ordinary functions are usually called using *prefix* notation – the name of the function is followed by the list of arguments –, where the arguments are listed in parentheses. The reason for this form is that programming languages allow functions of arbitrary arity, and the parentheses helps the compiler to determine the list of actual arguments – $f(arg_1, \ldots, arg_k)$. The arity of operators is fixed – even if the language supports operator overloading, it is not possible to change the arity of the operator. Therefore, there is no need for the parenthesis. Operators are often used in *infix* and *postfix* notation as well, especially when this is the common usage of the operator in mathematics.

Operators can be used in four different notations:

- In *prefix* – or Polish – notation, operators precede their arguments in expressions – e.g. for a k-ary operator: $op^k \ arg_1 \ldots arg_k$. As the arity of operators is fixed and well known, such expressions can easily be processed and evaluated from right to left. Unfortunately, this notation is quite hard for human readers – for example `-*bb**4ac` is the prefix form of $b^2 - 4ac$.
- In *postfix* – or reverse Polish – notation, the operator is placed after the operands – e.g. for a k-ary operator: $arg_1 \ldots arg_k \ op^k$. The example

above in postfix notation is `bb*4a*c*-`. These expressions are also very easy to evaluate using a simple algorithm and a stack: from left to right processing, if the next item is an operand – variable or constant –, it is pushed to the stack. If a k-ary operator is encountered, k elements are poped from the stack, the operator is evaluated on them, and the result is pushed back to the stack. At the end of the processing, the stack will contain a single element, the value of the expression. Notice that during the evaluation we used the fact that the arity of the operators is known.

- In *infix* notation, binary operators are placed between the two operands in $arg_1\ op^2\ arg_2$ form. This notation is used in mathematics, and therefore, it is the common notation of arithmetic operators. The drawback of this notation is that the evaluation is ambiguous. When evaluating the expression `b*b-4*a*c`, the order of evaluating operators cannot be determined based on syntax. To disambiguate this expression, we need two more notions *precedence* and *associativity* which are discussed in detail in Section 5.7.1.
- *Mixfix* operators do not properly fit the groups above. An example is the conditional expression operator of C-like languages, which has the form of ⟨*condition*⟩ `?` ⟨*expression-1*⟩ `:` ⟨*expression-2*⟩.Another group of examples are the control statements of ALGOL 68.

Programming languages usually support a mixture of prefix, infix, postfix and mixfix operators. Unary operators – having only one parameter – often use prefix notation, though for example the `++` and `--` operators of C-like languages can be used both in prefix and postfix form. Their mode of evaluation actually differs in the two notation, and in C++, which supports overloading operators, the two forms can have completely different implementations as well.

Binary mathematical operators, such as arithmetic operators or relational operators, are typically used in infix mode as this is the standard notation in mathematics. However, there are exceptions as well. In CLU all functions are called in prefix notation, the arguments are listed in parentheses, after the identifier of the function. To simplify the writing of mathematical expressions and to improve their readability, CLU provides an alternative notation called *syntactic sugar* for certain function calls. The binary `add(a,b)` function call can be written as `a + b` as well. It is important to notice that this is simply an alternative notation which the compiler replaces with the original prefix form function call as a preprocessing step, before attempting to interpret the expression at all. This means that the `a + b` notation is usable on *any* type if it has a binary function called `add`.

Functional languages, such as LISP, often prefer prefix notation for arithmetic operations as well. An interesting property of LISP is that – due to the extensive usage of parentheses – associative operators can be used with arbitrary arity without any ambiguity. For example, the expression `( + 1 2 3 4)` is a 4-ary addition which evaluates to 10.

5.7.2 Evaluating expressions

In the previous section, we have defined expressions as function calls embedded at arbitrary levels. To help the description of expression evaluation, we will define the notion of *expression tree*:

- The expression tree of a zero argument function – including variables or constants – consists of a single node labeled with the function;
- The root of the expression tree of a k-ary function ($k > 0$) is labeled with the function, and it has k subtrees, that is, the expression trees of its k operands;
- The root of the expression tree of a control statement is labeled with the control statement, and it has edges pointing to the roots of the expression trees of the expressions used in the control statement.

The evaluation of an expression is simple if we have its expression tree. The tree needs to be evaluated in a bottom-up order, starting with the leaf nodes.

- A leaf node is always a 0-argument function, variable or constant. It can be evaluated at any time.
- A node representing a k-ary function can be evaluated once all of its direct descendants have been evaluated. The value of the node is the result of the function applied on the values of the operand nodes. If f^k be the label of the node and the values of the direct descendants are $v_1, \ldots, v_k$, the value of the node is $f^k(v_1 \ldots, v_k)$.
- To evaluate a node labeled with a control statement, we need to execute the statement in its label. During the execution we may have to evaluate the direct descendant expression trees several times. The value of the node is the value of the last evaluated expression.

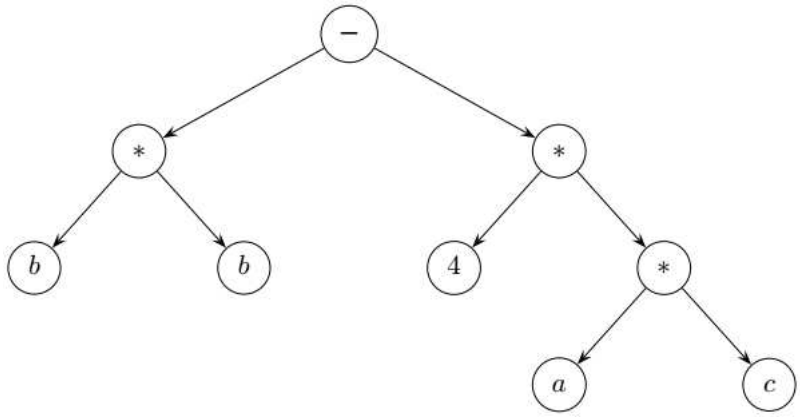


Figure 5.3: Expression tree of the expression $b^2 - 4ac$

Figure 5.3 is the expression tree of the expression $b^2 - 4ac$. From the prefix `-*bb**4ac` and postfix `bb*4a*c*-` variants, we would get the exact same tree. Both forms lead to the same syntax tree. In these notations by the means of syntax we can ensure unambiguous semantics.

The results are much less clear when using infix notation. The expression tree in Figure 5.3 matches the expression `b*b-4*a*c`, but so does the tree in Figure 5.4 as well.

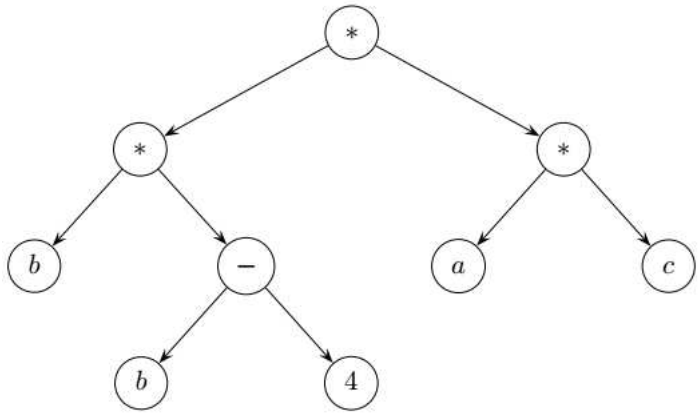


Figure 5.4: Expression tree for the expression $b(b-4)ac$

In case of infix notation, the means of syntax are not enough to unambiguously define the expression tree and thereby the semantics of the expression. This problem is not specific to programming languages; in fact this notation is borrowed from mathematics. To achieve unambiguity, mathematics uses a secondary tool, the so called *precedence*. Operator precedence is a rule used to define the order of evaluating operators in a mathematical expression. This rule states that in the expression $b^2 - 4ac$, we first need to calculate the multiplications, then the subtraction; and therefore, the correct interpretation is the one shown in Figure 5.3. The ordering defined by the rules of precedence can be overridden by using parentheses.

By using precedence, we can classify the operators; which, however, is not enough to completely resolve the ambiguity of expression trees. Operators of the same precedence level can be present in the expression. There are different operators on the same level – e.g. addition and subtraction – and the same operator may be used multiple times in the expression. In mathematics this is not a problem – operations can be performed in arbitrary order, as it does not affect the end result. However, in the world of computers this is not always true. Assume that we have a small program written in Ada. We use an integer type with a type-value set of the $-100..100$ interval, and we want to evaluate the expression $70+70-50$. In Figure 5.5 we can see the possible expression trees for this expression. From the point of view of mathematics the two trees give

the same result. However, in light of the (known) details of the implementation there is a subtle difference between the two. If we use the first variant, the result will be *90*, as expected. However, if we use the second variant we get a *CONSTRAINT_ERROR* exception because a partial result (i.e. *70+70*) falls outside of the permissible range of the used type.

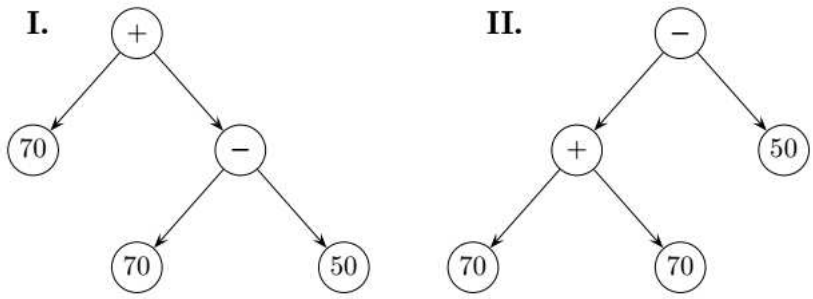


Figure 5.5: Two different expression trees for the expression $70 + 70 - 50$

We have a similar problem when evaluating the expression $10 * 10/100$. The mathematical value of the expression is 1, regardless of the evaluation order, but the result can be very different in many programming languages. In mathematics it is $10/100 = 0.1$, while in most programming languages the value of the expression 10/100 is 0. The reason for the difference is that in mathematics there is only one division operator, and the "type" of the operation – whether it is integer or real – is determined after performing the division. However, in most programming languages *the type of an expression depends only on the types of the operands, and it is determined at compilation time.* In many programming languages the operator / denotes integer division, and therefore, its result is an integer too.

To resolve these ambiguities and determine the order of evaluation of operators of the same precedence level, the notion of *associativity* has been introduced. An operator is called *left associative* if the evaluation of the expression $v_1 \odot v_2 \odot \cdots \odot v_n$ happens *from left to right*, i.e. it corresponds to the evaluation of the expression $(\ldots(v_1 \odot v_2) \odot \ldots) \odot v_k)$. Similarly, *right associative* operators are evaluated *from right to left.* The direction of associativity within a precedence class is uniform; therefore, it defines the evaluation order for mixed use of the operators as well. The evaluation order can be overridden using parentheses.

The majority of operators in most programming languages are left associative. C-like languages show a more colorful picture. Table 5.2 summarizes the operators of C in a decreasing order of precedence, defining the direction of associativity for each precedence class.

Prec.	Operators	Associativity
1.	Parentheses: `(` and `)`	none
2.	Function calls: `()`, member selection: `->` and `.`	left to right
3.	Unary operators: sign operators: `+` `-`, boolean negations: `!`, arithmetic (bitwise) negation: `~`, reference: &, dereference: `*`, pre- and postfix increment: `++`, -decrement: `-`, typecast: (⟨*type*⟩), size: `sizeof`	right to left
4.	Multiplicative operators: multiplication: `*`, modulus: `%`, division: `/`	left to right
5.	Additive operators: addition: `+`, substraction: `-`	left to right
6.	Bitwise shift: left: `<<`, right: `>>`	left to right
7.	Relationa operators: `<`, `<=`, `>` and `>=`	left to right
8.	Equality: `==`, inequality: `!=`	left to right
9.	Arithmetic (bitwise) "and": &	left to right
10.	Arithmetic "exclusive or": `^`	left to right
11.	Arithmethic "or": `|`	left to right
12.	Boolean "and": &&	left to right
13.	Boolean "exclusive or": `^^`	left to right
14.	Boolean "or": `||`	left to right
15.	Assignment: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, & =, `^=`, `|=`, `<<=`, and `>>=`	right to left
16.	The comma operator: `,`	left to right

Table 5.2: Precedence classes and their directions of associativity in C.

The precedence of operators and their directions of associativity are not modifyable in most languages. An interesting exception is ALGOL-68, where the developer can specify the precedence of non-unary operators on a scale from 1 to 10. The codesnippet below will print 45:

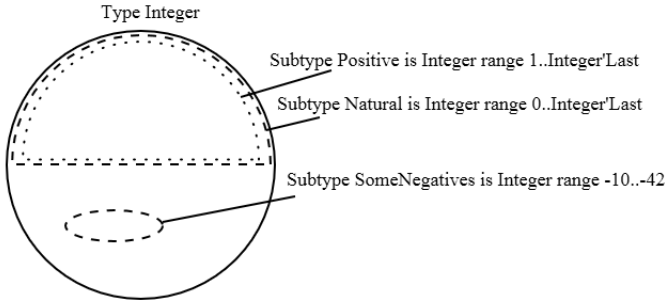
```
begin
    prio +=3, *=2;
    print(6 + 3 * 5);
end
```

5.8 Other language specialties

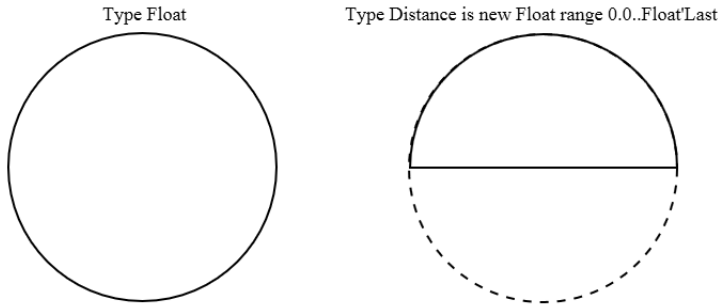
In this section we will discuss other language peculiarities related to types, which did not fit in the with taxonomy-driven overview provided above.

5.8.1 Ada: Type derivation and subtypes

In Ada types may be created by using two methods.



a.) Type value set of type Integer and its subtypes



b.) Type value set of type Float and its derived type Distance

Figure 5.6: Difference between type derivation and subtyping

Subtypes are created by constraining the type-value set of an existing type. Subtyping does not create a new type, it simply defines a designated subset of the base type (see part a. of Figure 5.6). Therefore, objects of the subtype are objects of the base type as well. They can be assigned to each other; the assignment is correct at compilation time, but during execution a range check is performed. If the range check fails, a *CONSTRAINT_ERROR* exception is raised. In the sample below we may see two standard subtypes of the language:

subtype *Natural* **is** *Integer range 0..Integer'Last;*
subtype *Positive* **is** *Integer range 1..Integer'Last;*

In contrast to subtyping, *type derivation* creates new types by "copying" an existing type (see part b. of Figure 5.6). The derived type will use the representation of the base type and will inherit its set of operations, which, however, will be treated as a completely independent set. The derived type is a new type, assignment between the two types is considered invalid. However, the language allows the use of explicit – interpretation changing – conversion between them.

What are the benefits of type derivation? Assume that we need to create an application which can determine speed (v) based on distance (s) and time (t) by using the well-known equation $v = s/t$. Distance, time and speed are real quantities which suggests the following solution:

```
S : Float := 42.0;
T : Float := 21.0;
V : Float := S * T;
```

However, the solution contains a ”small” error and it will produce false results. Finding this error might require tedious testing and debugging, depending on how deep this small calculation is embedded into our system.
The improved solution below uses type derivation:

```
type Distance is new Float range 0.0 .. Float'Last;
type Time   is new Float range 0.0 .. Float'Last;
type Speed  is new Float range 0.0 .. Float'Last;

function "/"(S: Distance; T: Time) return Speed is
begin
   return Speed(Float(S)/Float(T));
end "/";

S : Distance := 42.0;
T : Time   := 21.0;
V : Speed  := S / T;
```

In the solution above, we have created three distinct types for the three physical quantities and have defined a division operation which only allows the division of distance by time and ensures that the result is speed. To implement this operation, we have to use the conversion between the base type and the derived types. In the example, this has been the only operation between these types, which has prevented the error of the first example. Even if we define, for example, multiplication between *Speed* and *Time*, the operation ensures that the result is of type *Distance*, which cannot be assigned to any other variable. Overall, by using the strong typedness of Ada, we may express semantic relations between these quantities, which the compiler can verify at compilation time.

5.9 Summary

In the following, we summarize what aspects of the languages we need to examine when studying its type system:

- *Is the language strongly or weakly typed?*
  What kind of type checks are made by the compiler? Is the type of objects, variables, functions, subprogram formal parameters, etc. known

at compilation time? These question are fundamental for the usage of a language. Strongly typed languages are Ada, CLU, Java, C++, etc.; weakly typed languages are Smalltalk, Perl, JavaScript, dBase, etc.

- *Does the language use type classes?*
  As we have seen the types of programming languages may be assigned to the type classes defined in our taxonomy or to some other similar taxonomy. Sometimes these type classes are used consciously in the specification, and they are applied consistently in various constructs of the language. For example in Ada the type class determines the set of operations available for a type and the usability of the type. Type classes also appear as the "type" of the formal parameters of generics. In other languages the role of type classes is weaker but still present. For example in Pascal, arrays can be indexed by discrete types and there are operations which are uniformly applicable to all discrete types. Finally, in some languages the classification of types is quite artificial, which only helps when comparing the languages to each others. Such languages are C or FORTRAN.
- *What are the built-in types of the language?*
  The set of built-in types shows a great degree of similarity in the languages though there are smaller differences, e.g. in the precision of representation of numeric types. Nevertheless, there are types that are rather rare, e.g. the type of complex numbers which is supported in a few languages (e.g. FORTRAN or ALGOL 68) only.
- *Does the language support enumeration types?*
  Enumeration is the simplest type construction method, yet programming languages are divided in supporting it. Even modern languages such as Java – before Java 5.0 – or Eiffel do not support it, or provide only "syntactic-sugar-like" support for enumerations. However, the example of Java shows that there is great demand for this type construct among developers.
- *If there is no enumeration in the language, is there an idiomatic way to substitute it?*
  We have seen some examples of how enumerations can be substituted in languages which do not support it like the labeled unions of CLU, the **unique** declarations of Eiffel, or the type safe enum pattern of Java. In the study of these solutions, an important criterion of evaluation is if and how they provide type safety.
- *What are the integer types of the language?*
  All programming languages support some built in integer types, but they show a great variance in terms of type-value set. In most languages, integer types are based on the integer arithmetics built into the CPU – mostly for performance reasons – but the actual set of types can vary. Usual ones are the 8, 16, 32 or 64-bit signed or unsigned integer types, but in case of Python we have seen an example of a built-in integer type which is only bound by the size of memory.

- *Does the language have a boolean type?*
  In many languages, there is some boolean type, but for example in C integers play the role of booleans.
- *If there is a boolean type, does it belong to enumerations or integers?*
  Even those languages which have a built-in boolean type show variance on where it belongs in the type taxonomy. In Pascal or Ada the boolean type belongs to enumerations, it has enumeration-like properties. In other languages, such as CLU or Delphi it belongs more to integer types.
- *Are boolean operators lazy or greedy or does the language support both?*
  The majority of languages support lazy evaluation only, but as we have seen Ada, CLU and Eiffel offer a choice for the developer. Binary boolean operators of these languages have a lazy and a greedy version as well.
- *Is the character type an enumeration or an integer type?*
  Similarly to boolean type, character types are half way between integers and enumerations. In some languages – e.g. C, C++ or Java – they fall under the category of integers, in other languages such as Ada or Pascal they are enumerations.
- *Does the language define character encoding?*
  This question is mostly relevant in languages where characters are an integer type. In the case of C we have seen that the language is mostly independent of the actual encoding, while Java explicitly specifies the encoding to be used.
- *What real types does the language support?*
  Most languages nowadays use the floating point arithmetics defined in IEEE 754 mainly becuase this is supported by most modern, generic purpose CPUs. However, there are different levels of precision. Additionally, some languages – for example, Ada – support fixed point representation as well.
- *What memory management does the language support?*
  In the case of C or C++, we have seen examples where the developer is fully responsible for memory management. However, most object-oriented languages support garbage collection instead of explicit deallocation.
- *If the language supports garbage collection, does it still allow explicit deallocation?*
  When implementing performance critical systems, it might be necessary for the developer to take full control over memory management. Some languages, which are essentially garbage collection based – for example Ada –, offer means for the developer to explicitly deallocate certain objects. In other languages, it is possible to trigger garbage collection at least – e.g. in Java using the *System.gc()* method call.
- *Does the language support untyped pointers?*
  C and Pascal have a pointer type which is untyped, meaning it points to no particular data types. The advantage of this pointer type is that it

is convertible to all typed pointers. Such pointers are inherently unsafe; therefore, many languages do not support it. In modern object-oriented languages where there is a common class hierarchy, references of the common base class offer a much safer alternative.

- *Does the language support subprogram pointers?*
  In C, C++ or Ada 95, pointers can reference subprograms, which can be invoked through the pointer. Other languages such as Ada 83 or Java do not offer such possibility, but functors can replace subprogram pointers in object-oriented languages.
- *Does the language have a reference operator?*
  In Ada 95, C or C++, it is possible to obtain a pointer to an automatic or static variable using a reference operator. Other languages limit pointer types to dynamic variables only.
- *Is there constant pointer in the language?*
  Can a pointer be constant? Can a pointer point to a constant? Can a constant be modified through a pointer?
- *What are the specifics of the type system of the language?*

5.10 Exercises

Exercise 5.1. Give an example of a strongly typed programming language which is not staticly typed!

Exercise 5.2. What is the type of the following literals in Java?

```
5465435      '\u2343'
123          "\u2343"
'a'          "\\u2343"
3.14
```

Exercise 5.3. Compare the advantages and disadvantages of fixed and floating point representations!

Exercise 5.4. What are the risks of using integer constants as a substitute for enumerations?

Exercise 5.5. Create an enumeration type in Java for the days of the week. Provide a solution for Java 1.2 as well as for Java 5.0.

Exercise 5.6. What is the difference between the following two expressions of C: *$A&B$*, *$A&&B$*?

Exercise 5.7. What is the value of the expression `3 / 2` in C, Ada and Pascal? Explain the difference!

Exercise 5.8. What is the difference between lazy and greedy boolean operators? Give examples for the application of each!

Exercise 5.9. Why is the following C language code snippet incorrect?

```
long* find(long *first, long *last, long value)
{
    long *p = first;
    while (p <= last) {
        if (*p == value) {
            return p;
        }
        p += sizeof(long);
    }
    return NULL;
}
```

Exercise 5.10. Why do we need the **aliased** qualifier for automatic variables in Ada 95 if we want to use the reference operator on them?

Exercise 5.11. Create a program to evaluate expressions given in reversed Polish notation!

Exercise 5.12. What does *forward declaration* mean and why do we need it?

5.11 Useful tips

Tip 5.1. The difference between strongly types and statically typed languages is that the former only requires that the language guarantees that the usage of types is consistent, while the former requires knowing the exact type at compilation time. Consider the effects of polymorphism.

Tip 5.2. Consider the literal definitions of Java and the type-value set of the primitive types.

Tip 5.3. Consider the precision loss of operations and the elements of the type-value set. Notice that some property being advantage or disadvantage depends on the application!

Tip 5.4. Consider how many ways an enumeration value can be abused if its actual type is integer.

Tip 5.5. In Java 5.0 enumerations can be defined simply by listing their elements, the syntax is very similar to C/C++ enums. The same could be achieved before Java 5.0 by a class. Keep in mind that enumerations are: typesafe, ordered, can

be converted to and from *String*s, can be compared using the == operator and they can be enumerated. Try to make sure your implementation satisfies all these requirements!

Tip 5.6. Consider the difference between bit arithmetics and boolean operations!

Tip 5.7. Consider how the different languages handle the fact that 3/2 is not an integer value!

Tip 5.8. Consider boolean expressions where evaluating the second argument has some side effect!

Tip 5.9. Consider what *p=p+1* means for a pointer p in C pointer arithmetics! What is the result of *p += sizeof(long)*?

Tip 5.10. Consider what happens if you set a global reference to an automatic variable and the execution of the corresponding block is finished?

Tip 5.11. Evaluation of an expression in reverse Polish notation can be done with the help of a stack. The expression is read token-by-token from left to right. If the next token is an operand – variable or literal – then it is pushed to the stack. If the next token is an operator with arity n then n elements are popped from the stack, the operator is applied to them and the result is pushed to the stack. If the expression was well-formed, at the end of the algorithm – after processing the last token – there is only one element at the bottom of the stack which is the value of the expression.

Tip 5.12. Consider recursive data constructs (lists, trees, etc.) which are implemented by using pointers. How do we define the required types?

5.12 Solutions

Solution 5.1. In staticly typed programming languages the type of all expression can be determined at compilation time. A programming language is called strongly typed is it guarantees that the type of all expressions is consistent, even if the exact type of the expression cannot be determined. Clearly all staticly typed programming language is strongly typed. The simplest example of strongly typed programming languages which are not staticly typed are strongly typed object oriented programming languages which support polymorphism, such as Java, C++ or Ada95. In these languages the exact type of certain variables, functions or expressions cannot be determined at compilation yet the consistency of all expressions is guaranteed by the language. For example in the following Java example the dynamic type of `o` can literally be anything. Yet, the expression printing the object is consistent:

```
String className = System.in.readLine();
Class c = Class.forName(className);
Object o = c.newInstance();
System.out.println(o.toString());
```

Solution 5.2.

```
5465435   : int
123       : int
3.14      : float
'a'       : char
'\u2343'  : char
"\u2343"  : String
"\\u2343" : String
```

Solution 5.3. Fixed and floating point representations complement each other in terms of features. Whether these are advantages or disadvantages depends on the usage.

The precision of fixed point types are is defined as the number digits they can represent. This precision loss of fixed point operations is independent of the operands, it only depends on the precision of the type. This property can be important for example in accounting applications. The drawback of this representation is that fixed point types cannot efficiently represent very small and very large values at the same time. A given number of digits has to be split up between the integer part and the fraction.

The precision of floating point numbers is defined as the number of digits used to represent the mantissa and the exponent. The precision loss of floating point operations depends on the scale of the operands and the scale of the result as the mantissa always contains the most significant digits of the represented value. This property allows making calculations with numbers of different scale which is often a requirement in scientific applications.

Solution 5.4. There are multiple risks:

- Lack of type safety: the value of one enumeration ”type” can be assigned to a variable of another. Or even to any integer variable.
- Similarly, any integer value can be assigned to any enumeration, even values outside of the typevalue set.
- The enumeration ”types” inherit a large set of operations (e.g. arithmetics) which is not meaningful for them but which are defined not only for the type but for any pairs of enumeration types or even for enumerations and integer types.

Solution 5.5. Java 1.2 (or rather ”before Java 5.0”):

```
public final class DayOfWeek implements Serializable, Comparable {
    private static final long serialVersionUID = 358506482305518507L;
    private static int counter = 0;
    private static final List VALUES = new ArrayList();
    public static final DayOfWeek SUNDAY = new DayOfWeek("SUNDAY");
    public static final DayOfWeek MONDAY = new DayOfWeek("MONDAY");
```

```
    public static final DayOfWeek TUESDAY = new DayOfWeek("TUESDAY");
    public static final DayOfWeek WEDNESDAY = new DayOfWeek("WEDNESDAY");
    public static final DayOfWeek THURSDAY = new DayOfWeek("THURSDAY");
    public static final DayOfWeek FRIDAY = new DayOfWeek("FRIDAY");
    public static final DayOfWeek SATURDAY = new DayOfWeek("SATURDAY");

    private final String name;
    private final int ordinal;

    private DayOfWeek(String name) {
        this.name = name;
        this.ordinal = counter++;
        VALUES.add(this);
    }

    public static DayOfWeek[] values() {
        DayOfWeek[] res = new DayOfWeek[VALUES.size()];
        for( int i = 0; i<res.length; ++i ) {
            res[i] = (DayOfWeek) VALUES.get(i);
        }
        return res;
    }

    public String name() {
        return this.name;
    }

    public static DayOfWeek valueOf(String name) {
        for( Iterator i = VALUES.iterator(); i.hasNext(); ) {
            DayOfWeek d = (DayOfWeek) i.next();
            if( d.name().equals(name) ) return d;
        }
        throw new IllegalArgumentException("No enum constant " +
              DayOfWeek.class.getName() + "." + name);
    }

    public String toString() { return this.name; }

    // Overriding readResolve to ensure that only one object with a given
    // ordinal can exist. This guaratees the e1.equals(e2) implies e1 == e2
    private Object readResolve() throws java.io.ObjectStreamException {
        return VALUES.get(ordinal);
    }

    public int compareTo(Object o) {
        return this.ordinal-((DayOfWeek)o).ordinal;
    }
}
```

Java 5.0:

```
public enum DayOfWeek {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
}
```

Solution 5.6. The expression *$A & B$* is an arithmetic (bitwise) "and" operation between *A* and *B*. *A* and *B* must be some integer type or a type that can automatically be converted to an integer type. The result of the operation will be an integer type too, the smallest integer type both *A* and *B* can be automatically converted to.

The expression *$A && B$* is a boolean "and" operation between *A* and *B*. *A* and *B* can be any type that can be converted to an integer type, any non-zero value represents the boolean value *true*, zero represents *false*. The result will be of integer type. The value will be 1 if both operands were true (non-zero) otherwise the result will be 0.

Solution 5.7. In C the value will be 1 as operator / with integer operands denotes integer division. The type of the expression is **int**. Due to automatic conversions of C, this can then be assigned to various integer types.

In Ada the result is 1 too but the type is *Universal_Integer*, a special type of Ada, which can automatically be converted to any integer type. This is the type of all integer literals to make sure that they can be assigned to variables of any integer type.

In Pascal, the type of the expression is some real type and the value is 1.5. Even though the operands are integer literals, the operator / in Pascal denotes real division.

Solution 5.8. The difference between lazy and greedy boolean operators is that in case of lazy operators the second operand is not evaluated if the value of the expression can be determined based on the first operand only. For example in ADA the expression *A* **or else** *B*, if the value of *A* is true, the expression *B* never gets evaluated. If *A* and *B* are side effect free expressions, this is a simple performance enhancing optimization. However, if evaluation of expression *B* has some side effect, we might want to force the evaluation of *B* even if the result does not contribute to the value of the expression. For this cases Ada offers the operator **or**, the greedy version of **or else**, which always evaluates both arguments. Similarly the boolean **and** operation can also have greedy and lazy variants. Many programming language offers only one version - typically the lazy one - though there are some, such as Ada or Eiffel where both variants are available for the developer.

Solution 5.9. The code snippet demonstrates a typical mistake in using C pointer arithmetics. The developer wants to advance the pointer *p* to the next long value in the array of longs and adds the size of long (the number of bytes occupied by a long value) to pointer *p*. However, in case of typed pointers adding 1 to a pointer advances the pointer by 1 unit of the referenced type, in our case by **sizeof (long)**. Therefore, if for example **sizeof (long)** is 8, *p+=***sizeof (long)** will advance *p* by 8 longs.

Solution 5.10. Normally, automatic variables are allocated in the call stack and they are cleaned up when execution of the corresponding block of code is finished.

If an external (global) reference is set to such a variable during the execution of the block, the reference will point to an invalid memory address after the block is finished. To prevent this, Ada does not allow references to point at ordinary automatic variables. The aliased qualifier, introduces in Ada 95, informs the compiler that we might want to set a reference to an automatic variable. Such variable will then be allocated on the heap; therefore they can survive finishing the declaring code block.

Solution 5.11.

```
import java.util.HashMap;
import java.util.Map;
import java.util.Stack;
public class ReversePolishEvaluator {
static interface Expression {
    public Integer evaluate( Map<String, Integer> variables);
    public int getArity();
    public void setArguments(Expression[] args);
}
static abstract class InfixOperationExpression implements Expression {
    protected Expression left;
    protected Expression right;
    public int getArity() { return 2; }
    public void setArguments(Expression[] args) {
        if( args.length != 2 ) throw new IllegalArgumentException(
            "Invalid arguments: " + args);
        this.left   = args[0];
        this.right  = args[1];
    }
    protected abstract String getSymbol();
    public String toString() {
        return "(" + left.toString() + getSymbol() + right.toString() + ")";
    }
}
static class AddExpression extends InfixOperationExpression implements
        Expression {
    public Integer evaluate(Map<String, Integer> variables) {
        Integer l = this.left.evaluate(variables);
        Integer r = this.right.evaluate(variables);
        return Integer.valueOf(l.intValue() + r.intValue());
    }
    protected String getSymbol() { return "+"; }
}
static class SubExpression extends InfixOperationExpression implements
        Expression {
    protected String getSymbol() { return "-";}
    public Integer evaluate(Map<String, Integer> variables) {
        Integer l = this.left.evaluate(variables);
        Integer r = this.right.evaluate(variables);
        return Integer.valueOf(l.intValue() - r.intValue());
    }
}
static class MulExpression extends InfixOperationExpression implements
        Expression {
    protected String getSymbol() { return "*"; }
    public Integer evaluate(Map<String, Integer> variables) {
        Integer l = this.left.evaluate(variables);
        Integer r = this.right.evaluate(variables);
        return Integer.valueOf(l.intValue() * r.intValue());
    }
}
static class DivExpression extends InfixOperationExpression implements
        Expression {
    protected String getSymbol() { return "/"; }
```

```
    public Integer evaluate(Map<String, Integer> variables) {
        Integer l = this.left.evaluate(variables);
        Integer r = this.right.evaluate(variables);
        return Integer.valueOf(l.intValue() / r.intValue());
    }
}
static class VariableExpression implements Expression {
    private final String variableName;
    public VariableExpression(String variableName) {
        this.variableName = variableName;
    }
    public Integer evaluate(Map<String, Integer> variables) {
        return variables.get(variableName);
    }
    public int getArity() { return 0; }
    public void setArguments(Expression[] args) {
        throw new UnsupportedOperationException(
            "VariableExpression has no arguments.");
    }
    public String toString() { return this.variableName; }
}
static class LiteralExpression implements Expression {
    private final Integer value;
    public LiteralExpression(Integer value) { this.value = value; }
    public Integer evaluate(Map<String, Integer> variables) { return value; }
    public int getArity() { return 0; }
    public void setArguments(Expression[] args) {
        throw new UnsupportedOperationException(
            "LiteralExpression has no arguments.");
    }
    public String toString() { return value.toString(); }
}
static class ExpressionTokenizer extends java.util.StringTokenizer {
    private static final String DELIMITERS = " \t\n\r\f,";
    private static final String ADD = "+";
    private static final String SUB = "-";
    private static final String MUL = "*";
    private static final String DIV = "/";
    public ExpressionTokenizer(String expression) {
        super(expression, DELIMITERS);
    }
    public Object nextElement() {
        String token = super.nextToken();
        Expression res = null;
        if( token.equals(ADD) ) {
            res = new AddExpression();
        } else if( token.equals(SUB) ) {
            res = new SubExpression();
        } else if( token.equals(MUL) ) {
            res = new MulExpression();
        } else if( token.equals(DIV) ) {
            res = new DivExpression();
        } else {
            try {
                Integer value = Integer.decode(token);
                res = new LiteralExpression(value);
            } catch( NumberFormatException ignore ) {
                res = new VariableExpression(token);
            }
        }
        return res;
    }
}
    public static void main(String[] args) {
        if( args.length == 0 ) help(null,0);
```

```
        Expression exp = parseExpression(args[0]);
        Map<String, Integer> variables = parseArgs(args);
        System.out.println(exp.toString()+"="+exp.evaluate(variables));
    }
    private static  Map<String, Integer> parseArgs(String[] args) {
        Map<String, Integer> vars = new HashMap<String,Integer>();
        for(int i=1; i<args.length; ++i) {// args[0] contains the expression
            int p = args[i].indexOf('=');
            if( (p<1) || (p==args[i].length()-1) ) {
                help("Invalid variable specification: " + args[i], -1);
            }
            String name = args[i].substring(0, p);
            Integer value = Integer.decode(args[i].substring(p+1));
            vars.put(name,value);
        }
        return vars;
    }
    private static Expression parseExpression(String exp) {
        Stack<Expression> s = new Stack<Expression>();
        ExpressionTokenizer et = new ExpressionTokenizer(exp);
        for( ; et.hasMoreElements(); ) {
            Expression next = (Expression) et.nextElement();
            if( next.getArity() == 0 ) s.push(next);
            else {
                Expression[] args = new Expression[next.getArity()];
                for( int i = 0; i<args.length; ++i ) args[i] = s.pop();
                next.setArguments(args);
                s.push(next);
            }
        }
        return (Expression) s.pop();
    }
    private static void help(String message, int exitCode) {
        if( null != message ) System.err.println(message);
        System.err.println("Usage:\n\t" +
            ReversePolishEvaluator.class.getName()+
                        " <expression> [<variable>=<value>]...");
        System.exit(exitCode);
    }
}
```

Test is with the arguments: "`c x b * x x a * * + +`" `a` = 1 `b` = 2 `c` = 3 `x` = 5
The output should be: ((((`a` ∗ `x`) ∗ `x`) + (`b` ∗ `x`)) + `c`) = 38

Solution 5.12. Forward declaration of an identifier is a statement for the compiler. It declares the existence of some construct without giving its full definition. In the case of data types this means that we declare that the type with the given name exist but do not specify its structure at all.

The typical use case of forward declarations is the definition of recursive data types which use pointers. Consider the following code snippet in C:

```
struct Node;
typedef struct Node* BinaryTree;
struct Node {
  int        data;
  BinaryTree left;
  BinaryTree right;
};
BinaryTree create(int data) {
   BinaryTree res = malloc(sizeof(struct Node));
   res->data  = data;
   res->left  = NULL;
   res->right = NULL;
   return res;
}
```

The snippet is the definition of a *BinaryTree* data type. The **struct** *Node* represents a node of the tree, the tree itself is represented as a pointer to its root node. A node consists of three components: the data stored in the node, which is an integer value in the example and pointers to the left and right child nodes. The data structure is recursive. For the definition of the **struct** *Node* we need the definition of the **struct** *Node** pointer type. But we can define pointer types for existing types only. To resolve this "chicken and egg" problem, we use forward declaration. We inform the compiler that the type **struct** *Node* exists, without giving a definition. However, the representation of the **struct** *Node** pointer type does not depend on the internal structure of **struct** *Node*. Hence we can define the pointer type and after that we can give the definition of the type **struct** *Node* as well.

6 Composite types

No matter how rich the set of data types built into a programming language is, it will not be enough to solve all possible problems. In order to create usable models of the world, the language needs to provide means to create new types. Enumerations are one such means of creating a new type by simply enumerating its type-value set. In this chapter we look at the many composition methods which the different languages offer. By using these methods we show how new types can be created on the basis of existing types.

In the previous chapter we have seen the built-in data types which are available in almost all programming languages. They form a useful but basic toolset, which is far from enough for solving real life problems. Therefore, most programming languages[1] provide methods for extending the existing set of data types and for combining existing types into new ones.

In this chapter we will categorize the different type composition methods into three categories. We do not necessarily have all the three in a particular programming language and they do not necessarily have the same form; yet, they do preserve some of their generic properties, which we can study. These archetypical type composition methods are the following:

- *Cartesian product types,*
- *Union types,*
- *Iterated types.*

These names are the abstract names of these type constructs. Different programming languages can use different names for them and there can be variations between their realizations as well. First, we will study the common properties of these type constructs and then we will have a look at the specifics of certain languages. To emphasize the distinction between the construction in general and its realization in different languages, when we refer to a construct in general, we will use the abstract names given above. When referring to the language specific realization, the language specific term for the type construct will be used.

As in the previous chapter, when introducing types, the type composition methods will be described by the type-value set and by the operations of the created constructs.

[1] Interestingly, there *are* programming languages which offer no type composition methods. Assembly languages and some domain specific languages do not have this feature. For example SuperNova is a DSL designed for manipulating database tables and it has no type composition support. Nevertheless, generic purpose programming languages – with the exception of assembly languages – do offer type composition methods.

Note: this chapter will focus on the type composition methods of imperative programming languages. The composite types of functional programming language are discussed in Chapter 15, while Section 16.4 provides an introduction to data structures of logic programming languages.

6.1 Type equivalence

Equivalence of types is a crucial question of programming languages. The type-checker of a statically typed language must verify that the type of any expression is consistent with the type expected by the context in which that expression appears. Type equivalence determines if the value of an expression can be assigned to a particular variable, which formal parameter an actual parameter corresponds to, etc. In the context of type composition, we need to reconsider our definition of equivalence. Consider the following – Ada language – example:

```
R: Array(1..10) of Integer;
S, T: Array(1..10) of Integer;
```

Similar declarations are possible in most programming languages. The question is the relationship between the types of R, S and T.

Many strongly typed programming languages – including Ada – would consider the types of all three variables distinct. The form used to declare S and T is just a shortened variant of two distinct declarations. A variable declaration like above implicitly contains the declaration of an anonymous type as well. As the type is anonymous, we have no means to reference it later in any form. The three variable declarations create three – distinct – anonymous types. Therefore, due to strong typing, assignment between these variables is not allowed. Such languages consider types equivalent only if their names are equal. Therefore, the type equivalence of these languages is called *name equivalence*.

C uses a different equivalence definition. Writing multiple variable names in the same declaration – separated by comma – is semantically different from writing separate declarations with exactly the same type structure. In the example below, b and c has the same – anonymous – type, while a has a different one.

```
struct {
    int x;
} a;

struct {
    int x;
} b, c;
...
b = c;     /* Valid assignment */
a = b;     /* Invalid assignment */
```

Another difference is that by using the **typedef** keyword, we can introduce aliases to existing types. **typedef** does not create a new type just provides an alternative name for the same type. This form of type equivalence is called *declaration equivalence*, because two types are equivalent, if they lead back to the same original declaration.

```
struct A {
    int x;
};

typedef struct A T1;
typedef struct A T2;

struct A a;
T1 b;
T2 c;
...
c = a;    /* Valid assignment */
a = b;    /* Valid assignment */
b = c;    /* Valid assignment */
```

The other extreme end of type equivalence is *structure equivalence*, which considers two types equivalent when their structures are *isomorphic*, regardless of their names. Such equivalence is used for example in Modula-3. Obviously, this equivalence definition significantly weakens the type safety of the language as it considers semantically distinct types equivalent because they have the same representation. The following two type definitions are equivalent in Modula-3:

```
TYPE Stack = RECORD
   top: INTEGER;
   elements: ARRAY(1..100) OF CHAR;
END;

TYPE Text = RECORD
   length: INTEGER;
   text: ARRAY(1..100) OF CHAR;
END;
```

6.2 Mutable and immutable types

Another aspect of classifying types is mutability. An object is called *immutable* if its state cannot change after it has been created. A type is called immutable if its type-value objects are immutable. If the state of an object can change after creation, it is called *mutable*.

Even immutable types can have operations which ”modify” the type-values. The difference is that these operations always create a new object instance, which represents the result of the operation. The code snipped below is an immutable implementation of complex numbers in Java:

```
public final class Complex {
    private final double re;
    private final double im;

    Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public Complex add(Complex z) {
        return new Complex(re + z.re, im + z.im)
    }
    ...
}
```

The class uses canonical representation of complex numbers. The two fields of the object store the real and imaginary parts of the number. The fields have a modifier **final**, which ensures that the fields cannot get new values after initialization. This ensures the immutability of the object. Nevertheless, the type has an *add* operation, which adds a complex number to **this** object. The operation does not modify **this**, but rather creates a new object which represents the sum instead.

This solution might appear uneconomical and inefficient. Firstly, as in the case of evaluating a complex expression, a large number of temporary objects need to be created and destroyed until we get the final result. Indeed, if object creation is ”expensive” for a particular type, it is not recommended to make it immutable. However, immutable types have numerous advantages. Probably the most important one is that immutable types are inherently thread-safe, when using them in a multi-threaded application they can be safely shared between the execution threads without the need for any synchronization mechanism. As the object state cannot change, it cannot happen that a thread accesses the object in a partially updated, inconsistent state. Another advantage is that immutable objects can be safely shared between objects as well. Multiple objects can safely reference the same instance of an immutable type even if they have no semantic relationship. Modifications made through one instance will not interfere with the other. This way we can save lots of memory in our applications.

Depending on the nature of the problem at hand, use of immutable types makes our application safer, sometimes more efficient as well. Therefore, it is recommended to examine what support the different programming languages provide for implementing immutable types. The example above demonstrates

how immutability in Java can be achieved. Java alone makes use of this feature: it provides a large number of immutable classes in its standard libraries, e.g. *String*, *BigNumber*, *Integer* and all other wrapper classes of the primitive types are immutable. However, Java has no explicit means to declare a type immutable. We will see examples – e.g. in CLU – which make immutability an explicit property of the type.

For the effective use of immutable types, it is important for the language to store objects by reference. Immutable objects stored by value are constants, meaning they cannot be modified. Support of garbage collection is also very important, otherwise the burden of dealing with creating and destroying temporary objects during expression evaluation would make the type completely unusable. Thus, both CLU and Java share these properties, and they both use immutable types.

6.3 Cartesian product types

Assume that we design an application which manages the data of employees of a company. We have to store many different pieces of data about an employee: name, address, bank account number, salary, social security number, etc. We need this data with reference to every single employee. Furthermore, we need to keep the data which belongs to the same employee together as related data. Such relationships can be expressed using *cartesian product* types.

A cartesian product is a very common composition method supported by most languages (the exceptions are, for example FORTRAN and BASIC). It is typically called struct(ure) or record. However, classes of object-oriented programming languages are also special cases of cartesian product.

6.3.1 Type-value set

The declaration of a cartesian product typically consists of the enumeration of its component types and their respective *selector*s (see Section 6.3.2). The Ada code snippet below is a possible representation of the employee data from the example above:

```
Type Employee_Type is record
   Name          : Name_Type;
   Address       : Address_Type;
   AccountNumber : AccountNumber_Type;
   Salary        : Salary_Type;
End record;
```

The identifiers ending with *_Type* are the names of the component types, while the identifiers before the colons designate the selector of the corresponding component.

In cartesian products we can use the same component type – $\mathcal{T}_i$ – multiple times, but all selector names must be different. The order of specifying components is also relevant, as it may affect the memory usage of the cartesian product (see Section 6.3.3). In languages which use structural equivalence, two cartesian products are equivalent if their component types and their order of specification are the same.

Taking all this into consideration, the formal definition of cartesian product type is the following: $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$ are types and $T_1, T_2 \ldots T_n$ are their type-value sets, respectively. The type $\mathcal{T}$ is the *cartesian product* of the types $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$ if its type-value set is $T = T_1 \times T_2 \times \cdots \times T_n$.

6.3.2 Operations

Next, we will look at the operations of the created cartesian product type.

Selectors

The most important operation of cartesian product types is *selection*. If we have a type-value $t \in T$, where $t = (t_1, t_2, \ldots t_n) \wedge \forall i \in [1 \ldots n] : t_i \in T_i$, we likely want to access the t_i component values.

The functions $s_i : T \to T_i$ are called *selectors* if $\forall i \in [1 \ldots n] : s_i(t) = t_i$ using the notation above. Very often the type of the value of s_i is not T_i but T_i *reference*,[2] because through them we can modify the value of the corresponding component. In other words s_i is an *L-value*[3] even in those languages which otherwise do not support reference types.

Most programming languages use selectors in a qualified form – instead of $s_i(t)$ the notation $t.s_i$ is used. The selectors are identified as part of the declaration of the cartesian product.

The selectors – often called *field*s or *member*s of the type – provide access to the components of the cartesian product. Through them we may apply the operations of the component type to the corresponding component. Selectors are available in *all* languages supporting cartesian products as without them the constructed types would unusable.

Assignment

Selectors are essential in all languages, but the support for *assignment* is a lot controversial. The natural definition of the assignment of cartesian products

[2] See Section 5.6 for details about reference types.

[3] An *L-value – locator value* – represents an object with an identifiable location in memory, in fact a reference. An important property of L-values is that they can stand on the *left side* of assignments.

is the component-wise assignment. Unfortunately, this approach may not be applicable if one of the components have no assignment operations.[4]

Some languages do not consider these risks, and define assignment operation for cartesian products automatically. For example, in C the assignment of **struct**s is performed by copying the corresponding piece of memory byte by byte. It is the responsibility of the developer to avoid using assignment when the semantics is not appropriate.

A safer solution is offered by Ada where the developer can prevent assignment by declaring the type **limited**. Non-limited cartesian products have an implicit assignment operation which corresponds to the component-by-component assignment. This assignment operation is not allowed for limited types. Using a limited type as a component of a non-limited type leads to a compilation error.

A much more flexible solution would be if we could define the assignment operation ourselves. Unfortunately, in Ada, the assignment operation is a *statement*, which cannot be overridden by the developer. In C-style languages, assignment is an *operator*. If such a language (e.g. C++) supports *operator overloading* (see Section 7.6.1 for details), a solution is to override the default semantics if it is not suitable for a particular type. We can even prevent assignment – achieve similar semantics to that of limited types in Ada – by restricting the visibility of the assignment operator (see Section 4). An important consequence of this flexibility in C++ is that it cannot apply simple memory copying in the default implementation of assignment operator, but rather it has to perform assignments component by component. This way it can be ensured that the potentially overridden assignment operations are applied on the component types.

Constants

When dealing with assignment, we also need to check if the language allows the declaration of constants of the cartesian product type. Constant variables need to be initialized at the place of declaration. Therefore, to support them the language must allow the assignment of the cartesian product types, and we have to have means to specify the value. It can happen by referencing a variable – which raises the question how it has been initialized – or we need other means to create instances of cartesian products.

In object-oriented languages we can use the *constructors* of the type, which initialize an instance of the type based on some arguments. In non-object-oriented languages, if the language supports *dynamic evaluation* and function-like subprograms, we can create functions which return cartesian product objects. There exists another possibility if the language supports cartesian product literals (see Section 2.4 for details). Consider the following example in Ada:

[4] For example, if the corresponding component is a file, it is not guaranteed that it has assignment operations.

```
type Complex is record
    RE : Float := 0.0;
    IM : Float := 0.0;
end record;
type Point is record
    X : Float := 0.0;
    Y : Float := 0.0;
end record;
I : constant Complex:=(0.0,1.0);
Z : Complex;
P : Point;
begin
   Z:=(42.0,42.0);
   P:=(42.0,42.0); -- Aggregate type is Point, inferred from context.
   Z:=Complex'(1.0,1.0);--Direct indication of the type of the aggregate
   P:=(Y=>1.0, X=>1.0);
end;
```

Cartesian product literals – called *record aggregate* – are typed in Ada. In the initialization of the constant I, or in the first assignments to Z and P the type of the aggregates are *inferred* by the compiler. Therefore, though formally the aggregates assigned to Z and P are identical, they will be of different types. Nevertheless, it is more readable if the expected type is explicitly indicated as in the second assignment to Z. In the first assignments, the aggregates are specified using *positional notation*; the first value is assigned to the first field – in the order of declaration –, and the second vale is assigned to the second field. The last assignment demonstrates the use of *named components*. Positional notation is more compact; however, named components have a number of advantages:

- The code is more readable, it is clear which value corresponds to which component;
- The type of the aggregate is clearer;
- Components can be specified in an arbitrary order;
- The compiler can print more elaborate diagnostic messages.

Cartesian product literals are supported by other languages as well. For example in C and C++, literals can be specified by listing the values of the fields between curly braces – { and } –, but the type of the literal cannot be specified and named components are not supported, either.

Equality check

The natural semantics of equality is that two cartesian product objects are equal if they have the same set of components and the component values are equal. The challenges listed at assignment are present here as well but there are additional

complexities which prevent the trivial solution of byte-by-byte comparison of the corresponding pieces of memory. These issues are discussed in detail in Section 6.3.3. However, as a consequence of these additional challenges, programming languages are more restrictive on equality check of cartesian products. For example, in C the assignment of *struct* is allowed, but the equality is not defined for them.

WITH Statement

In Pascal-like languages, there is a special operation for cartesian product – record – types called **WITH**. A **WITH** statement specifies a record variable and a statement sequence. In these statements, the qualification of field identifiers may be omitted, if they are to refer to the variable specified in the with clause. The following code snippet is from Modula-2:

```
TYPE DateType = RECORD
   year: YearType;
   month: MonthType;
   day: DayType;
END;
TYPE ManType = RECORD
   name: NameType;
   bithday: DateType;
END;
VAR
   myDarling: ManType;
BEGIN
   WITH myDarlin.birthday DO
      year := 1974; month := January; day := 22;
   END;
END;
```

Usage of **WITH** statements simplifies access to fields of records and makes the code more readable, especially when dealing with records embedded at several levels.

6.3.3 Representation of cartesian product types

The representation of cartesian product types in memory is often language specific. Nevertheless, there are properties and characteristics which are common to most languages.

The physical representation of a cartesian product usually contains the components in their declaration order. Each component is represented according to the representation defined in its type. This is logical, the implementation of

the operations of the type will not change just because a particular object is a component of a cartesian product. At least *usually*.

In Ada for certain fields it is possible to specify their physical representation within the record, which may differ from their "natural" representation. The solution is provided by the selectors. Selectors can hide the actual representation from the operations of the type. When accessing a component through the selector, it can perform the required conversion between the different representation. Such conversion requires intricate knowledge of the represented types.

The example below (from [Ada95]) is the declaration of a record type which represents the state of a program. In the execution environment this state is represented on two words (64 bits altogether), where individual bits or small groups of bits have distinct functions. The specified record is an abstract, high-level, and strongly typed representation of this state descriptor.

Normally this record would be represented on several bytes according to the representation of the different component types. However, the fields of this record type are mapped to the corresponding bits of the state descriptor using the specified *record representation clause*. The usage of this clause help access of the components of this compact state descriptor through a strongly typed *view* which suits the strongly typed type system of Ada, and uses the built-in data types of the language. The usage of standard types and the representation clause also greatly improve the readability of the code, and ensure that all particularities of the used representation are described at a single location in the code.

```
Word : constant := 4;--storage element is byte, 4 bytes per word

type State        is (A,M,W,P);
type Mode         is (Fix, Dec, Exp, Signif);
type Byte_Mask    is array (0..7) of Boolean;
type State_Mask   is array (State) of Boolean;
type Mode_Mask    is array (Mode) of Boolean;

type Program_Status_Word is
  record
      System_Mask      : Byte_Mask;
      Protection_Key   : Integer range 0 .. 3;
      Machine_State    : State_Mask;
      Interrupt_Cause  : Interruption_Code;
      Ilc              : Integer range 0 .. 3;
      Cc               : Integer range 0 .. 3;
      Program_Mask     : Mode_Mask;
      Inst_Address     : Address;
end record;
```

```
for Program_Status_Word use
  record
      System_Mask    at 0*Word range 0 .. 7;
      Protection_Key at 0*Word range 10 .. 11;--bits 8,9 unused
      Machine_State  at 0*Word range 12 .. 15;
      Interrupt_Cause at 0*Word range 16 .. 31;
      Ilc            at 1*Word range 0 .. 1; -- second word
      Cc             at 1*Word range 2 .. 3;
      Program_Mask   at 1*Word range 4 .. 7;
      Inst_Address   at 1*Word range 8 .. 31;
  end record;
for Program_Status_Word'Size use 8*System.Storage_Unit;
for Program_Status_Word'Alignment use 8;
```

Consider for example the field *System_Mask*. Its type is *Byte_Mask*, an array of boolean values. The representation of *Boolean* is implementation specific, meaning it always takes at least one byte (see Section 5.4.3), but typically more, for better performance. However, the representation clause allocates 8 bits altogether to the whole array. The compiler must represent each element of the array on exactly one bit.[5] These bits function as some sort of state indicators, handling them as an array of booleans matches their function much better than interpreting *System_Mask* as an unsigned integer. Mapping the bit to an array makes their access simpler and the resulting code more readable.

Representation clauses are powerful tools when implementing a low-level system program, e.g. a device driver or a communication protocol. However, in most cases we do not want to specify the representation of our cartesian product types bit by bit, but leave it for the compiler. The order of components in the representation usually matches their declaration order in the type specification, and normally the usual representation of the component type is used. However, often the compiler adds *padding* to the components to ensure that they start at special addresses in the memory, e.g. at word boundaries. This is an optimization which makes the access of these component faster in memory. Clearly, the size of padding depends on the computer architecture, language, and even the realization of the language too. This usage of padding makes it difficult to check the equality of two cartesian product objects by comparing their representation byte by byte. Byte by byte comparison includes the padding as well. However, these bytes are often uninitialized, meaning their value is not specified.

6.3.4 Language specific features

Many programming languages have additional, language specific feature which make cartesian products in that language more flexible or safer to use.

[5] Though the specification of the language ([Ada95]) allows the use of representation clauses, not all compilers are able to generate code in line with to them.

Default values

In the declaration of Ada record types we may specify the default initial value of fields. When an instance of the type is created, the corresponding field is set to its default value. This feature is particularly useful when the record is used in the representation of an abstract data type. By using the default value we can ensure that the invariant of the type is true after creating the instance (see Section 9 for further details). In the following snippet the default salary of each employee is 100000 forints.[6]

```
type Employee_Type is record
   Name        : Name_Type;
   Address     : Adress_Type;
   Bankaccount : Bankaccount_Type;
   Salary      : Salary_Type := 100000;
end record;
```

Cartesian product types of CLU

The realization of cartesian product type in CLU is unique in several ways. One of the peculiarities is that – similarly to all type composition constructs of the language – there are two cartesian product variants in the language. One of them is the *mutable* `record`, the other is the *immutable* `struct`. The other unique feature is that the selectors are implemented as real subprograms. And due to the difference between the mutable and immutable constructs, the corresponding selectors are different as well.

Selectors in CLU are not functions which return a reference to the corresponding component of the type, but rather they are pairs of subprograms which are automatically defined for each component of the cartesian product. These subprograms represent the two distinct tasks of a selector: reading and writing the value of the corresponding component. In the following code snippet we can see the definition and selectors of a `record` type:

```
EmployeeType = record [ name : NameType,
                        address : AddressType,
                        bankaccount : BankAccountType
                        salary : SalaryType
                      ]
boss : EmployeeType :=
          EmployeeType${ name: "Tim L. Eader",
                         address: "42 Carrot Street",
                         bankaccount: "12345678-87654321",
                         salary: 800000
                       }
```

[6] Forint (HUF) is the currency of Hungary.

```
salary: SalaryType := EmployeeType$get_salary(boss)
EmployeeType$set_salary(boss,salary+100000)
```

The function `get_salary` is used to read the value of the field `salary` and its value is updated using the procedure `set_salary`. These subprograms are automatically generated for the field `salary` and similarly a `get_`⟨*fieldname*⟩ and `set_`⟨*fieldname*⟩ subprogram pair is created for each field. As these subprograms are not qualified by the name of the record, it needs to be passed as parameter to them.

The immutable ***`struct`*** types also have `get` operations for reading the field values, and update procedures in the form of `replace_`⟨*fieldname*⟩. This difference emphasizes the semantic difference between the mutable and immutable types. While the `set_` procedures simply update the corresponding field of the record, the `replace_` operations create new ***`struct`***s which are identical with the old ones in all fields but the one being replaced, which will contain the new value.

Note: CLU supports the more ”traditional” qualified notation of selectors as it is more compact and developers are more familiar with its application. However, that notation is just ”syntactic sugar”, that is, simplified notation of calling the `get_`, `set_` or `replace_` subprograms described above. Qualified references are automatically replaced by the compiler with the corresponding subprogram call.

6.4 Union types

Let us assume now that we have to create an application for handling demographic data. About males we have to store their names, addresses and the length of their military service in years. As we have seen, this can be solved using a cartesian product type. About females we have to store their names, addresses and maiden names. This can be done using *another* cartesian product. However, now we are faced with the problem that the population of a country consists of males *and* females. How can we express that each citizen of the country is either a male or a female? The solution is the *union* composite type.

Union as a type composition method is less widely used, and therefore less widely supported construct. Languages such as BETA, Oberon, Python or the dBase language family do not support it at all. In modern, object-oriented languages its role can be fulfilled using inheritance and polymorphism (see Chapter 10). Therefore, it is not supported in Eiffel, Java or Smalltalk, either.

Given that sometimes we need to use more traditional, non object-oriented languages, it is worth discussing it briefly. Another consideration is that in object-oriented languages, the union of types can be created with arbitrary types, whereas in the inheritance based solution the component types of the union must have a common base class, that is we have to know in advance whether they will have to be components of a union.

6.4.1 Type-value set

The declaration of a union type includes the list of component types and their corresponding selectors. However these declarations show such wide variety that they will be introduced among the language specific features. Nevertheless, in each case the type value set of a union type is the union of the type-value sets of the component types, so the formal definition is as follows:

Let $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$ be types and let $T_1, T_2 \ldots T_n$ denote the corresponding type-value sets. The type $\mathcal{T}$ is the *union* of types $\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n$ if its type-value set is $T = T_1 \cup T_2 \cup \cdots \cup T_n$.

6.4.2 Operations

As in the case of declarations, the set of operations is also very language specific. Selectors can be defined for union types as well, but their function is very different from the selectors of cartesian product types.

Formally, the $s_i : \mathcal{T} \to \mathbb{L}$[7] function is the *selector* function for the component type $\mathcal{T}_i$ if $\forall t \in \mathcal{T} : (s_i(t) = \text{true} \iff t \in \mathcal{T}_i)$. However, very few programming languages implement selectors in this format with this semantics. Due to the wide variety of implementations, it is more correct to speak about "union like" type composition methods, and to examine them one by one.

Tagged and free union

In strongly typed languages the compiler is responsible for verifying that the types of operands in an expression are consistent. These verifications are made during compilation time by considering – among others – the fact that the types of variables are determined at compilation time, and cannot change during execution, and that these variable can contain objects compatible with the type of the variable only. However, this assumption cannot be fulfilled if the language supports union type composition.

If we could create unions of arbitrary types, the compiler would have no means to determine consistency of a variable expression as the validity of the expression would depend on the actual type of the object stored in the variable, which is a runtime property. This is similar to the concept of polymorphism in object-oriented languages but is more unsafe as there is no requirement for the component types to meet – there is no common base class. Hence, the compiler has no information about the set of operations available for the corresponding object.

Though there are many – formally different – solutions for this problem, essentially there are two main approaches only:

- One possible solution is to use runtime type checking. If the language – at runtime – verifies that the *dynamic type* of the object stored in the

[7] $\mathbb{L}$ is the set of boolean values, i.e. $\mathbb{L} = \{\text{true, false}\}$

variable is compatible with the context, we call the union *tagged*. Such a type always has a *tag* which indicates the dynamic type of their value. The price of these solutions is the runtime cost of checking this tag.

- The other alternative is that the language makes no runtime verification, and thus the developer is responsible for ensuring the consistency of usage. This solution focuses on runtime efficiency and simplified code generation. The union construct which makes no runtime type checking is called *free union*.

6.4.3 Union-like composite types

union types of ALGOL 68

Probably the union construct of ALGOL 68 matches the definition of union types best. The following code snippet is the definition of *NumberType* as the union of integer and real types.

```
mode union (int, real) NumberType;
NumberType number;
...
number := 3;
number := 3.14;
...
case number in
    (int i): print(("integer", i)),
    (real r): print(("real", r))
esac
```

Both assignments are valid in the example, the variable *number* can contain integer and real values as well. The type of the current value is determined by using a special variant of **case** statement. This matches the formal definition of selectors.

The value of the *number* variable – unless assigned to another variable of *NumberType* – can only be accessed within the **case** statement. This structure ensures type safety – that is if we want to use the value of *number* as integer, it can only happen in the *int* branch of the **case** structure, which is executed only if the dynamic type of *number* is integer.

union types of C

C took over many things from ALGOL 68; nevertheless, there are significant differences. The following code snippet is a C language example:

```
typedef union {
    MaleType male;
    FemaleType female;
} HumanType;
```

In the example we define *HumanType* as the union of *MaleType* and *FemaleType*. An object of *HumanType* will occupy enough memory to be able to store either an object of *MaleType* or an object of *FemaleType*.

The selectors *male* and *female* do not help in determining the dynamic type of an object of *HumanType*. In C the selectors of **union** types are used to access the memory area of the object as if it were an object of the corresponding component type. In other words they function as interpretation changing type conversions (see Section 5.1.4). The language provides no means to determine the dynamic type of the union, that is, it cannot verify if the actual value is indeed of *MaleType* or of *FemaleType*. This makes the use of **union** types difficult and unsafe and it can easily lead to errors which are hard to discover and fix, such as in the following example:

```
#include <stdio.h>

typedef char NameType[100];

typedef struct {
    NameType name;
    int      was_in_army;
} MaleType;

typedef struct {
    NameType name;
    NameType maiden_name;
} FemaleType;

typedef union {
    MaleType male;
    FemaleType female;
} HumanType;

int main() {
    MaleType m = {"John Doe",1};
    HumanType h;
    h.male = m;
    printf("Name: %s\nMaiden name: %s\n",h.female.name,
           h.female.maiden_name);
    return 0;
}
```

The example can be compiled without a warning, but when executing the program the printed maiden name will be gibberish, the program might even stop with some runtime error.

Neither the language, nor the compiler offer any means to prevent this error. The developer is responsible for ensuring the safe usage of **union** types. The following snippet offers one possible solution:

```
typedef enum { MALE, FEMALE } GenderType;
typedef struct {
    GenderType gender;
    union {
        MaleType male;
        FemaleType female;
    } data;
} HumanType;
```

In the example we manually implement the selectors defined in the formal definition of union types by wrapping the **union** in a **struct** and adding an extra *gender* field. This also means that the **union** is pushed one level lower in the type structure, it becomes accessible via the *data* selector of the top-level **struct**, and the former reference *h.female.name* becomes *h.data.female.name*, etc. Notice that it is still the responsibility of the developer to ensure consistency between *gender* and the actual value of *data*. It is possible to change the gender of a human without updating his or her data. Similarly, the developer is responsible to verify the value of *gender* before trying to access *data*.

As **struct**s can be used as components of **union**s, equality check is not supported on unions. However assignment is allowed as it can be performed simply by copying the corresponding piece of memory byte by byte.

union types of C++

The **union** types of C++ are very similar to C as backward compatibility with C was an important design goal of the language. This essential requirement has lead to some very interesting compromises.

In C++ **struct**s are variants of classes (for the definition of **class** – see Chapter 10), in fact, they are equivalent. Classes of C++ can define *constructor* methods. The compiler is responsible for ensuring the execution of a constructor when an instance of the corresponding class is created. This is a very useful feature, as the constructor is made responsible for the initialization of the object and for ensuring its internal consistency upon creation. However, such strong support of constructors can cause problems when trying to create a union of classes (e.g. **struct**s). In an object of a union type, objects of different classes can be stored overlapping in the same piece of memory. Through the union, the corresponding memory area can be accessed as *any* of the component types. However, running the constructors is not possible, execution of a constructor would overwrite the

results of the previous one. Even if there was a pointed component of the union which could be used during initialization, the language would provide no means to verify the dynamic type of the value of the union and thus the corresponding memory area could be accessed through any of the selectors.

To resolve this conflict C++, has made some compromises. C++ restricts the component types of unions. If a class has a *non-trivial* constructor, it cannot be the component of a **union**. A class has a non-trivial constructor if it is a subclass of a base class which has a non-trivial constructor, or if the definition of the class contains a constructor definition. The trivial constructor of a class makes no initialization. Unions of classes which only have trivial constructors are safe, as the constructors cannot overwrite each others results. Besides, this semantics is compatible with C which has no support for constructors. With the given restriction, **union** of C++ is backwards compatible with C.

This solution is restrictive and a very artificial compromise. Luckily in C++ we have other means to achieve union-like semantics in the form of inheritance and polymorphism (for details see Chapter 10). A base class in a class hierarchy is the union of its subclasses and the *instanceof* operator functions as the selector of the union, as it can be used for runtime verification of the dynamic type.

Tagged unions of CLU

Similarly to ALGOL 68, the union types of CLU are straightforward realizations of the formal definition. In line with the philosophy of CLU, there are two union types, the *mutable* `variant` and the *immutable* `oneof`. Both are *tagged unions*, each component has a tag, which is stored along with the corresponding object value. The selectors of the type can be used to verify the tags and ensure type-safe usage of the unions. The following snippet is the CLU implementation of the usual `HumanType`, omitting the details of the component type:

```
a_woman: FemaleType${name: "Jane Deer", maiden_name: "Jane Doe"}
a_man: MaleType${name: "John Deer"}
HumanType = variant [ male: MaleType, female: FemaleType]
f  : HumanType := HumanType$make_female(a_woman)
m  : HumanType := HumanType$make_male(a_man)
h  : HumanType := n
mm : MaleType
ff : FemaleType
tagcase e
  tag male :
    mm:=HumanType$value_male(h)
    HumanType$change_female(h, a_woman)
  tag female (female_tmp : FemaleType) :
    ff := female_tmp
end
```

```
if HumanType$is_male(h) then
  mm := HumanType$valueue_male(h)
  HumanType$change_female(h,a_woman)
else
  ff := HumanType$valueue_female(h)
end
```

The semantics of the *tagcase* and *if* structures is the same. The *tagcase* statement is very similar to the `case` structure of ALGOL 68 but it contains some simplifications. The use of *branch variable* – as seen in the `female` branch – is optional, we can use the `value_`⟨*tagname*⟩ operations instead, which are automatically defined for each defined tag. The `is_`⟨*tagname*⟩ selectors can be used to identify the actual tag of the *variant*. These operations are generated automatically. There is a third – automatically defined – group of operations for the *variant* structure, these are the `change_`⟨*tagname*⟩ operations which can be used to modify the value of the object. The operations also update the tag accordingly.

In comparison to ALGOL 68, in CLU the relaxed syntax can lead to runtime errors. For example the call below is syntactically correct, but can cause runtime error if the value of `f` is a female.

```
mm := HumanType$valueue_male(f)
```

On the other hand, by making the tags explicit the meaning of the components can be differentiated from their type, and it is possible to use the same type as a component multiple times.

The immutable version of union types is called *oneof*. It only differs from the *variant* in that it does not have the `change_`⟨*tagname*⟩ update operations.

Variant record

The union-like construct of Pascal-type languages is the *variant record*. As the name indicates, it is based on the record construct which is the realization of the cartesian product type in these languages. Logically it is similar to the solution we have used above for preserving type information in the case of C **union**s. The snippet below is a possible implementation of *HumanType* in MODULA-2:

```
TYPE GenderType = (male, female);
TYPE MaleType = RECORD
   name: NameType;
   was_in_army: BOOLEAN;
END;
TYPE FemaleType = RECORD
   name: NameType;
   maiden_name: NameType;
END;
```

```
TYPE HumanType = RECORD
   CASE gender: GenderType OF
   | male:   m: MaleType;
   | female: n: FemaleType;
   END;
END;
```

In variant records there are pointed fields which affect the structure of the record. The example above is the direct translation of the C language example, but even this solution has an important advantage over C. The system – at runtime – verifies the value of the *gender* field. If the field value is *male*, when trying to access *female* part, or vice versa, we will get an error at runtime. This cannot be prevented as the actual type of the variant record may just as well depend on user input, and it is not possible to verify it at compilation time. Nevertheless, the advantage of this solution is that the error is immediate and guaranteed, while in C the error might be missed or its result can manifest later; therefore, finding and fixing the error is much more difficult.

If we do not need to separate *MaleType* and *FemaleType*, we can have a simpler solution as well:

```
TYPE GenderType = (male, female);
TYPE HumanType = RECORD
   name: NameType;
   CASE gender: GenderType OF
   | male:   was_in_army: BOOLEAN;
   | female: maiden_name: NameType;
   END;
END;
```

In this solution the overlapping parts of the *MaleType* and the *FemaleType* are combined, and the variable part of the record only contains the gender-specific fields. This solution does not fit the category of union in the classical sense of notion as the component types of the union are only implicitly given.

From practical point of view, the tag fields of the record are not distinguished from the other fields. However, when the value of the tag field changes, it may mean a change in the record structure. As the representations of the different variants are physically overlapping, changing the tag field might easily leave the record in inconsistent state. It is the responsibility of the developer to ensure the proper update of the variable part when the tag field is modified.

There is only one restriction for the tag field: it must be of a discrete type.

Discriminated records of Ada

The *discriminated records* of Ada are the improved versions of variant records. Discriminated records have parameters – called the *discriminants* of the type

– and the structure of the record can vary depending on the value of the discriminants. The snippet below is a solution for the above problem:

```
Type GenderType is (male, female);
Type HumanType(gender: GenderType) is record
   name: NameType;
   Case gender is
      When male => was_in_army: Boolean;
      When female => maiden_name: NameType;
   End case;
End record;
```

Syntactically discriminated records are very similar to the variant records of Modula-2, but their behavior is significantly different. In line with the declaration above, we need to specify the gender of the person whenever a new object is created, and changing the gender later is not possible. This ensures that no part of the record becomes undefined later. Unfortunately, this way the type only partially functions as a union. The type-value set is union-like, but in each object creation we restrict the type-value set to one of the component types, in fact, we create a subtype of the union implicitly.

What is the use of such a construct then? We can use it as the type of formal parameters of a subprogram, in which case we do not have to specify the value of the discriminant. We can create subprograms that are equally applicable to male and female categories, for example the one that prints the name of the person. We can access the fields of the variable parts as well, but to make sure that the referenced fields indeed exist, we need to verify the value of the discriminant. Otherwise we will get runtime errors.

However, it is possible that we need a real union-like structure, that is we want to create objects for which we can modify the value of the discriminant after creation. This can be achieved by using a *default discriminant*. We modify *HumanType* so that we can create object without specifying the gender:

```
Type GenderType is (male, female);

Type HumanType(gender: GenderType:=male) is record
   name: NameType;
   Case gender is
      When male => wa_in_army: Boolean;
      When female => maiden_name: NameType;
   End case;
End record;

M: HumanType(male);
F: HumanType(female);
H: HumanType;
```

```
Begin
   F := M; -- The compiler warns here that the value of discriminant
   H := M; -- on the right hand side is not compatible and a
   H := F; -- Constraint_Error exception will be raised at runtime.
End;
```

The example above shows three different ways of creating objects of *HumanType*. When creating M, the memory reserved for the variable is suitable for storing the data of a male. Similarly, when creating F, the reserved amount of memory is enough for storing data of a female. Changing discriminant in these cases is not allowed because it might also change the required amount of memory.

When the object H is created, its *gender* field will be *male*, and its structure will be suitable for storing data of a male. However, in this case the compiler allocates enough memory to store the object with any possible value of the discriminant. This allows modification of the discriminant but we must update the whole record to ensure that no field remains uninitialized.

The values of the discriminants cannot only control the set of fields, but also the length of the fields of array types. In the code snippet below we define a *Text* type which is an improvement of the built-in *String* type of Ada. The *String* type is an unconstrained array, its index range is specified when an object is created. However, the length of the text is fixed afterward. In the *Text* type, the length of the text is variable within a range. The actual text is stored in a *String* but we only use the first *length* characters:

```
Type Text(maxlength: Natural) is record
   length : Natural := 0;
   data   : String(1..maxlength) := (others => ' ');
End record;
```

In practice we create a union of *Text*s with maximum length of *1*, *2*, ..., *Natural'LAST*.[8] It is important to note that the language allows the use of a default discriminant here as well. However, the rule is the same – if we create an object without specifying the value of the discriminant – i.e. by using the default value – the compiler will allocate enough memory to store the object with *any* value of the discriminant. In this example, this would mean allocating memory for storing a *String* of length *Natural'LAST* characters, which would likely take more memory than available on the platform.

6.5 Iterated types

Assume that our task is to administrate the students of a class in school. In the previous sections we have seen how we can represent a single person – student –

[8] The attribute *Natural'LAST* denotes the biggest element of the *Natural* type. The actual value is implementation specific but it is at least $2^{31} - 1$.

according to the requirements. Now the task is to represent the whole class as the collection of its students. We need a type composition method which allows the creation of sequences or sets of objects of an existing type. Such constructs are called *iterated* types, and as we will see, they have a great number of variants in the different programming languages.

This type composition method offers the widest variety of all. Unlike unions and cartesian products, this construct is homogeneous, and it uses a single component type.[9] Another difference is that programming languages often support more than one kind of iterated composite types.

The most common variants are the following:

- Array (or vector);
- Multi-dimension array;
- List;
- Hashtable;
- Queue;
- File;
- Set.

Programming languages often provide standard implementations of various container data structures which are similar to these composite type constructs. This chapter *does not* discuss these data structures. These structures are discussed along with standard library functions in Chapter 14. This chapter deals with construction methods that are *part of the core language*, and which enable the creation of composite types. Typically, the implementation of those library types rely on the composition methods discussed in this chapter.

6.6 Array

Assume that we identify the students of our class by consecutive integer numbers, and that we find students by their number. We need an iterated type which enables efficient access to students based on their number. This construct is the *array*. Arrays are probably the most widely supported and the simplest iterated composite type. Almost all programming languages support them in some form.

6.6.1 Type-value set

There are two important components of the definition of an array: the type of elements – values – and the index interval. The former requires simply naming the type of values, and therefore, it is quite similar in all languages; by contrast, the specification of the index interval shows wider variety. In certain implementations

[9] Some associative data structures use two component types, one for the keys and one for the values.

of BASIC it can just as well be omitted, and then it defaults to the $0 \ldots 10$ integer interval. If we need an array of different size, we can indicate that by specifying the upper limit of the index interval. The lower limit is always 0.

In C-like languages[10] arrays are defined by specifying their size, the number of elements they can store. The index interval of an array of n element is the $0 \ldots n-1$ integer interval. The *ClassType* defined in the following snippet can store the data of 42 students:

```
typedef HumanType ClassType[42];
```

In Pascal-like languages, the index interval of an array can be a non-empty range of an arbitrary *discrete* type. The following examples are in Ada, but there is no significant difference in other languages either:

```
Type ClassType is array (1..42) of HumanType;

Type DwarfType is (Bashful,Doc,Dopey,Grumpy,Happy,Sleepy,Sneezy);

Type SevenDwarfs is array (DwarfType) of HumanType;

Type FiveDwarfs is array (DwarfType range Doc..Sleepy) of HumanType;
```

There are programming languages between the two extremes as well. For example in ALGOL 68, the index type must be an integer type, but any interval can be used as index. The array constructs of CLU[11] are also indexed with integer values, but their interval is extensible.

Taking all this into consideration the formal definition of arrays is: let $(\mathcal{H}, \leq)$ be a totally ordered finite or countable set – the *index set* or *index type* – and $\mathcal{T}_0$ an arbitrary type, the *value type*. The type-value set of the $\mathcal{A} = \text{array}(\mathcal{H}, \mathcal{T}_0)$ *array type* is $\{a : [m \ldots n] \to \mathcal{T}_0 | m, n \in \mathcal{H} \wedge m \leq n\}$. In other words, an $a \in \mathcal{V}$ array is a mapping of an index interval to the value type.

Notice that the index interval can be specified in three different ways depending on the languages. In Pascal or C/C++ the index interval is static, defined in the declaration of the type. In Ada we can create unconstrained array types where the index interval is specified only when an array object is created. CLU and Eiffel are even more flexible, they allow the dynamic modification of the index interval.

6.6.2 Operation

The set of operations of array types are largely the same in all languages; there are only a few specifics.

[10] Just like in case of Pascal, many languages used C as the basis for their syntax definition. Such languages include Java or C++.

[11] In CLU there are two array constructs the mutable `array` and the immutable `sequence`.

Selection

The most important operation of arrays is *selection*. The natural definition of selection suggests that for an arbitrary $a \in \mathcal{A}$ and $i \in \mathcal{H}$ selection is the evaluation of $a(i)$. Similarly to the cartesian products, this is a bit of oversimplification. In most programming languages, $a(i)$ is an L-value, the type of $a(i)$ is not $\mathcal{T}_0$ but $\mathcal{T}_0$ *reference* (see Section 5.6 for details). Most programming languages do not make this distinction in the specification of the language: they treat $a(i)$ as of type $\mathcal{T}_0$, yet allow using it on the left-hand side of assignments. This is so because the full formal distinction would make the definition of the language very complex. One of the exceptions is the specification of ALGOL 68. In that language the type of variables is always a reference to some type $\mathcal{T}$ while the type of objects is $\mathcal{T}$. Assignment is always defined between a reference type – L-value – and the referenced type – R-value – (see [Tan76] for details).

Notation of selection is quite uniform in the different languages, i.e. usually the index is specified after the name of the array in square brackets (`a[i]`). One of the few exceptions is Ada which uses parentheses (`a(i)`).

Assignment

Similarly to cartesian product types, assignment is much more controversial operation. Many languages do not support it at all, claiming that an array can be copied to another array by a simple loop. Or as another alternative, sometimes the language offers an implementation as part of its standard libraries. On the other hand, Pascal-like languages typically provide assignment support between arrays of equal type and length.

6.6.3 Language specific features

While the basic concepts are similar, programming languages provide many interesting features for their respective array types. This section provides an overview of the most interesting ones.

Arrays in C and C++

Pointers and arrays are closely related in C – an array is represented as the pointer to its first element, and often they can be used interchangeably. For example, the operation of selection is realized by replacing the expression `a[i]` with the expression `*(a+i)` where `+` is the addition operator between integers and typed pointers[12] and `*` is the dereference operator. This equivalence has two very interesting implications. Firstly, the selection operator can be applied to any typed pointer (in the expression above a is used as a pointer). Secondly, the

[12] C provides a very powerful pointer arithmetic toolset which is described in details in Section 5.6.

selection operation in C is "commutative", `a[i]` and `i[a]` designate the same element, because the `+` operator of pointer arithmetics is commutative.

Given this relationship between pointers and arrays, we can create generic purpose subprograms which can handle arrays of arbitrary length – similarly to unconstrained arrays of Ada – because in practice, the subprogram is always invoked with the pointer to the first element. An important difference to Ada is that the array type does not know its length, thus, we either need to specify the length as a separate argument, or we need to use some special value to designate the last element of the array.

C and therefore C++ treats selection as an operator. C++ supports operator overloading which applies to the selection operator (`[]`) as well. By using this property, we can define the selection operator for our own types, e.g. for different container types (queues, hashtables, etc.).

6.6.4 Arrays in Java

Arrays of Java closely resemble arrays of C and C++ *syntactically*. However, there are subtle differences in their semantics. It is not possible to declare *named* array types in Java. For each type T there is an implicit array type T`[]`. This is an unconstrained array type, as what we have in Ada. Arrays of Java are always indexed by integers, the lower limit of the index interval is necessarily 0. The length of an array is determined when the corresponding array object is created, and an array of length n is indexed by the integer interval $0 \ldots n-1$. Similarly to Ada – but quite unlike C/C++ – arrays know their number of elements. For an array a it is available as $a.length$.

Arrays in Ada

When declaring an array type in Ada we do not necessarily have to define the exact index interval, it is enough to specify the index type. Any discrete type can be used to index arrays. If only the index type is specified the array type is called *unconstrained*. For such types the index interval needs to be specified when an array object is created. This always generates an *implicit, constrained subtype* (see Section 5.8.1) of the unconstrained array type.

The unconstrained array type cannot be used in variable declarations, but it can be used as the type of formal parameters enabling subprograms to handle any instance of the unconstrained type in a generic fashion.

```
type RealArray is array (Integer range <>) of Float;

function MaxIndex(T: in RealArray) return Integer is
MINDEX: Integer := T'First;
begin
    for I in T'Range loop
        if T(I) > T(MINDEX) then
```

```
            MINDEX := I;
        end if;
    end loop;
    return MINDEX;
end Max;

A: RealArray(1..4) := (5.0, 6.7, 2.11, 1.0123);
MAX: Integer := MaxIndex(A);
```

In the example above, the variable *MINDEX* is initialized as *T'First*. This expression is a special operation – *attribute* – of the array type of Ada. It specifies the lower bound of the index interval. Similarly, the attribute *T'Last* specifies the upper bound, while the attribute *T'Range* designates the index interval – *range T'First..T'Last* – itself. Additionally, we may use the *T'Length* attribute which is the number of elements of array *T*. By using these attributes, we can write generic purpose subprograms.

Ada allows assignment between arrays if they are subtypes of the same array type and they have the same number of elements. This property is particularly useful when combined with slicing (see below).

Arrays can be initialized as seen in the code snippets above and below. *Array aggregates* are literals of type array. In *positional* array aggregates elements are assigned to indexes based on their position (order). The first element will be assigned to the lowest index, the second one to the second lowest index, etc. In *named* array aggregates we can explicitly specify which element is assigned to which index and we can use the **others** notation for the remaining, unspecified indexes. Since the release of Ada 95, positional and named notations can be mixed within the same aggregate, starting with positional notation and continuing with named one. For example, see array *C* in the code snippet below.

There are two additional operations available for arrays in Ada: *slicing* and *concatenation*. By using slicing we can create a subarray of an array by specifying its index subinterval. The other operation is concatenation which can be used to join two arrays if they are of the same base type. The following code snippet demonstrates the usage of aggregates, slicing and concatenation:

```
type IntegerArray is array (Integer range <>) of Integer;
A: IntegerArray(11..31) := (11..20 => 5, others => 42);
B: IntegerArray(101..142) := (1, 2, 3, 4, others => 0);
C: IntegerArray(1..42) := (1 => 7, 5|6|11|41 => 6, others =>0);
C := A & B(111..131); -- The index intervals do not have to match
```

Array types of CLU

As all other composite type construct, CLU has two variants of array types: the mutable `array` and the immutable `sequence`.

The index boundaries of `arrays` are modifiable. We can create an *`array`* as empty – in this case the lower bound of the index interval is 1 while the upper bound is 0 – or by enumerating its element in some sort of aggregate notation. In the latter case we can also specify the lower bound of the index interval. There are no multi-dimensional arrays, but we can create arrays of arrays.

```
IntegerArray   = array[ int ]
ia1 : IntegerArray := IntegerArray$new()
ia2 : IntegerArray := IntegerArray$[3:42, 21, 14, 7, 6, 3, 2, 1]
i   : int := IntegerArray$high(ia2)
while i >= IntegerArray$ℓow(ia2) do
   e : int := IntegerArray$fetch(ia2, i)
   IntegerArray$addℓ(ia1, e)
   i := i - 1
end
```

The array `ia1` is created empty by the constructor `new` while `ia2` is initialized with the positive dividers of 42 in descending order. The code copies the content of `ia2` to `ia1` in a reverse order. The functions `high` and `low` are used to query the index boundaries of `ia2`. CLU also offers a function `size` which is the number of elements in the array. All three functions return integer (*`int`*) values. The arrays of CLU are always indexed with integers; therefore, in their declaration we only need to specify the type of the elements. Notice the array aggregate used to initialize `ia2`. Syntactically it is a constructor call. The number before the colon indicates the start index. The elements are listed after the colon; there must be at least one element listed when this form is used.

For selection, we use the operation `fetch` in the example. Its opposite is `store` which is used to update the array at a specific index. CLU supports the short `ia2[i]` form – syntactic sugar – as well, which translates to either a `fetch` or a `store`, depending on whether it is on the right or left side of an assignment.

The `addl` operation is used to extend the index interval of `ia1` at the lower end and insert the specified element at the new index. Symmetrically there is an operation `addh` to extend the array at the upper end, and there are operations `reml` and `remh` which retrieve and remove to element at the lower (upper) end of the array, reducing the index interval.

The external behavior of *`sequence`* types is very similar to `arrays`. The set of operations is mostly the same, though the update operations do not modify the *`sequence`* object, but create a new instance as the result of the operation. Other differences are that the operation `store` is renamed to `replace` and that the lower index bound cannot be specified in the constructor (it is always 1). There is an additional operation for concatenating two *`sequence`*s called `concat` or `||`. Such an operation is not needed for the mutable *`array`*; it can be implemented by a simple loop moving the elements from one *`array`* to the other one by one. The same implementation on the immutable *`sequence`*s would mean the creation of a large number of temporary objects and a lot of copying into them.

6.6.5 Generalization – multi-dimensional arrays

Assume that we need to create a chess game. We need to store a chessboard and for each square of the board we need to store the piece that occupies that place, if any. The squares of the chessboard are identified by their coordinates, that is by the coordinates of the rows and columns. The best construct to represent the chessboard in the computer memory is a *multi-dimensional array*.

A *multi-dimensional array* is the generalization of the composite type array. Let $(\mathcal{H}_1, \leq_1), \ldots (\mathcal{H}_d, \leq_d)$ be ordered finite or countable sets, the *index types* and $\mathcal{T}_0$ an arbitrary *value type*. The type value set of the $\mathcal{A} = \text{array}(\mathcal{H}_1, \ldots, \mathcal{H}_d, \mathcal{T}_0)$ *(multi-dimensional) array* type is:
$\{a : [m_1 \ldots n_1] \times \ldots \times [m_d \ldots n_d] \rightarrow \mathcal{T}_0 | \forall i \in [1 \ldots d] : (m_i, n_i \in \mathcal{H}_i \wedge m_i \leq_i n_i)\}$.

Often, programming languages have no explicit support for multi-dimensional arrays as they can be represented as arrays of arrays, provided that the language allows the use of arrays as value type. The following code snippet illustrates how we can create multi-dimensional arrays in C using this solution:

```
int a[42][42][42][42];
int **p;
int l = 0;
p = a[21][21];
l = (p[21][21] == a[21][21][21][21]);
```

As the example shows, we can omit indexes from right to left during selection. The result is an array with lower number of dimensions showing that the created construct is indeed an array of arrays embedded in the required depth. Notice also the symmetry with pointers, each new dimension is expressed by adding a new level of reference (see Section 5.6).

Other languages support multi-dimensional arrays but may impose a limitation on the number of dimensions. For example FORTRAN 77 supports multi-dimensional arrays but it requires implementations to support *at least three dimensions* only. FORTRAN 90 kept this option for limited dimensions, but increased the minimum required dimensions to seven.

In the languages which support multi-dimensional arrays there is a syntactic difference between a multi-dimensional array and an array of arrays as illustrated by the following code snippet written in Ada:

```
Type Vector is array(1..10) of Float;
Type Matrix is array(1..10, 1..10) of Float;
Type NonMatrix is array(1..10) of Vector;
V  : Vector;
M  : Matrix;
NM : NonMatrix;
M(3,4) := NM(3)(4);
NM(2) := V;
```

Notice the differences between the usage of *Matrix* and *NonMatrix* types. The indexes of *Matrix* are written within the same pair of parentheses and indexes cannot be omitted. In the case of *NonMatrix*, the different indexes are written in separate pairs of parentheses, and they can be omitted from right to left.

Representation of arrays

Multi-dimensional arrays are multi-dimensional while computers typically use a linear addressing model. Therefore, the elements of the array need to be put in sequential order. Theoretically, arbitrary ordering could be used as long as we are able to determine the address of the elements based on their indexes, but this computation needs to be very efficient as it is used in every single selection operation.

In practice two methods of ordering are used: *row-major order* and *column-major order*. The idea behind the two is very similar, functionally there is no difference between them. Some languages – for example FORTRAN, MATLAB or R – use column-major representation, while PL/I or Python use row-major ordering. The chosen representation might have an impact on performance when traversing the array as loading elements from close memory locations is usually faster due to caching used in the CPU. Otherwise the two representations are equivalent.

To demonstrate how the addressing works in the case of *row-major ordering*, consider the following example. Let A be an k-dimensional array which has $n_1 \times n_2 \times \cdots \times n_k$ elements. The index intervals of the array are $[0 \dots n_i - 1]$ for $i \in [1 \dots k]$. Let $p_k = 1$ and for $j \in [1 \dots k-1]$:

$$p_j = \prod_{i=j+1}^{k} n_{i-1}$$

The *base address* of the array A is the address of element $A(0, 0, \dots, 0)$ which is denoted by A_{base} and let s be the size of the elements of A. With this notation the addres of the element $T(i_1, i_2, \dots, i_k)$ is

$$A_{base} + \Big(\sum_{j=1}^{k} i_j p_j\Big) s,$$

6.7 Sets

Assume that the students in our hypothetical class can join a number of optional courses and we need to store and decide if a particular student attends a particular course. Therefore, for each student we need to store the *set* of courses he or she attends.

6.7.1 Type-value set

Let $\mathcal{T}_0$ denote an arbitrary *element type* and T_0 be its type-value set. The type-value set of the $\mathcal{S} = \text{set}(\mathcal{T}_0)$ *set* type is the 2^{T_0} *powerset*, an $s \in \mathcal{S}$ set is a subset of the type-value set of the element type ($s \subseteq T_0$).

Sets are much more uncommon composition type than the previously discussed arrays. Even if the language supports sets, it has strict restrictions for the element type. It is typical to restrict it to discrete types only, often limiting the size of its type-value set too. For example, in Pascal the element type is restricted to enumeration types of maximum 256 elements, while in Modula-2, it is limited to discrete types represented as one machine word at maximum. The reasons for these restrictions are rooted in the representation of the type.

Set types are usually represented as bit vectors, which consist of as many bits as the size of the type-value set of the element type. Each bit in the vector corresponds to an element of the type-value set. The bitvalue 1 means that the corresponding element is in the set, while the value 0 means it is not. The advantage of this representation is that the most common set operations – e.g. union, intersection, complement – can be implemented by using simple bit arithmetics operations – i.e. or, and, negation –, which have efficient realizations at the processor level. The drawback of this representation is that it can grow very big if the type-value set of the element type is big.

Due to these restrictions, the applicability of the sets of this form is quite limited. Therefore, most programming languages do not offer such a composite type, but offer a set data structure in their standard libraries instead. Typical examples are the *Set* implementations of the Java Collection Framework or the set templates in the Standard Template Library of C++, see Chapter 14.

6.7.2 Operations

Assignment and equality check

As a result of all the restrictions and specific representation, implementing assignment and equality check is simple, and therefore permitted in all realizations.

Set operations

The usual set operations – e.g. union, intersection, difference – are usually supported to a different degree in the various languages. However, support for complements is rare because it might create very big sets.

Membership

One of the most fundamental operations of sets is *membership*, which decides if a particular value is element of the given set. Interestingly this operation may be present in languages which do not provide a set composite type. For example,

in Ada the operator **in** can be used to check if a value belongs to a specified interval, but the language itself does not support sets.

6.8 Other iterated types

This section provides an overview of some other specific iterated types.

Hashtables in Perl

Assume that we want to change our student management system and wish to identify students by their names instead of their indexes. We could continue using an array and we could now consider creating our own selector or *find* operation which could in turn be used to access the data of a student based on his or her name. However, this solution would have an impact on our system's performance. Finding a student by his/her index in an array takes $O(1)$[13] time, while finding them by name takes $O(n)$.[14] *Hashtables* are associative data structures which store *key-value pairs* and support efficient retrieval of data based on its key.

The operations of hashtables are similar to the operations of arrays with some differences. For example, the selection operation takes the key as an argument rather than an index. As the set of keys is not necessarily ordered, there are no "key intervals"; therefore, slicing is not supported. To improve access to the stored data, most hashtable realizations provide support for enumerating the keys used in the table.

Programming languages usually support hashtables as part of their standard libraries, such as *Hashtable* class in Java. However, in Perl the language itself offers a hashtable composite type. The language restricts the keys and values to scalar types but treats strings as scalar. The following simple example (taken from [Til96]) counts the frequency of words in a text file.

```
#!/usr/bin/perl

while ( $inputℓine = <STDIN> ) {
   while ( $inputℓine =~ /\b[A-Za-z]\S+/g ) {
      $w = $&;
      $w =~ s/[;.,:-]$//;
      $words{$w} += 1;
   }
}
```

[13] The *big O notation* is the usual measure of complexity. $O(1)$ time means that the time needed to perform the task does not depend on the number of elements in the array. $O(n)$ complexity means there is a linear relation between the time needed to complete the task and the size of input (for example the number of elements in the array).

[14] If the array is sorted by the names, we can find the student in $O(\log n)$ but then adding a new student to a class will be more expensive.

```
print("Words and their frequencies:");
foreach $w (keys(%words)) {
   print("$w: $words{$w}\n");
}
```

Curly braces – { and } – are the selection operator. Notice that when selection is used, the name of the hashtable is prefixed with the dollar sign ($). In Perl this indicates that the following expression has scalar value. However, when referencing the hashtable as a whole, the name of the table is prefixed with a percent sign (%). If no value was assigned to a particular key, it has a special "empty" value. Values are set or updated by a simple assignment. An existing value is removed using the `delete` operation.

Perl allows assignment of hashtables which means copying the whole content of the table. Another important operation is `keys` which provides the list of keys used in the hashtable. This can be used to enumerate the data stored in the table, e.g. when printing the result in the example above.

6.9 Summary

This section summarizes the most important issues in the area of type composition that we need to answer when studying a new programming language. We also give possible answers or solutions to these questions by looking at specific examples from the languages listed in this book. However, answering these questions is not always easy, the familiar patterns do not necessarily apply to new languages. Most probably the list of questions below is not complete – the existing languages are developing and new constructs are being invented. However, these questions are a good starting point and help to shed light on the most important issues in type composition.

- *What type equivalence does the language use?*
  We have seen examples of strongly typed languages like Ada, which use a very strict equivalence definition – that is, name equivalence. Modula-3 represented the other extreme which requires structural equivalence only.
- *Does the language support the creation of immutable types?*
  In Java we can create immutable types by using the means provided by the language, but there is no explcit immutable type construct. In CLU all composite types have a mutable and an immutable variant. For the efficient use of immutable objects the language must support garbage collection and objects should be stored by reference.
- *Does the language have cartesian product composite type?*
  There are languages such as FORTRAN which do not support the cartesian product composite type. Other languages – e.g. CLU – offer multiple solutions. If the language does not support cartesian products, does it

offer any alternative? For example, Perl has no cartesian product type, but hashtables are a good substitute for them.

- *How do the selectors of cartesian product types work?*
  The most common form is qualifying the selector with the cartesian product object in the form $t.s_i$, which designates a reference of the corresponding component so that it can be used as an L-value as well. However, for example in CLU, selectors are realized as a getter-setter pair of subprograms.
- *Does the language support assignment of cartesian product objects?*
  In most languages, the assignment of cartesian products is allowed, the default semantics corresponds to a component by component assignment of the values. However, in some languages – e.g. in the case of the limited types of Ada – it is possible to restrict assignment.
- *Can we influence how assignment works?*
  In C the answer is a simple no. In Ada we have the opportunity to restrict assignment on certain types using the **limited** keyword. C++ offers the most sophisticated solution where we can define the semantics of assignment by overloading the assignment operator.
- *Is there equality check defined for cartesian products?*
  Most modern languages allow equality check but for example C does not.
- *Can we influence the definition of equality?*
  As equality is an operator in almost all languages, if the language supports operator overloading, it applies to the equality operator as well. Java is an interesting case where the equality of references is an operator which cannot be overloaded, whereas the equality of objects is checked using the *equals* method of the class, which can be overridden by the developer.
- *Does the language have a* **WITH***-like construct?*
  We have seen this operation in the Pascal-like languages. It can be used to simplify the complex qualified expressions.
- *What means does the language offer to influence the physical representation of cartesian product types?*
  Some languages offer no means at all, whereas other languages offer compilation directives, pragmas to specify alignment of components, to control the use of padding and other physical aspects of the representation. In Ada we can specify the details of representation bit by bit by using the representation clauses.
- *Can we specify the default value of components of the cartesian product?*
  We have seen this feature in Ada where the declaration of a **record** type can include default values for the fields.
- *Does the language have cartesian product literals?*
  We have seen aggregate notation of cartesian products in multiple languages. It is often used to initialize variables or defined constants. Naturally, this assumes support for assignment.

- *Does the language support union type composition?*
  Support for union is less frequent in languages than for cartesian products. If the language does not support it, it is worth to check what alternatives it offers. We have seen how inheritance and polymorphism can provide similar functionality in object-oriented languages.
- *Free or tagged union?*
  We have seen languages such as C where the union does not have a selector as defined in the type construct, and the language provides no means to verify the dynamic type of the union object. On the other hand, the content of a **union** of ALGOL 68 can only be accessed in a *tagcase* structure which ensures type safe usage.
- *Does the language support arrays?*
  Though in most languages the answer is yes, there are special cases like SuperNova which does not offer this construct.
- *What are the allowed element types of arrays?*
  There is large variety in this area. Arrays of FORTRAN can only store scalar values. Perl also restricts the element type to scalars, though in the case of Perl this includes strings as well. In Ada the element type cannot be an unconstrained type. Can arrays be elements of arrays?
- *What types can be used for indexing?*
  Arrays of C are indexed by integers from 0. CLU allows arbitrary index intervals, but the index type must be an integer. In Pascal-like languages typically any discrete type can be used for indexing.
- *How flexible is the index interval?*
  In most languages, the index interval is specified in the declaration of the array type. In C-like languages only the number of elements can be specified, and indexing always starts at 0. In Java, array types have no fixed length, it can be specified when the corresponding instance is created. Similarly, Ada allows the declaration of unconstrained array types where the index interval can be arbitrary, and it is specified when the instance is created. The arrays of Eiffel or CLU are even more flexible, the index interval can be modified dynamically, arrays can expand or shrink as needed.
- *Does the array contain information about its index interval?*
  Arrays of C or C++ do not contain any structure information, they are barely a pointer to the first element of the array. In these languages, it is the responsibility of the developer to make the required information available if needed. In Pascal-like languages arrays usually ”know” their size and index interval. Depending on compilation options, the selectors might do range checks at runtime to ensure that the specified index is in the permissible interval. Many languages – for example Ada, Java or CLU – offer some means for querying the index boundaries.

- *Does the language support assignment of arrays?*
  Often the answer is no. C-like languages define an assignment operator for array types but as arrays are represented by the pointer of their first element, this does not mean copying the array, but rather sharing the reference. Pascal-like languages usually allow assignment, but only for named array types. Most languages that use structural type equivalence allow assignment between arbitrary array types as long as the number of elements is the same.

- *Can we use unconstrained arrays as a formal parameter of subprograms?*
  In languages which have unconstrained array type – fe.g. Ada – it can be used as the type of formal parameters of subprograms. Using these types makes the subprogram more generic. We can have similar results in other languages as well. For example in C or C++, arrays are simple pointers to the first element of the array, which also enables creating subprograms that work on arrays of arbitrary length.

- *Does the language support multi-dimensional arrays?*
  Many languages support one dimensional arrays – vectors – only. However, most of them allow using vectors as element types of vectors. Other languages support real multi-dimensional arrays, though they might impose limitations on the number of dimensions (e.g. FORTRAN).

- *Does the language support the set composite type?*
  Set is a much more infrequent composite type than arrays, cartesian products or even unions, though some languages like Pascal or Modula-2 support it. Often sets are implemented as a standard container data type in some standard library of the language.

- *What are the restrictions of the element type of sets?*
  Languages that support the set composite type impose strict limitations on the element type. They usually require discrete or enumeration types and often limit the size of the type-value set as well.

- *Does the language have other composite types?*
  Some languages support other type composition methods. We have seen examples like the lists and hashtables of Perl. Other languages may also have constructs that do not fit in with the archetypical composites described in this chapter. When encountering a new composite, we should try to determine its type-value set and the set of its operation as seen in the previous sections. However, we have to pay close attention to the type composition methods of the language which need to be distinguished from the container data structures that are usually provided as part of the standard libraries of the language.

6.10 Exercises

Exercise 6.1. What are the advantages and disadvantages of immutable types?

Exercise 6.2. What does structural type equivalence mean?

Exercise 6.3. What are the advantages of unconstrained array types?

Exercise 6.4. Specify the formulas to calculate the address of elements of quadratic matrices when using row-major, column-major or spiralic[15] order of the elements.

Exercise 6.5. Explain the challenges of the complement operation in the case of set types!

Exercise 6.6. Create a quadratic matrix type in Ada. The size of the matrix should be the parameter of the type. How can you guarantee that the matrix is quadratic?

6.11 Useful tips

Tip 6.1. Consider thread safety and the impact on memory usage.

Tip 6.2. This is the weakest form of type equivalence.

Tip 6.3. Consider what kind of generalizations the use of unconstrained arrays enable!

Tip 6.4. Consider the addressing of multidimensional arrays:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Row major

0	5	10	15	20
1	6	11	16	21
2	7	12	17	22
3	8	13	18	23
4	9	14	19	24

Column major

0	1	2	3	4
15	16	17	18	5
14	23	24	19	6
13	22	21	20	7
12	11	10	9	8

Spiralic

Tip 6.5. Consider the complement set of $\{apple, peach\}$ or $\{0, 1, 2, 42\}$!

Tip 6.6. The index boundaries of unconstrained two-dimensional arrays can be specified independently. The array should be wrapped in a discriminated record which has a single discriminant, the size of the quadratic matrix. The actual array is then defined using the discriminant. The drawback of this solution is that getter and setter subprograms need to be created for accessing and setting the elements of the matrix.

[15] Spiralic order is highly impractical and it is not used in any programming language. The goal of this exercise is to demonstrate that *any* address function could be used.

6.12 Solutions

Solution 6.1. The biggest advantage of immutable types is that they are safe for concurrent or even parallel use. As the internal state of the object cannot change after initialization, there is no risk of conflicting concurrent modifications. The drawback of the use of immutable types is that as each modification will create a new copy of the complete object. This adds to the memory footprint of the application and to the execution time as well. There are many possibilities to optimize this copying process but still the overhead can be prohibitively large for certain complex types.

Solution 6.2. Structure equivalence is the weakest form of type equivalence, which considers two types equivalent when their structure is isomorphic, regardless of their names. Such equivalence is used for example in Modula-3. Even structural equivalence can have different levels, e.g. whether the name of fields in a cartesian product type are considered part of the structure. In the case of Modula-3 the following definition is used:

Two types are the same if their definitions become the same when expanded; that is, when all constant expressions are replaced by their values and all type names are replaced by their definitions.[16]

According to this definition, the following types are equivalent:

```
TYPE Coordinates = RECORD
   X : INTEGER;
   Y : INTEGER;
END;

TYPE RationalNumber = RECORD
   N : INTEGER;
   D : INTEGER;
END;
```

Solution 6.3. Unconstrained array types allow the creation of more flexibly usable subprograms. Sorting, finding or aggregating the elements of the array and other often used algorithms can be implemented in a more generic fashion.

The following Ada generic implements conditional maximum search, that is it select the maximum element of an array which satisfies a given condition (e.g. find the greatest odd value). The parameter of the generic is an arbitrary array type. In the generic the array type is treated as unconstrained. Notice the use of array attributes!

[16] Source: Modula-3 language definition
`http://www.cs.purdue.edu/homes/hosking/m3/reference/`

```
generic
   type Element is limited private;
   type Index is (<>);
   type Vector is array(Index range <>) of Element;
   with function Cond(E: Element) return Boolean;
   with function "<"(A,B: Element) return Boolean is <>;
procedure CondMax(V: in Vector; FOUND: out Boolean; I: out Index);

procedure CondMax(V: in Vector; FOUND: out Boolean; I: out Index) is
begin
   FOUND:=False;
   I:=V'First;
   for J in V'Range loop
      if Cond(V(J)) then
         if (not FOUND) or else (FOUND and then V(I)<V(J)) then
            I:=J;
            FOUND:=True;
         end if;
      end if;
   end loop;
end CondMax;
```

Solution 6.4. Let N denote the size of the quadratic matrix and A denote the base address of matrix A. For simplicity, let us assume that the matrix is zero-indexes (i.e. both row and column indices are from the $[0..N-1]$ interval). The address function of the matrix

- Row-major order: $A(i,j) \rightarrow A + i * N + j$
- Column-major order: $A(i,j) \rightarrow A + j * N + i$
- Spiralic order (assuming clockwise order starting from $A(0,0)$:
  $A(i,j) = A + A'(i,j,N)$ where
  $A'(0,0,1) = 0$
  $A'(0,j,N) = j$ where $0 <= j < N$ and $N > 1$
  $A'(i,N,N) = N - 1 + i$ where $1 <= i < N$ and $N > 1$
  $A'(N,j,N) = 3 * N - 3 - j$ where $0 <= j < N - 1$ and $N > 1$
  $A'(i,0,N) = 4 * N - 4 - i$ where $1 <= i < N - 1$ and $N > 1$
  $A'(i,j,N) = 4 * N - 4 + A'(i-1,j-1,N-2)$ where $N > 2$ and $1 <= i <= N - 2$ and $1 <= j <= N - 2$

Solution 6.5. The biggest problem with complement operation is that the resulting set is often infinite. Consider for example a set of strings. There are infinite possible strings but even if the length of possible elements is limited, the size of the complementer set is prohibitively large. For this reason complement operation is not supported for set types except for languages where the supported base type is highly restricted (for example in Pascal it must be discrete type with maximum 256 elements).

Solution 6.6.

```
Package QuadraticMatrix is
    -- QMatrix is defined as discriminated record to ensure that the matrix is
    -- quadratic
     Type QMatrix(N:Positive) is private;

     -- Unfortunately we need to create getters and setters for the matrix
     -- elements.
     Procedure Set(M: in out QMatrix; I,J: Natural, VALUE: Float);
     Function Get(M: QMatrix; I,J: Natural) return Float;

     Function "+"(A,B: QMatrix) return QMatrix;
     Function "-"(A,B: QMatrix) return QMatrix;

     Function "*"(A,B: QMatrix) return QMatrix;

     -- Multiplication by a constant. F*M and M*F need to be declared separately
     Function "*"(F: Float; M: QMatrix) return QMatrix;
     Function "*"(M: QMatrix; F: Float) return QMatrix;

Private
    Type Matrix is array(Integer range <>, Integer range <>) of Float;

    Type QMatrix(N: Positive) is record
        M : Matrix(0..N-1, 0..N-1) := (others=>(others=>0.0));
    End record;
End QuadraticMatrix;

Package body QuadraticMatrix is
    Procedure Set(M: in out QMatrix; I,J: Natural; VALUE: Float) is
    Begin
       M.M(I,J) := VALUE;
    End Set;

    Function Get(M: QMatrix; I,J: Natural) return Float is
    Begin
        Return M.M(I,J);
    End Get;

    Function "+"(A,B: QMatrix) return QMatrix is
    R : QMatrix(A.N);
    Begin
        If A.N /= B.N then
            Raise CONSTRAINT_ERROR;
        End if;
        For I in R.M'Range(1) loop
            For J in R.M'Range(2) loop
                R.M(I,J) := A.M(I,J) + B.M(I,J);
            End loop;
        End loop;
        Return R;
    End "+";

    Function "-"(A,B: QMatrix) return QMatrix is
    Begin
        Return A + (-1.0*B);
    End "-";

    Function "*"(A,B: QMatrix) return QMatrix is
    R: QMatrix(A.N);
    Begin
        If A.N /= B.N then
            Raise CONSTRAINT_ERROR;
        End if;
        For I in R.M'Range(1) loop
            For J in R.M'Range(2) loop
```

```
                For K in A.M'Range(2) loop
                     R.M(I,J) := R.M(I,J) + A.M(I,K) * B.M(K,J);
                End loop;
            End loop;
        End loop;
        Return R;
    End "*";

    Function "*"(F: Float; M: QMatrix) return QMatrix is
    R : QMatrix(M.N);
    Begin
       For I in R.M'Range(1) loop
            For J in R.M'Range(2) loop
                R.M(I,J) := F * M.M(I,J);
            End loop;
        End loop;
        Return R;
    End "*";

    Function "*"(M: QMatrix; F: Float) return QMatrix is
    Begin
       Return F*M;
    End "*";

End QuadraticMatrix;
```

7 Subprograms

In this chapter we discuss the subprograms – the programming language features for implementing control abstraction. We will learn why these subprograms are useful? What differences are between procedures and functions? The related notions such as the specification, definition and calling of subprograms will be described. This chapter also reviews the parameters for subprograms, and which techniques can be applied to pass them. Recursive subprograms will also be dealt with, like how much they differ from macros, co-routines and iterators. We examine how subprograms fit the frame defined by the structure of the program, and how they can be nested and what scope rules apply to them. At the end the most important language elements for subprogram implementations will be shown.

The most important task of the programmers is to design quality software. Data abstraction skills help finding good solutions, but for program language implementations also proper tools are needed. Subprograms offer support for control abstraction possibilities. "Subprograms have already existed before the first programming languages" - writes Ravi Sethi in the introduction to his chapter about subprograms [Set96], and explains: "there were attempts already since 1944 to make the writing of programs easier and faster by copying code snippets from each others note-books and pasting it into own programs."

Sebesta mentions an even more preliminary example: the first programmable computer, the Analytical Engine built in 1840 by Charles Babbage was also capable of reusing a batch of control cards at more locations during program execution. The subprogram is such a language structure which allows to map a name to a code snippet for easy reuse later at will. Execution of the code snippet, or *subprogram calling* is initiated by specifying the given name (and possible actual parameters). By using parameters, subprograms can easily implement parametrized computing. In this case the named code snippet, the subprogram can rely on data specified as actual parameters. The notion of subprograms is very similar to that of the function already used by mathematicians for centuries.

The following Ada program snippet shows a function subprogram which computes the maximum of two integers. This code can be called from anywhere where such functionality is needed, and permits the easy reuse by parametrization.

```
function Max (A, B: Integer) return Integer is
begin
    if A > B then return A; else return B; end if;
end Max;
```

Calling this subprogram can take place in the following way. Assuming K, L and M being integer variables:

```
K := 3; L := 2; M := 5;
K := Max(K,Max(L,M));
```

7.1 The effect of subprograms on software quality

The subprogram is a construct to implement *control abstraction*. Beside the fact that it can be used to pass parameters to sequences of program statements and reuse them multiple times, it also allows to handle logically independent code snippets and well defined calculations as a separate unit, meanwhile hiding the concrete program statements sequence. That is why the subprogram is a crucial language element not only for *reusability*, but also for other software quality aspects such as *readability*, *changeability* and *maintainability*. The importance of these software quality aspects can be acknowledged if we think of the life cycle of software products where the production itself is usually just a small proportion of the whole process.

Reusability

The role of the subprograms from this aspect is obvious: the same code snippet can be reused with different parameters without the need to invest more in its production (design, implementation, compilation, correctness proving and checking, testing). This makes software development more economical. Besides this, the endproduct usually has a smaller size, as well.

Readability

A properly chosen name can be more expressive to the reader of the program source than the implementing code, since the subprogram usually implements a well defined, logically separated subtask. If the call of a subprogram appears multiple times in the program source, its function does not need to be understood again and again, only once, at its first definition. (In fact, its goal could be deduced just from its name, or documentation, and the reader does not even need to look at the implementation.).

The arrangement of the program source into subprograms can positively influence readability even if it will be called only once. In this way the subprogram is not only appropriate for logical structuring, but is also a construct for *physical arrangement*[1] of the program source.

Based on this one can say that using subprograms makes the program source shorter and more readable, and *the code complexity* can be *reduced* significantly.

[1] It could be vital for the arrangement of the program source how the size of the subprograms are chosen. As a rule of thumb a subprogram should be maximum one screen long, that is approximately 20-25 lines. Longer subprograms which still implement an autonomous task should be broken-up, some parts of them should be arranged into separate subprograms – even if only used once.

Changeability

The subprograms solve well defined tasks. For their users and caller program units it is indifferent how these subprograms solve their tasks. The statement sequences from which the subprograms are made of can be changed without any effects on the caller program units. (This statement will be clarified later, see 7.3.2.) For example, if the task of a subprogram is to sort a number sequence, it can be used without the need to know the concrete sorting algorithm applied. In case of a well-designed subprogram its implementing statement sequence can be changed (for example, from bubble sort to a more efficient sorting such as quick-sort), without the need to modify anything on the caller program units.

Maintainability

The subprograms reduce the amount of unnecessary code repetitions in program source (the redundancy of the source), and this – besides easier readability and changeability – has also a positive effect on the maintainability of the program. If a change is needed in the solution of an often used subtask, this modification must only be made in one place, in the solving subprogram.

Although subprograms offer language constructs primarily for implementing control abstraction, they also play an important role in implementing data abstraction. The essence of data abstraction is that during the development of the program units the representations of the data worked with can be abstracted: data is handled with the help of some predefined operations. These operations are subprograms that is control abstractions: these are used independently of their concrete implementations as abstractions to handle our data. For more detail about data abstraction please refer to Chapter 9.

The role of subprograms will be analyzed also in respect to program design. A complex task is usually solved by breaking it up into simpler ones which are tried to be solved independently. This above mentioned top-down design method creates a simple task from a complex one, then from the simplified ones – even simpler subtasks, until reaching a simplicity level where subtasks can be solved easy. The bottom-up, so-called fountain design method focuses on how solutions of simple tasks can be combined to solve more complex tasks. Both design methods are based on the principle that a complex task can be expressed with simpler subtasks. According to this when trying to solve a problem namely the production of the software, more complex programs are built from simpler ones where these simpler ones can be implemented as subprograms which are called by the more complex programs.

7.2 Procedures and functions

In programming theory there are two common kinds of subprograms: procedures and functions. *Procedures implement transformations* on the state space defined

by their variables or on the environment of the program. *Functions* on the other hand *compute a value*, but do not make transformations, and have no effect either on the values of the program variables, or on the program environment; this is often said the functions have *no side-effects*.

Calling procedures can be considered as a statement, calling a function is more like an expression. It can be seen that procedures extend the statement set of the language, and functions extend the operator set used for expressions. The *Max* subprogram on the beginning of this chapter is a function. It has no side-effects, it computes the result value from two input values. A call to it occurs in an expression. The following Ada subprogram is a procedure which swaps the values of two variables passed as parameters:

```
procedure Swap (A, B: in out Integer) is
  Temp: Integer := A;
begin
  A := B; B:= Temp;
end Swap;
```

The *Swap* procedure implements a transformation on variables, calling it results in the modification of two variables. (Assuming again that K and L are integer variables.)

```
K := 42; L := 24;
Swap(K,L);
```

As shown, the call of the procedure is a separate statement. Procedures and functions should be differentiated also by naming them properly. As procedures are used as statements and describe some kind of activity, it is practical to name them like verbs. By contrast functions are used in expressions to compute values – so a noun, an adjective or a participle is more expressive. A subprogram for sorting a sequence of numbers can be called *Sort* if implemented as a procedure, or *Sorted* if it is a function.

Subprograms execute their operations with the values received as parameters. These are called *input parameters*. Functions return their computed values to the caller as the so-called *return value*. By contrast procedures use *output parameters* for this purpose. For example, the two parameters of the *Swap* procedure are input and output parameters at the same time. Functions do not have output parameters, only input parameters are allowed.

We will consider how programming languages implement procedure and function subprograms.

7.2.1 Languages with no difference between procedures and functions

Many programming languages, such as C, C++, Java, C# or CLU do not really differentiate between procedures and functions. In these languages there is no sharp dividing line between statements and expressions either. An expression

can stand alone as a statement, and its execution means the evaluation of the expression which often causes some side-effects. For example, the value of the *c++* expression is the same as the value of the *c* variable, but as a side-effect, the value of the variable is incremented by one. In these languages all subprograms are "functions", but some of them have a return value of an empty "void type". These kinds of functions are practically procedures. Consider the following example how to implement the maximum searching and swapping subprograms in C:

```
int max (int a, int b) {
    if (a > b) return a;
    else return b;
    /* Or simpler: return (a > b) ? a : b; */
}

void swap (int *a, int *b) {
    int c = *a;
    *a = *b;
    *b = c;
}
```

Functions with **void** return type are used rarely in expressions,[2] but real functions with non-**void** return values are used more often as statements, "discarding" the return value and having it evaluated only for the side-effects. In such languages where there is no dedicated construct for exception handling (see Chapter 8.), these discardable return values are usually used to signal an abnormal termination of the subprogram. A good example for this are the *printf* and *fprintf* standard library C functions which are used to output formatted texts. (The first uses standard output, the second writes to a file passed as a parameter.) These functions return an integer value after execution: if this value is negative, it signals an error during the output, otherwise the execution was successful. It is obvious that the caller of these functions is not mainly interested in the return value, but their side-effect is more important, namely how they influence the program environment (the standard output, or the file). If the programmer wishes, the return value can be checked, but it is not mandatory. In the code below the first return value will be checked, the second discarded:

```
FILE *f = fopen ("apple.txt", "w");

if (fprintf (f,"jonathan\n") < 0) {
    printf ("Unable to write to file!\n");
}
```

[2] In C for example, there are expressions applying the comma operator which have void subexpressions. The value of the *swap($&$k,$&$ℓ), max(k,l)* expression is equal to that of the *max(k,l)* expression, but before the evaluation of the latter, *swap* is also executed.

7.2.2 Languages which distinguish between procedures and functions

There are programming languages which have a sharper line between procedures and functions. In Ada, as shown before, there are two different keywords for procedures and functions which can be used in quite different contexts: procedure calls can only be statements, function calls only be expressions. In other languages, such as Modula-2, ALGOL 60 or PL/I, there is a difference, but both kind of subprograms are defined with the same keyword, e.g. **procedure**. In Modula-2 for example, there is only a syntactical difference between procedures and functions: for functions the return type is specified after the parameter list is separated by a colon.

```
PROCEDURE Max (VAR a, b: INTEGER) : INTEGER;
BEGIN
  IF a > b THEN RETURN a ELSE RETURN b END
END Max;
```

The Modula-3 language also differentiates between procedures and functions, but there is the possibility (with the **EVAL** statement) to have a function evaluated just for its side-effects and discarding its return value.

Most of the programming languages do not force the programmer to implement function subprograms without any side-effects, strictly according to the semantic model and only mimicking pure mathematical functions. In those languages which do differentiate between procedures and functions, this characteristic has some feasibility reasons as well.[3] So the absence of side-effects in functions remains mostly a matter of programming style. The programmer must assure that a function, for example, does not change any global variables, because this would lead to a less clear and harder to maintain program, on the other hand it would inhibit the reusability of the subprogram. In case of parallel processes the usage of global variables could even make the functioning of the program untraceable.

Ada assumes the programmer to follow the above style. This assumption pays off directly when writing parallel programs. Protected units (see 13.10.5.) are objects of which procedures can only be executed under mutual exclusion, but their functions can be called simultaneously (concurrently) – knowing that these have no side-effects, so these cannot interfere with each other. So for complying with the proper style the language offers more efficient parallel programs.

At the end we mention those languages where there is no way to write functions, or even procedures with side-effect. Such are the so-called purely functional languages (see Chapter 15.), such as Haskell. In these languages subprograms can directly map to mathematical functions. By eliminating side-

[3] It would not be wise to forbid a function to call a procedure, since it can be useful for debugging and logging purposes, if the function could write to a file. This is already a kind of side-effect. If a function can call a procedure, in there – and so indirectly in the function – other side-effects could also be caused.

effects, the complexity of the programs is intended to decrease. Undoubtedly this has certain benefits, for example, when trying to prove program correctness with mathematical tools.

Avoiding side-effects also improves program readability, clarity and portability. If functions with side-effects are called in an expression, their value (and the order of the statements causing side-effect) also depends on the evaluation order (precedence) of operators within the expression. This makes the understanding of the code more problematic even if the evaluation order within expressions is precisely defined. If this is not defined clearly, expressions can occur easily with uncertain results undefined by the language, and this error is not even captured by the compiler. In such a case, the program can "accidentally function properly" on a given architecture, but on a different system, or just after switching to another compiler everything goes suddenly wrong. The value of the `++i+t[i]` expression in C, for example, depends on the C implementation used: either `++i` or `t[i]` is evaluated first. In Java, expressions are unambiguous (here the left side of addition, then its right side gets evaluated), usage of this kind of expression is thus strongly discouraged: it makes the reading of the program source too difficult.

7.3 Structure of subprograms and calls

The definition of *subprogram program units* are usually made of two parts: the *specification* and the *body*. Specification provides information about how the subprogram must be used and called. The body contains the statement sequence which will be executed if the subprogram is called.

Calling a subprogram is usually initiated by specifying its name and the values of required parameters. It is usually said that the subprogram becomes *active* by its calling. Important notions are also the *formal* and *actual parameters*. Formal parameters are those specified in the definition of the subprogram. They are used throughout its body to describe the operations. Actual parameters are passed by the caller to the subprogram which has to perform its operation with these received parameters. This means that actual parameters *must be matched* against formal parameters. The formal or actual parameter sequence of a subprogram is commonly called as the *parameter list*.

7.3.1 What could be a parameter or return value?

It is an interesting question what kinds of entities are allowed as parameters and return values for a subprogram. In a programming language entities are usually vales, types, subprograms and modules.[4] To use entities as parameters is allowed almost in every programming language. (Except, for example, some

[4] In Ada additional entities are processes (see Chapter 13.), or in Clean the type constructors (see Chapter 15.).

early variants of BASIC among others, which did not allow any parameters for subprograms at all.) Value entities as parameters were shown in the above examples in multiple languages, such as the function computing the maximum and the parameter swapping procedure. Now we look at which parameters are handled differently by the various languages.

Unconstrained array

Many languages allow formal parameters to have indefinite types. With these, such programs can be made which can handle unconstrained arrays.[5] The length of array typed variables is determined differently in various languages. In some languages (such as ASA FORTRAN, BASIC, Pascal) it takes place in compile time, in other languages at declaration evaluation time (Ada, FORTRAN 90 etc.), and there are languages where this happens dynamically (APL, Perl, Common LISP). In most of the languages formal parameters are less strictly handled than variables. Even in Standard Pascal it is possible to write a sorting procedure which is able to use arbitrary length arrays.

procedure *sort(***var** *V:* **array** *[low, high: integer] : integer);*

In PL/I, Pascal, Modula-2, Ada etc. the actual parameter will fully determine the type of the formal parameter. In the above example *low* and *high* can be used to reference the lowest and highest index of the actual parameter. In Ada, if the array *V* is a formal parameter of a subprogram, the lowest index of the matched actual parameter array can be referenced by the expression *V'First*, the last with *V'Last*, the full index range with *V'Range*. So the subprogram implementation can rely on these expressions. (In Ada because of the equivalence of types is based on their names, the type of the formal parameters cannot be constructed within the specification of the subprogram that is why we defined the *Vector*

[5] In Ada, not only the unconstrained, i.e. variable length array can be indefinite, but also the unconstrained discriminated record type. These indefinite types can also appear as formal parameters.

type separately.)

```
type Vector is array (Integer range <>) of Integer;
procedure Sort (V: in out Vector) is
  J: Integer;
  Temp: Integer;
begin
   for I in V'Range loop
     J := V'First;
     while J < I and then V(J) < V(I) loop
        J := J + 1;
     end loop;
     Temp := V(I);
     V(J+1..I) := V(J..I-1);
     V(J) := Temp;
  end loop;
end Sort;
```

In C, arbitrary length arrays can be passed with the help of pointers as parameters. The length of the actual parameter cannot be accessed through the formal parameter, so usually it is passed as an extra parameter. This solution is, of course, more complex and more problematic to read, and tends to be more error-prone: attention must be paid for every call, to set the *size* formal parameter to the right value.

```
void sort (int[] v, int size)
...
```

In Java arrays are objects too, and ”know” their own length. This information can be queried with the *length* attribute. In this language the length of the array is not even part of its type: the declaration of an array variable or formal parameter must not contain any length information. In the next section there will be an example showing how to pass an array as a parameter in Java.

Multidimensional array

The question of unconstrained arrays is getting more exciting if the array is multidimensional. In those languages which support the runtime query of array index boundaries, this is not much of a problem. In Java, multidimensional arrays are such one-dimensional arrays which contain arrays. The length of the contained arrays can, of course, differ, but this is not a problem: the containing array only stores object references. For example, the compiler looks up an element within a two-dimensional array by searching for the subarray reference specified by the index in the first dimension which will be unreferenced and in the result array the

value targeted by the index in the second dimension will be accessed. Consider the following example how to sum the elements of a two-dimensional matrix:

```
double sum(int[][] mat) {
  int sum = 0;
  for (int i = 0; i < mat.length; ++i) {
    for (int j = 0; j < mat[i].length; ++j) {
      sum += mat[i][j];
    }
  }
  return sum;
}
```

In Ada, multidimensional arrays behave like real multidimensional arrays. So every row of a matrix has the same number of elements.[6] The compiler will store the elements more efficiently than Java, most likely with row- or column-major mapping. The boundaries of the actual parameter can be queried with the indexed versions of *Range*, *First* etc. attributes by the number of desired dimension.

In FORTRAN, C, C++ etc. multidimensional arrays are also stored with row- or column-major mapping, but the compiler cannot access the size of the actual value through the formal parameter. This is a problem, since, for example, in row-major mapping the number of columns must be known to be able to locate an element. In FORTRAN the declaration of the formal parameter can include the desired information. The following is a FORTRAN 77 implementation of the above function summing all the elements:

```
      INTEGER FUNCTION MATSUM(MATRIX, ROWS, COLS)
        INTEGER ROWS, COLS, MATRIX(ROWS, COLS), I, J
        MATSUM = 0
        DO 20 I = 1, ROWS
          DO 10 J = 1, COLS
            MATSUM = MATSUM + MATRIX(I,J)
10        CONTINUE
20      CONTINUE
      END
```

The usage of this FORTRAN function requires much more attention than the Java or Ada variant: and this is a rather negative aspect of FORTRAN. Why? Because in the calling of the `MATSUM` function, the programmer must set the actual sizes to the `ROWS` and `COLS` parameters. If accidentally wrong values are specified for these two parameters, the function will use wrong data, and

[6] In Ada it is possible to define an array of arrays, but in this case the element type of the array must be an already definite type. (This definite element type could be, of course, such a pointer type which has an indefinite base type. This corresponds to the array representation in Java the most.)

instead of the elements of the matrix junk content will be accessed from the memory.

In C and C++ the situation is even worse. The type (or just the size) of a formal parameter cannot be passed as another formal parameter, like in FORTRAN. Of course, a plain numeric value can be used for this purpose. Because of the row-major mapping representation of multidimensional arrays the number of columns must be specified for the compiler to be able to generate the proper index function to locate array elements. The number of rows need not be specified, or at least not in the formal parameter array type:

```
int sum (int matrix[][10], int rows)
{
  int i, j, s = 0;
  for (i = 0; i < rows; ++i)
    for (j = 0; j < 10; ++j)
      s += matrix[i][j];
  return s;
}
```

Such a subprogram can be parametrized by the number of rows in the array (carefully matching the *rows* formal parameter to the actual value), but the number of columns is ”wired in” into the subprogram definition: only matrices with 10 columns can be passed to this *sum* function. For a more general form of this function an other solution must be found. For this, for example, pointer arithmetic could be used! In this case the number of rows and columns can be passed in extra formal parameters. This approach is sufficiently general, but it has the same problem as in FORTRAN, namely, it is easy to mess up the function calling, and in addition, this has another problem: indexing must be implemented by the programmer, because the compiler cannot automate this for the lack of information.

```
int sum (int *matrix, int rows, int cols)
{
  int i, j, s = 0;
  for (i = 0; i < rows; ++i)
    for (j = 0; j < cols; ++j)
      s += *(matrix + i * cols + j);
  return s;
}
```

At calling a two-dimensional array can be created, and passed after an explicit typecast as the actual parameter to *sum*:

```
int m[2][2] = {{1, 2}, {3, 4}};
printf ("%d\n", sum ((int*)m, 2, 2));
```

In C it is also possible to declare arrays of arrays, more precisely arrays made of pointer types. This results in a significantly different solution.

```
int sum (int **matrix, int rows, int cols)
{
  int i, j, s = 0;
  for (i = 0; i < rows; i++)
    for (j = 0; j < cols; j++)
      s += matrix[rows][cols];
  return s;
}
```

This function cannot be called with a two-dimensional array. The actual parameter must be defined quite differently – in a much more complicated way – than in the examples before. The following can be used, for example:

```
int s0[2] = {1, 2}, s1[2] = {3, 4};
int *m[2] = {s0, s1};
printf ("%d\n", sum(m,2,2));
```

As the elements of the array m are pointers, a wrong index can lead to reference an unused memory location, which can result in a runtime error (such as segmentation fault).

Subprogram

To use a subprogram as a parameter for another subprogram – this would be sometimes very practical. Consider, for example, a subprogram which has the task to estimate the definite integral of functions. To assess the area under the curve of a function, its value in some basic points must be determined, as shown in the next Pascal example:

```
function integral(function f(n: real):real; low, high, step: real):real;
var x, sum: real;
begin
  sum := 0.0;
  for x := low to high by step sum := sum + f(x) * step;
  integral := sum;
end;
```

To generalize the integral function, there must be a way to pass a subprogram as a parameter. This was not allowed at all in some early imperative languages (ASA FORTRAN, COBOL etc.). The possibility appeared in the functional languages (including LISP already born in the end of the 50's), in ALGOL 60 and in numerous languages designed later (PL/I, SIMULA 67, ALGOL 68, Pascal, FORTRAN 77, Modula-2, Modula-3 etc.).

In several current languages subprograms cannot be, but pointers to them can be passed as parameters. Such languages are, for example, C, C++ or Ada. The above integral function would be implemented in C in the following way:

```
double integral (double (*f)(double),
        double low, double high, double step) {
  double sum = 0.0;
  while (low <= high) {
    sum += f(low) * step;
    low += step;
  }
  return sum;
}
```

Within the body of the *integral* function *f* can be called without explicit dereferencing. Of course the `(*f)(low)` form can also be used.

The same duality also exists in Ada. In some modern languages using subprograms as parameters for subprograms, it is not so obvious as in the languages mentioned before. In Ada 83 only actual parameters that are evaluated in compile time can be used, with the help of the so-called "template" structure. In Java **interface**s can be used as parameter subprograms. These two last possibilities will be discussed in detail in Chapter 11.

Subprograms passed as parameters to subprograms raise many interesting questions, so this topic will also be treated in Section 7.7.

Label

In ALGOL 60, PL/I, SIMULA 67, ALGOL 68 etc. labels can also be passed to subprograms as a parameter. Consider the following ALGOL 60 function:

```
real procedure log(x, error);
  value x; real x;
  label error;
begin
  if x > 0
     then ...
     else goto error;
```

If the statement **goto** *error* is executed, the execution of the subprogram is terminated, and along with all other calling subprograms, until the calling level is reached where the given label exists.

Label parameters can be used for exception handling (see Chapter 8.). Such a label indicates a code which can react to an exceptional event (in this case if the logarithm function was called with a negative parameter). In more modern languages label parameters are replaced by the exception handling mechanisms.

Type and module

The reusability of a subprogram can greatly be improved if types could be passed as parameters. This topic is discussed in more detail in Chapter 11.

What could be a return value?

The type of the return value is usually more limited by the languages than the parameter types. In FORTRAN and in ALGOL60, for example, functions could only return scalar values. In Pascal and in early versions of Modula-2, beside scalars, pointers were also allowed. Many imperative languages are more flexible. In Ada, C or ALGOL 68 composite types can also be returned. In Modula-3 the return value can also be a subprogram. (In Ada 95, C and C++ it can be just a pointer to a subprogram.) In the functional languages (Haskell, LISP, ML etc.) returning a function from a function is not considered to be special case. [Sco09] In these languages, or for example in CLU, BETA, or Sather etc. a function can even return "multiple results". The following CLU subprogram accepts two integers as parameters and returns them in ascending order:

```
sort = proc (a, b: int) returns (int, int)
  if a < b then return (a, b)
  else return (b, a)
  end
end sort
```

This subprogram can be called in the following way:

```
x, y: int := sort(4, 2)
```

At the end please note that types can be more than just parameters (see Chapter 11.). In languages following the "types are also values" principle, there is nothing special about having a type as a return value of a function. For this kind of function an example will be shown on page 284.

7.3.2 Specification of subprograms

For the caller program units the specification of the subprogram is what they can access. It is sometimes also said that *the interface of the subprogram* to the outside word is its specification. Callers cannot see anything from the body. So the control abstraction is realized in this way: the callers face only an abstraction of the concrete statement sequence. The specification usually contains the name of the subprogram, whether it is a function or not, and if yes, the type of the return value, and also which parameters are accepted. Specification is also-called as the *header* of the subprogram.

function *Max(A, B: Integer)* **return** *Integer*

In some languages (such as FORTRAN, ALGOL, Pascal etc.) the header of the subprograms contain a keyword marking the given syntactical unit as a subprogram. Other languages (such as C and its descendants) do not use extra keywords for this. The above line would look like as the following in C:

```
int max (int a, int b)
```

Even the return type could be omitted from the above specification, as in C the subprograms are functions returning an **int** by default:[7]

```
max (int a, int b)
```

There are big differences between languages how much information a subprogram specification contains. The more precisely the specification describes how to use the subprogram, the easier it is to write correct programs in the given language. The compiler uses the specification to check whether the caller of a subprogram is using it correctly or not, for example passing the proper count of parameters. Let us examine in more detail what kind of information a subprogram specification can include in various languages.

Parameter types

One of the most important questions is if it is checked at the calling of the subprogram, if properly typed actual parameters are matched with the formal parameters. In FORTRAN 77 there was no such checking, but in FORTRAN 90, Ada, Pascal, Java and commonly in most of the modern languages it is already done automatically. The C language is an interesting transition between the two possibilities. In the original C language the compiler did not check the number of actual parameters, neither their type. In this language even the syntax was different, it was closer to ALGOL 60, PL/I, etc.[8]

```
int max (a, b)
  int a, b;
{
  return (a > b) ? a : b;
}
```

ANSI C already supports so-called *prototypes* that is the usage of such specifications which also contain the type of the parameters.[9] There was an example for this on page 271.If formal parameters are specified in the traditional way in ANSI C, the compiler does not check the number and types of actual

[7] Another strange default return type can be found in the Objective-C language, see Section 7.3.4.

[8] It is important to note that only the syntax was similar, since in ALGOL 60 and in PL/I the number and types of formal and actual parameters had to be identical.

[9] This possibility was in fact introduced in the C++ language first, ANSI C took it from there.

parameters, but if the new prototype approach is used it does.[10] In C++ only the prototype variant is allowed.

Another interesting question is how deep the type checking of parameters is performed. For example in ALGOL 60, in the early versions of Pascal or in FORTRAN 77, by passing a subprogram as a parameter, the type of the formal parameter could not be specified. Nonetheless in ALGOL 68, Standard Pascal or FORTRAN 90, this could be specified. The formerly shown Standard Pascal example on page 278was the following:

```
function integral (function f (n: real) : real;
                   low, high, step: real) : real;
var x, sum: real;
begin
  sum := 0.0;
  for x := low to high by step
     sum := sum + f(x) * step;
  integral := sum;
end;
```

At calling the *integral* subprogram, the *f* formal parameter can only be matched with such actual functions which return a *real* value for a *real* argument. The compiler can check, if the *f* function, or the matched actual parameter function is used with proper types within the *integral* function. Without this check the *integral* function could receive a parameter which does not return a *real* from a *real*, so a correct (accepted by the compiler) call could be pointless or cause type mismatch.

Type correctness of actual parameters can be influenced by the language having subtypes, or generally polymorphism (for more detail see Chapter 11.). In C, for example, an *int* actual parameter can be matched with a *double* formal parameter: the compiler converts the actual parameter to the proper type automatically. Of course, an array actual parameter cannot match a *double* formal parameter, since an array "cannot be casted" to *double* type.

In object-oriented languages (see Chapter 10.) types can build a hierarchy. These kinds of subtypes allow a formal parameter with a more general type (such as *Shape*) to be matched with a more specific typed (such as *Rectangle*) actual parameter.

In the Ada language, subtypes are not types, and the compiler ignores subtype information at type correctness checking. So, for example, an *Integer* formal parameter can be matched with a *Positive* actual, and vice versa; since *Integer* and *Positive* denote the same type.

[10] In case of a subprogram without parameters the prototype is distinguished from the traditional subprogram specification by directly signaling the absence of parameters in the parameter list: **int** *f* **(void)**.

Number of parameters

There are languages where the compiler does not even check the number of actual and formal parameters. Such languages are typically script languages. The following JavaScript subprogram is a function which can concatenate any number of text objects received as parameters, inserting the separator between them, which is passed as the first parameter. Within the function actual parameters can also be accessed through the *arguments* predefined array object, their count is given by the *length* attribute of that array. (Arrays are indexed from zero to the length minus one.) Please note that the specification of the subprogram neither includes the types of the formal parameters, nor of the return type.

```
function cat (separator) {
    result = ""
    for (var i = 1; i < arguments.length; ++i) {
        result += arguments[i]
        if (i != arguments.length) result += separator
    }
    return result
}
```

As already mentioned, the original C compiler did not check if the subprogram was called with the proper number (and type) of actual parameters. This by no means provided that kind of flexibility which was shown in the JavaScript example above, since in a C subprogram there was no information about the actual parameters. In C and C++ there is nevertheless, another possibility. If at the end of the formal parameters the *dots* is specified, then any number of formal parameters can be matched. Subprograms with this sequence must have at least one normal formal parameter. Usually the actual parameter(s) matching this(these) deliver the information, what other actual parameters follow. A good example for this is the already shown *printf* standard library function, which has the following specification:

```
int printf (char *format, . . .)
```

The *format* array holds control characters which specify what actual parameters should the subprogram expect. These control sequences start with the percent character. For example, `"%d"` denotes an integer actual parameter which must be printed out in a decimal format, `"%s"` signals a character array. Some correct calls of this function:

```
printf ("10*10=%d", 10*10);
printf ("%s=%d","10*10", 10*10);
printf ("no extra parameters needed");
```

In the case if the control signs in the *format* character array and the given additional actual parameters do not match, the function of the *printf* subprogram is not defined – unfortunately the compiler cannot check for such kind of errors.

There are also languages with rather complex type system and compiler to be able to express such things, like if the type of a parameter of the subprogram is dependent on the value of another parameter. Such a language for example is Cayenne, which has a type system based on the notion of "dependent type".

In the next example [Aug99] the type of the first parameter (called `fmt`) of the `printf` function is `String`. The type of the other parameters and of the return value depend on the value of this parameter: `PrintfType fmt`. For example, if the `fmt` string is started with the `%d` characters, the `printf` is awaiting an `Int` typed second parameter, and the type of the additional parameters and of the return value is defined by the further part of `fmt`. For example, if the further part of `fmt` is the empty string (`Nil`), then there are no more formal parameters, and the type of the return value will be `String`.

```
PrintfType :: String -> #
PrintfType (Nil)          = String
PrintfType ('%':('d':cs)) = Int   -> PrintfType cs
PrintfType ('%':('s':cs)) = String -> PrintfType cs
PrintfType ('%':( _ :cs))  = PrintfType cs
PrintfType (_ :cs)         = PrintfType cs
printf :: (fmt::String) -> PrintfType fmt
```

In some languages the formal parameters of the subprogram can have default values. This can help (a little) to achieve the flexibility shown in the previous example. The question will return on page 293.

In other languages passing an array can mimic variable length parameter lists. This possibility can be used only for parameters with the same type in strongly typed languages, but in weakly typed languages, such as Smalltalk, an array can hold multiple parameters with arbitrary types. In the strongly typed C# also arrays can mimic variable length parameter lists. Using the **params** keyword allows to avoid arrays within the calls:

```
void ShowNumbers (params int[] numbers) {
    foreach (int x in numbers) {
        Console.Write (x + " ");
    }
    Console.WriteLine();
}
...
int[] x = {1, 2, 3};
ShowNumbers (x);
ShowNumbers (4, 5);
```

In C and C++ languages the main programs have another interesting feature, according to the number of formal and actual parameters. In these languages the main program is a function called *main* which can have two kinds of parameter lists. If the command line arguments are not used within the program, the specification of *main* would look like this:

```
int main()
```

If the command line arguments are needed, *main* must be specified like this:

```
int main (int argc, char* argv[])
```

Here the array (*argv*) of command line arguments and their count (*argc*) are passed by the operating system as actual parameters.

In Java the fist variant – omitting the command line arguments from the formal parameter list – is not an option. The main program in Java must be specified this way:

```
class Arguments {
  public static void main(String[] args) {
    for (int i=0; i < args.length; ++i)
      System.out.println(args[i]);
  }
}
```

In other languages (such as Ada, Pascal, Modula-2) the main program cannot receive parameters. In this case usually standard libraries help programmers to access command line arguments.

The above Java subprogram looks the following in Ada:

```
with Ada.Command_Line, Ada.Text_IO;
procedure Arguments is
begin
   for I in 1..Ada.command_Line.Argument_Count loop
      Ada.Text_IO.Put_Line(Ada.Command_Line.Argument(i));
   end loop;
end Arguments;
```

Name of parameters (matching by name)

although in most languages the names of the formal parameters are part of the subprogram specification, they are not for the caller program units: for checking the validity of the call the compiler only uses the number of parameters and their type. (This is usually called as the *profile* of the subprogram. The profile of the subprogram, and – in case of a function – the return type is usually called as the *protocol* of the subprogram.) The names of the formal parameters only

have informal meanings, which helps the programmer to better remember their role.

In contrast, for example, in Ada, PL/SQL or Modula-3, the subprogram specification makes also the names of the formal parameters accessible to the caller program units. In these languages at calling a subprogram its formal parameters are not only be matched *by position*, but also *by name* with the actuals. Positional matching means that the first formal parameter is matched with the first actual, the second with the second, and so forth. When using matching by name, the order is not relevant. Consider the following Ada subprogram specification from the standard library which inserts a string into another before a given position:

```
function Insert(Source   : in String;
                Before   : in Positive;
                New_Item : in String) return String
```

The following function calls are all valid:

```
Insert("Duck", 3, "lo") -- positional matching
Insert(Source => "Duck", Before => 3, New_Item => "lo")
Insert(Before => 3, Source => "Duck", New_Item => "lo")
Insert(New_Item => "lo", Before => 3, Source => "Duck")
```

The programmer does not have to remember the exact order of the parameters: if the names of the formal parameters are known, actual values can be passed in any order. Positional and matching by name can even mix. In this case the first few parameters are matched by position, the rest by name, for example:

```
Insert("Duck", New_Item => "lo", Before => 3)
```

Parameter matching by name is particularly useful when combined with default parameter values. (See also page 293.)

Throwable exceptions

Exceptions that a subprogram can throw are ”special return values”. The caller should be prepared to handle these thrown exceptions. So it is fully justifiable to have the throwable exceptions of a subprogram listed in its specification. In Ada it is not possible, in C++ not mandatory, but for example in Java it is mandatory[11] to specify the throwable exceptions in the subprogram header:

```
int readin(InputStream in) throws IOException
```

This *readin* operation receives an *InputStream* parameter, and usually returns an **int**. In exceptional cases the execution terminates with an *IOException*. For more details about exceptions please refer to Chapter 8.

[11] To be precise, only the so-called ”checked” exceptions must be named in the subprogram specification.

Specialty of object-oriented languages

In some languages the specification of a subprogram can hold additional information. In object-oriented languages (C++, Java, C#), for example, the visibility of the subprogram (which is usually a member of the class definition) can be specified. For this purpose usually the **public**, **protected** and **private** keywords are used. The **public** modifier denotes the most wide, **private** the most narrow visibility. For example in Java like this:

```
public void start()
```

In object-oriented languages the **static** keyword can appear in the subprogram specification. This is to denote that the subprogram is not executed on an actual object, but on a class.

```
public static int exit()
```

For more details about object-oriented languages, please refer to Chapter 10.

7.3.3 Body of subprograms

The body of the subprograms contains the statements which have to be executed at every call. Usually it is possible to define so-called *local variables* within the subprogram, which can be used as auxiliary variables for the implemented computations. The variable declarations and the statements using them are completely separated in some languages (such as Pascal, Ada), in other ones (such as C) it is not that strictly regulated, and in some others (such as Java, C++) not at all. The body of a Pascal subprogram, for example, is made of two parts: the declaration part where – among others – local variables are defined, and the statement sequence part. These two parts are separated by the **begin** keyword.

```
procedure swap(var a, b: Integer);
var temp: Integer;
begin
  temp := a;
  a := b;
  b := temp;
end;
```

In C nothing is written between variable declarations and the statements. Statements must simply be preceded by the variable declarations.

```
void swap (int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

In Java, for instance, this limitation does not exist. Declaration statements can be mixed freely with other statements. In Ada (or, for example, PL/SQL) exception handling (see Chapter 8.) is not managed as a statement (such as in Modula-3, C++ or Java), but can be appended to a block. According to this, the body of a subprogram in Ada can consist of three parts: declaration part, statement part and an optional, exception handling part. The subprogram below is an operation of a stack. The stack is represented by a record which has an array field named *Data*: this array will hold the elements of the stack. The *Top* field of the record is the index within the array for the last stored element. The size of the array is set by the type definition of the stack, or at the creation of the stack object. If too many elements are attempted to be stored in the array, its index will overflow, and a *Constraint_Error* is raised. In this case, we have to throw a *Stack_Overflow* exception.

```
procedure Push (S: in out Stack; E: in Element) is
begin
  S.Top := S.Top + 1;
  S.Data(S.Top) := E;
exception
  when Constraint_Error => S.Top := S.Top-1; raise Stack_Overflow;
end Push;
```

In this subprogram definition the declaration part is empty, the statement part has two statements, and there is an exception handling part as well.

In many languages (such as Eiffel, Turing, Blue) the subprogram body can also contain logical assertions useful for correctness proving. This question is further discussed in Chapter 12.

Return from subprograms

By returning from a function, a properly typed value must be returned to the caller. This has a wide variety of forms in different languages. Perhaps the most common is to use the **return** statement. This is used, for example, in Ada (see page 267.),in C (see page 271),in JavaScript (see page 283.),and in most of the other modern imperative languages. The **return** statement causes the

subprogram to terminate immediately[12] and returns the value of the expression specified after it.

In the early imperative languages, such as FORTRAN, ALGOL 60 and Pascal, instead of the **return** statement an assignment to the name of the function had to be performed (see for example page 278.). This was unfortunate, because the name of the function could not be hidden within the function body, and this made the rules of hiding declarations a little too complicated.[13]

The **return** statement has its own problem as well. If a function has already computed its return value, but it still has to perform further operations (for example it has to close a file, or release a resource), a local variable must be used to temporarily store the intended return value. A similar problem occurs also in the following Ada function:

```
function Max_Pos (V: Vector) return Index is
  Rtn: Index := V'First;
begin
    if V(I) > V(Rtn) then Rtn := I; end if;
  end loop;
  return Rtn;
end Max_Pos;
```

A more efficient solution is supported by the SR language[14] [Sco09].

```
procedure Max_Pos (ref V[1:*]: int) returns rtn: int
  rtn := 1
  fa i := 1 to ub(V) ->
    if V[i] > V(rtn) -> rtn := I fi
  af
end
```

Here the memory allocated for the return value could be addressed as a plain variable. In Eiffel the return value of the function can be managed as a local

[12] An exception to this rule is, if the **return** statement is within a sequence which has exception handling, like in Java within a **try** block, which has a **finally** part as well. In this case the **finally** part gets executed before leaving the subprogram. Likewise in C++, the destructors of locally instantiated objects within the subprogram are also executed before exit.

[13] There is also a `RETURN` statement in FORTRAN, which causes the subprogram to return. Nevertheless this statement can not be used to specify a return value.

[14] In this example the meaning of **fa** is "for all", **ub** denotes "upper bound". The arrow has the same meaning as in other languages *do* and *then*. For closing composite statements, the language uses the statement starting keyword reversed.

variable, but the name of this variable is fixed: it is always called *Result*.

```
Max_Pos (v: ARRAY[INTEGER]) : INTEGER is
  local
    i: INTEGER
  do
    Result := v.lower;
    from i := v.lower until i>v.upper loop
      if v@Result < v@i then
        Result := i
      end
    end
  end
```

The Euclid language combines the above possibilities with the usage of the **return** statement interestingly. If a name is specified for the return value in the header of the subprogram, an assignment to this name must be done within the subprogram body, but if it is unnamed, the **return** statement must be used.

In functional languages and in ALGOL 68 (the latter one does not differentiate between statements and expressions) the return value of a function is the value of the function body, as an expression. The function computing the maximum of two numbers is the following in ALGOL 68:

```
proc max = (int a, b)int: if a > b then a else b fi
```

Let us turn back to the **return** statement for one moment. This statement can be used not only in functions, but often in procedures as well, but of course, without any return value specified in this case. For example, if a subprogram should be left from the middle of its code, one can use something like the following Ada 95 procedure:

```
procedure Search_Zero (M: in Matrix; I, J: out Index) is
begin
  I := M'First(1);
  while I <= M'Last(1) loop
     J := M'First(2);
     while J <= M'Last(2) loop
        if M(I,J) = 0.0 then return;
        end if;
        J := J + 1;
     end loop;
     I := I + 1;
  end loop;
end;
```

The **return** statement need not be used in a procedure, but it is obligatory in functions. In Java, for example, it causes a compilation error, if the compiler detects that a function could end without executing a **return** statement.

```
int notCompilable() {
   int i = 42, j = 42 * i - 42 * (i - 1);
   if (i == j)
       return 1;
}
```

This function is sound, but not correct – in the sense that the compiler will not accept it, even if we know that execution will always reach the **return** statement. The compiler in Ada is not so rigorous. In case of such a function definition we would receive only a compilation warning at most. Nonetheless if a function misses the **return** statement in runtime, the Ada system will throw a *Program_Error* exception.

Body in an other language

Sometimes the body of a subprogram should be implemented in a language different from the one where it should be called. This could have efficiency reasons (for example, if some code must be very efficient, so it should be implemented in assembly), or because of reusability (if the desired subprogram is already implemented, and should be used without rewriting it from another language).

In Ada, for example, special compiler directives and library units can be used to be able to call subprograms implemented in other languages from the Ada program, or vice versa. If, for example, the *Sqrt* function was implemented in FORTRAN, the *Import* compiler directive can be used to make it visible within the Ada program.

```
function Sqrt (X: Float) return Float;
pragma Import(FORTRAN, Sqrt);
```

As another example shows it: in Java the subprogram specification can contain the *native* keyword, which means that the body of this subprogram is implemented in a different language than Java:

```
public native float sqrt(float x);
```

The Java virtual machine approach also enables programs written in other languages (such as Ada) to be integrated without any special effort into Java programs. Only a compiler for the other language is needed which is able to produce machine code for the Java Virtual Machine (such as jgnat for Ada). The same principle is utilized by the .NET architecture: for a given code snippet it does not matter in which programming language it is written or presented, its meaning is described based on a Common Language Runtime.

7.3.4 Calling subprograms

Starting (activating) a subprogram is usually done by specifying its name and its actual parameters. In some early imperative languages, such as FORTRAN and PL/I, to call the procedures the `CALL` statement had to be used:

```
CALL SWAP (A, B)
```

The parameter list is in most of the languages enclosed by `(` and `)` characters. Usually empty parameter lists are also denoted by empty parentheses. For example, the following C "function" reads in a number and prints it out squared:

```
#include <stdio.h>
void square() {
   int n;
   scanf("%d",&n);
   printf("%d",n*n);
}
```

This subprogram can be called like this:

```
square();
```

In some languages the empty parentheses for empty parameter list need not (PL/I) or must not be (Ada) specified. In Ada, for example, the squaring subprogram can be implemented like this:

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Square is
  N: Integer;
begin
  Get(N);
  Put(N*N);
end Square;
```

To call this subprogram use this:

```
Square;
```

Functional languages, among others, have special parameter notations. In LISP, for example, not the actual parameter list, but the whole call is enclosed: `(+ 4 2)`. In the modern functional languages, such as Haskell, the function name and its parameters are only separated by a space, as in the `exp 5` expression. This form is very strange, since the function call has the highest priority in these languages. Despite this, the two parts (namely the name of the called function and its parameters) which have the most tight connection, are connected with a character, the space, mostly used for separation. The same holds also for Smalltalk, for example: `h at: k put: o`.

Default value of formal parameters

There are languages which support default values for formal input parameters. In this case it is not needed to match all the formal parameters with actual values: formal parameters without actual values will have their default values – if specified. In these languages the handling of the number (and types) of formal parameters is not so flexible, as in C/C++ with *dots* or in JavaScript that uses the *arguments* array. This is because such subprograms always have the same number of parameters, but not all of them must be explicitly set. Consider the following C++ example:

```
void draw_rectangle (int width, int height,
                     int x = 0, int y = 0, int color = 0)
```

When calling this subprogram it is enough to set the *width* and *height* parameters – the others can also be set, but it is not mandatory. The following, with exception of the last one are all correct calls:

```
draw_rectangle (10, 10);
draw_rectangle (10, 10, 20)
draw_rectangle (10, 10, 20, 15)
draw_rectangle (10, 10, 20, 15, 3)
draw_rectangle (10) // Error, height has no value.
```

This feature is very useful for subprograms with many formal parameters which usually have the same values. So there are parameters which are important to be set (such as *width* and *height* in the above example), and there are ones, that are less important (all the others above).

In C++ there is a rule that if the default value of a formal parameter is used in a call, all the following parameters must use their defaults as well. According to this, formal parameters with default values must be placed at the end of the parameter list, in a descending order of their importance.

In Ada, Modula-3, etc. parameter matching by name can be combined with default parameter values. So subprograms can be called very flexibly. In C++, for example, there is no equivalent to this Ada procedure call:

```
Draw_Rectangle(10, 10, color => 3);
```

In languages where formal parameters cannot have default values, overloading the name of the subprogram can be an – less elegant and flexible – alternative. We will return to this question in Section 7.6.

Entry points

In some languages, such as PL/I and FORTRAN 77, several entry points can be defined for a subprogram, so it can be started not only ”from its beginning”, but also ”from the middle”. Entry points can be placed anywhere within the

subprogram body, can have parameter list, and can be called the same way as the subprogram. The body belonging to a given entry point starts from the entry point, and ends at the end of the including subprogram. Local variables and statements of the subprogram are also reachable from the entry points.

With the help of entry points, for example, default valued formal parameters can be mimicked. As we can see, the following FORTRAN 77 subprogram increases a variable with the given value or with one if the default increment is used.

```
      SUBROUTINE INCWITH (K, L)
        INTEGER K, L, I
        I = L
        GOTO 10
      ENTRY INC (K)
        I = 1
10      K = K + I
      END
```

In contrast to the subprograms with formal parameters with default values, this FORTRAN 77 procedure must be called by two different names for the two different cases.

```
      CALL INCWITH (I, 3)
      CALL INC (I)
```

The ”main section” of the procedure body which is executed for both calls, starts from the label 10 and in this case it includes only one statement. The subprograms with multiple entry points in PL/I approximate the module concept of later languages. With their help abstract objects could also be implemented elegantly [Koz92]:

```
STACK: PROCEDURE(SIZE);
  DECLARE SIZE FIXED BIN;
  DECLARE S(*) CHAR(1) CONTROLLED,
         N FIXED BIN STATIC INIT (0);
  ALLOCATE S(SIZE);
  RETURN;
PUSH: ENTRY(X);
  DECLARE X CHAR(1);
  N = N+1;
  S(N) = X;
  RETURN;
POP: ENTRY RETURNS (CHAR(1));
  N = N-1;
  RETURN (S(N+1));
END;
```

This subprogram defines a stack object with hidden implementation. Creating this object takes place through the `STACK` entry point. For this, the maximal size of the stack must be specified. Allocation of `CONTROLLED` variables happens during runtime with the `ALLOCATE` statement, specifying the size of the variable to be created. `STATIC` variables keep their values also between multiple executions of the subprogram. The above subprogram can be used like this:

```
DECLARE C CHAR(1);
CALL STACK(10);
CALL PUSH('A');
CALL PUSH('B');
C = POP;
```

More modern languages replaced the entry points with more structured and elegant constructs, such as default values for formal parameters of subprograms, also modules and classes supporting data abstraction (see Chapter 9.).

Currying

In functional languages not necessarily all actual parameters must be specified by referencing a subprogram. The value of an expression which contains a function call with a partially filled parameter list, would be a new function with fewer expected parameters. For more details about this method, called currying, please refer to Chapter 15. The Sather language offers similar possibilities. With the help of `bind` a parameter (such as `3`) can be specified for an operation (such as `plus`), while other parameters can be left unset. The latter is substituted by an underscore. The following `a` function awaits only one `INT` parameter, and returns the same type:

```
a:ROUT{INT}:INT := bind(3.plus(_));
```

Subprograms defined this way can be called later with a `call` by specifying value(s) for unset parameter(s):

`x: INT := a.call(4);` – `x` *will become 7*

It is an important limitation that all output (also input-output) parameters must be left unset when using `bind`, since these parameters will return values after an actual call of the subprogram.

Calling subprograms in object-oriented languages

In object-oriented languages (see Chapter 10.) subprograms often rely on objects, they are called "on an actual object" and are called not subprograms, but methods. That is why their first parameter is not in the parameter list, but

comes before the name of the subprogram. For example, in Java the *translate* operation of an *p* object of type *Point* would be called like this:

```
p.translate(4.0, 2.0);
```

The same call in Ada would look like the following:

```
translate(p, 4.0, 2.0);
```

In object-oriented languages within the body of an operation the actual object, for which this operation was called, can usually be referenced by a special keyword (such as *this*, *self*, *Current* or *me*). The formal distinction of the object written before the operation and of the other parameters often covers differences in their meaning. In these languages by calling a subprogram, it is possible for the runtime environment to choose from multiple (alternative) bodies of the operation which has to be executed (see: late or dynamic binding in Section 10.7.2.). This choice is based on the dynamic (that is only known during runtime) type of the object, for which the given operation applies.

In some of the object-oriented C-derived languages it is very common to have nearly never operations with **void** return type. If an operation of an object would not return any value, it could be altered to have something to return: the object itself. With this method "call chaining" can easily occur on the object:

```
rectangle.translate(4.0,2.0).mirror(0.0,0.0).enlarge(2.0,0.5);
```

For this the operations could be defined the following way in the *Rectangle* class – using Java syntax:

```
public Rectangle translate (double dx, double dy) {
  xpos += dx;  ypos += dy;  return this;
}
```

Please note that some languages offer quite exotic syntax for specifying the name and formal parameter list of subprograms. For example, in the Objective-C language such a subprogram specification can be given:

```
- putObject: element atX: (int) x Y: (int) y
```

This is an instance level method (subprogram assigned to an object, see Chapter 10.). Instance level methods begin with the - sign, class level methods with the + sign. The name of the method is the following:

```
putObject:atX:Y:
```

The method has three parameters, the last two (*x* and *y*) are integers, the first (*element*) and the return value are object references (*id*). If the return type of a method is not specified, the language assumes the common *id* type. In this language it is usual for methods to return the actual object, if there is nothing else to return.

The calling of methods is also rather interesting in the Objective-C language. For example, if the above method can be executed on matrix typed objects, and m is of this type, furthermore e is an object, the following call could be used.

```
[m putObject: e atX: 5 Y: 3];
```

7.3.5 Recursive subprograms

Subprograms, which call themselves directly or indirectly are called recursive subprograms (see Section 3.11.). Multiple instances of a recursive subprogram can be active at a given point of the program execution, which makes their implementation more complex and costly than for subprograms without recursion. Using recursive subprograms, complex calculations can be implemented simply and understandably, with mathematical methods (the induction) in an easily manageable way. Recursion is an alternative to loops. The following Ada function sums the elements of an array by using recursion:

```
type Irray is array (Integer range <>) of Integer;
function Sum (V: Irray) return Integer is
begin
  if V'First >= V'Last
    then return V(V'First);
    else return V(V'First) + Sum(V(V'First+1..V'Last));
  end if;
end Sum;
```

The designers of the first few languages have not realized the possibilities of recursion [Set96], or just found its implementation too costly. In FORTRAN (until FORTRAN 90) there was no way to write recursive subprograms. This is possible in FORTRAN 90 and PL/I, but the specification of the subprogram must indicate that it is recursive. This allows the compiler to produce much more efficient code for non-recursive subprograms. The following PL/I function computes the factorial:

```
FACT : PROCEDURE (N) RECURSIVE RETURNS (FIXED BIN);
  DECLARE N FIXED BIN;
  IF N=0 THEN RETURN(1);
         ELSE RETURN(N * FACT(N-1));
END;
```

7.3.6 Declaration of the subprograms

In some languages there is a limitation for every entity appearing in the program so their type must be specified before usage. Technically, it is customary to say that all entities must be declared before usage. In such languages this is also true for subprograms. There are situations when subprograms cannot be declared by

giving their definition. For example, in case of an indirect recursion, that is, if subprogram `A` calls subprogram `B` which calls subprogram `A`, there is no way to give the definition of the two subprograms so that they both would be defined before usage.[15] In this case a *forward declaration* can help. It essentially consists of the specification of the subprogram, giving only information about how to use it. Consider the following not too creative, but simple enough Ada code snippet to illustrate this principle:

```
procedure Print_Tree (T: in Tree);

procedure Print_Children (T: in Tree) is
begin
  Print_Tree(Left(T));
  Print_Tree(Right(T));
end Print_Children;

procedure Print_Tree (T: in Tree) is
begin
  if not Empty(Tree) then
    Print_Node(Node(T));
    Print_Children(T);
  end if;
end Print_Tree;
```

Another case is if the subprogram should not be declared by its definition because of implementation hiding. In Ada, Modula-2 and other languages that support modules, modules are not only the constructs of encapsulation, but also of implementation hiding (see Chapter 9.3.). In the specification part of the module its usage is specified (such as its exported subprograms, types, variables etc. can be declared here), the body of the module contains the implementation of its exported entities. So the specification part of the module only contains the specification of the subprograms, which is enough to decide if these are called correctly by the user program units of the module. The full definition of the subprogram is contained by the body of the module. Consider the following Modula-2 example:

```
DEFINITION MODULE Stack;
EXPORT Push, Pop, IsEmpty;
TYPE ELEM = INTEGER;
PROCEDURE Push (i: ELEM);
PROCEDURE Pop() : ELEM;
PROCEDURE IsEmpty() : BOOLEAN;
END Stack.
```

[15] Unless they are nested in each other. But this has the effect that the nested subprogram is not accessible from outside of the including one, so it cannot be called from outside.

For the user program units of the *Stack* module the subprograms *Push*, *Pop* and *IsEmpty* were declared in the specification part of the module, called as the definition module in Modula-2. The implementation of the subprogram is delivered by the so-called implementation module.

In the object-oriented languages the notion of the module usually overlaps with that of the classes. In Objective-C the specification (**interface**) and the implementation (**implementation**) of a class are separated: in the specification subprograms and types are declared, but their definition is contained in the implementation. There are object-oriented languages, such as Java, which do not separate the specification and the implementation of classes (for example, in two separate compilation units). Furthermore, the operations of a class are not in a sequence, but in a set, so at indirect recursion there is no need for separate declarations. (The functional languages are similar in this respect: the order of functions defined within the same compilation unit is arbitrary, the definition does not have to precede the usage.) In C++ there are two possibilities to define the methods. One form of it is like in Java, when the body of the subprogram is given within the code of the class, the other form only specifies the method there (according to C/C++ notion, only its prototype is given), the body is defined later.

```
class Stack {
  void push (int element); // only a specification
  int pop () { . . . }      // full definition
};
void Stack::push (int element) { . . . } // definition
```

In connection with separate compilability we will return to subprogram declarations in Section 7.5.1.

7.3.7 Macros and inline subprograms

Macros can be considered as the ancestors of subprograms. They were introduced in the assembly languages, but can be used in RATFOR (a preprocessor for FORTRAN), LISP, BLISS, or in the C preprocessor, probably this is the best known. The C macro can be used to arrange the program source, like the subprograms. It is an efficient, but not too safe language construct. It can accept parameters, but the type of actual parameters will not be checked against the types of the formal parameters. Macros are more efficient than subprograms, from the aspect that the extras of subprogram calling and returning are omitted. The body of the macro is substituted at every call into the program source, so no extra administration is required for calling (such as saving registers and return address, etc.). For shorter running time, it is more efficient to use macros, but the final size of the program with macros will be greater than with subprograms.

The main concern about macros is that their substitution is performed before the actual compilation and program correctness checking, so their correctness is

not checked by the compiler. Errors within macro bodies only occur at the first usage of the given macro, and it is relatively difficult to understand where to find the cause of the error from the substituted source within the original source. Besides, the proper usage of the macro (proper types of the actual parameters) can only be figured out by reading the source of the macro. After modifying the body of the macro, all the macro calls must be rethought, and if needed, they have to be modified as well. All of these violate the principle of implementation hiding. Consider the following C macro:

```
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
```

Please note, how many seemingly unnecessary parentheses the body of the macro contains. They are indeed needed, as the body of the macro must be prepared for any environment where it should appear after substitution. For example, the parameters of the macro are always between parentheses, because if the actual parameter is an expression, it must be evaluated before the body of the macro.[16]

Regarding C macros there is another negative issue: within the macro no language structure can be used which is not allowed at the place of the macro call. For example, if the macro is returning a value, and is used within an expression, its body cannot contain variable declarations or composite statements (branching, loop). This again violates the principle of implementation hiding, since not only the specification of the macro, but its implementation also depends on the source context where the macro will be used, and vice versa: possible calls depend not only on the specification of the macro, but also on its body.

Some languages (Ada, C++, Euclid, LISP etc.) try to combine the advantages of macros and subprograms by supporting the usage of inline subprograms. In these languages the programmer can usually give hints to the compiler, which subprograms should be compiled as substitutions. In Ada, for example, the *Inline* compiler directive can be used for this purpose.[17] Consider the following C++ example how to hint the compiler to compile for substitution:

```
inline int max (int a, int b) { return a > b ? a : b; }
```

Inline programs beside having the efficiency of macros, can offer safety by their subprogram specification, and can also ensure the production of semantically clearer code. The inline subprograms represent only an implementation technique, they do not have an effect on the function of the program. This statement can be fully understood by considering an example where an actual parameter has also side effects.

[16] The efficiency of the macros is greatly reduced if the actual parameter is a complex, time consuming expression. In this case the expression is evaluated as many times as often it appears within the macro body.

[17] In Ada other compiler directives (pragmas) can also be used to influence the optimization made by the compiler. For example, with the *Optimize* pragma one can specify if a given subprogram should be optimized for running time or for memory usage.

Calling the *MAX(++x, y)* macro the value of variable x will be increased by two, which seems inexplicable, until the body of the macro is examined. The same problem will not appear if calling the inline subprogram: *max(++x, y)*. We will return to some other problems concerning macros on page 307.

7.3.8 Subprogram types

In some programming languages (such as ALGOL 68, PL/I, Modula-2 or 3) variables of subprogram type cannot only occur as formal parameters for subprograms. In Modula-2, for example, such a type can be defined, contrary to the Standard Pascal that can have subprograms as type values:

```
TYPE Sorting = PROCEDURE (VAR ARRAY OF INTEGER);
PROCEDURE QuickSort (VAR V: ARRAY OF INTEGER); ...
PROCEDURE HeapSort (VAR V: ARRAY OF INTEGER); ...
VAR Sort: Sorting;
VAR V: ARRAY (1..10) OF INTEGER;
...
Sort := QuickSort;
...
Sort(V);
```

In C, C++, Ada etc. only types that point to subprograms can be defined. In Ada 95 the previous code would look like this:

```
type Vector is array (Integer range <>) of Integer;
type Sorting is access procedure(V: in out Vector);
procedure QuickSort (V: in out Vector) is ...
procedure HeapSort (V: in out Vector) is ...
Sort: Sorting;
V: Vector(1..10);
...
Sort := QuickSort'Access;
...
Sort(V);
```

In some languages subprogram typed literals are also supported. In functional languages for example, (see Chapter 15.) the λ-expression can be used to assign a function definition to a variable. ALGOL 68 supports something similar:

```
1.  proc(int)int f;
2.  f := (int i)int:i+1;
3.  print(f(10));
```

f here is a function variable which will be assigned a function literal in line 2 which will be called in line 3 through f.

7.4 Passing parameters

Subprogram parameters can be grouped in three groups based on the direction of information flow. The *input parameters* can be found in the first group. They deliver information from the caller into the called subprogram. The second group is for the *output parameters* which return information from the subprogram to the caller. The last group of *in- and output (inout) parameters* ensure a two way information flow.

Programming languages implement these input, output, and in- and output parameters in different ways. Parameters even from the same group are implemented differently in various languages. Different *parameter passing modes* for the various languages exist to pass the actual to the formal parameters. This section will cover the most widespread parameter passing modes.

7.4.1 Parameter passing modes

The classification of parameters by information flow direction is semantics based. In this section the implementation based classification of parameters will be covered. By elaborating the various parameter passing modes not only semantic issues were raised, but also those of efficiency. This kind of comparison will be discussed in Section 7.4.2.

Parameter passing by value

This parameter passing mode supports only the implementation of input parameters. It means that the value of the actual parameter at calling time is used within the subprogram. The formal parameter can be seen as a local variable of the subprogram which will get its starting value assigned at the calling of the subprogram from the value of the actual parameter. Changes of the formal parameter have no effect on the actual parameter. The actual parameter does not even have to be a variable, a complex expression can be matched with the formal parameter. At the calling of the subprogram, the expression in place of the actual parameter is evaluated, and then the result value is assigned to the formal parameter.

This is the default parameter passing mode in Pascal, and in C this is essentially the only mode. Arrays are the only exception from this last statement. The notion of arrays and pointers in C are a little blurred together. So by passing an array as a parameter, in fact, its address is passed. If an array element is modified within the subprogram, the modification is made in the array referenced by the actual parameter. So arrays are, in fact, passed by address – see next section – in C.

Consider the following example for parameter passing by value. This C subprogram computes the greatest common divisor for two positive integers:

```
int gcd (int a, int b) {
    while (a != b)
      if (a > b) a -= b;
      else b -= a;
    return a;
}
```

How would this function be called?

```
int x = 10, y = 5;
y = gcd(x, x + y + 1);
```

After executing the second line, the value of variable x stays the same that is *10*, the variable y will be set the greatest common divisor of *10* and *16* as a new value, so its new value will be *2*.

It is important to note that the actual parameter will be evaluated only once, at calling time, and will be set to the formal parameter. The consequence of this is that not only the actual parameter is indifferent to the changes of the formal parameter (in the example above x against the changes of a), but also vice versa, the changes of the actual parameter will have no effect on the formal parameter. For example, if the actual parameter is a non-local variable for the subprogram, changing it through a non-local reference will not change the value of the local parameter. Consider the following C example to understand this:

```
int x = 1;
int f (int a) {
    ++x;
    return a + x;
}
int main () {
    int y = f(x);
}
```

By calling f, the formal parameter a is set to the value of the global variable x, to 1. Afterwards x is incremented by one – this will not change a –, and the return value will be $1 + 2$ that is 3.

Parameter passing by reference

This parameter passing mode is most often used to implement in- and output parameters. This means, that the formal parameter gets the address of the actual parameter, or a reference to it. So the formal parameter is another name (an alias) for the actual parameter. During the subprogram execution both denote the same instance (the same memory location) in every moment. Changes of

the actual parameter cause the same changes on the formal parameter, and vice versa. This is how outward information flow is implemented: the result of the computation performed by the subprogram in the formal parameter is reached after its returning through the actual parameter. Of course, the actual parameter has to be an L-value that is such an expression to which a value can be assigned, so that it can be used on the left side of an assignment. The actual parameter could be, for example, a variable, an array element, a record field, etc. The evaluation of the actual parameter means the determination of the memory address of the L-value: this address will be passed into the formal parameter. The evaluation of the actual parameter is performed once, before the beginning of the execution of the subprogram.

For output, as well as in- and output parameters, the compiler checks if the actual parameter is an L-value. If not, there would be no target for the result computed by the subprogram "to store into". FORTRAN is an exception to this rule, as it will be described on page 312.

In Pascal, the declaration of a formal parameter can specify if the actual parameter should be passed by address. For this the *var* keyword is used.

```
procedure swap (var a, b: Integer);
var temp: Integer;
begin
  temp := a;
  a := b;
  b := temp;
end;
```

The fact that the actual and the formal parameters are the same instance, can cause strange anomalies. The reason for this, is the formation of so-called "aliases". By this, it is meant that at some point within the program two different variables denote the same instance. Consider the procedure p having two formal parameters by address, a and b. Let us assume that the task of this procedure is to print out and set to zero the two parameters. This is not exactly a real life problem, but it illustrates the anomaly well. Let us assume that there is an auxiliary procedure s which already implements the print out and zeroing with one parameter. So the following implementation of p looks quite simple:

```
procedure p (var a, b: integer);
begin
  s(a); s(b);
end;
```

If this procedure is called with the same actual parameter passing twice as formal parameters (for example *p(x,x)*), a and b will denote the same instance within the body of p. What is done with one of them occurs immediately to the other. Until the call *s(b)* is reached, b has already been set to zero. So *p(x,x)* will not

print out twice the value of x at the calling, as we would expect: the second print out will be zero.

Aliases could occur also in less trivial ways. For example, if the above p procedure is called with two array elements, *t[i]* and *t[j]*, there is no alias at first sight – but there is, if i actually equals j. Aliasing can also occur, if the subprogram is awaiting a composite and a scalar typed variable (such as an array and an array element) as parameters, and the second actual is a component of the first. An alias also occurrs if a global to the subprogram variable is passed as a parameter, as in the next example.

```
var global: Integer;
procedure r (var local: integer);
begin
  global := global + 1;
  local := local + global;
end r;
...
global := 1;
r(global);
```

After calling r the value of the *global* variable will be 4.

The emergence of aliases is very harmful. It can easily lead to programming errors, negatively impacting not only the program reliability, but also its clarity, and makes the formal proving of program correctness very hard. One of the basic design principles of Euclid, a language also using parameter passing by address, was to avoid alias emergence consistently, but also OCCAM enforces a strict zero-aliasing.

Parameter passing by result

This parameter passing mode is appropriate for implementing output parameters. The computed result in the formal parameter of the subprogram will be put back into the actual parameter. The formal parameter, just like at parameter passing by value, is a local variable of the subprogram, but at the end its value is copied into the actual parameter. Because of this, the actual parameter has to be an L-value. The formal parameter does not receive the value of the actual parameter at the calling of the subprogram, that is why the information flow is only one-way.

This parameter passing mode is quite seldom – many times it is not even differentiated from the next discussed passing mode by value/result. Nevertheless, in Ada certain parameters are passed by result. For example, the operation reading in a character in the standard *Ada.Text_IO* library looks like this:

```
procedure Get (Item: out Character);
```

The problems of this parameter passing mode will be discussed together with those of the next discussed by value/result mode.

Parameter passing by value/result

This parameter passing mode is usually used to implement in- and output parameters. Basically it is the joint application of by value and by result parameter passing modes. The formal parameter is a local variable of the subprogram, which at the calling of the subprogram becomes the value of the actual parameter, then at the end its value is copied back into the actual parameter. Like in the case of the two previous parameter passing modes that implement output semantics, the actual parameter can only be an L-value here as well, so that it can receive the result computed by the subprogram.

The first language which chose this form of parameter passing was ALGOL W. Please consider the following Ada example.

```
procedure Swap (A, B: in out Integer) is
  Temp: Integer := A;
begin
  A := B;
  B := Temp;
end Swap;
```

On the face of it there is no difference between the Pascal *Swap* procedure (using parameter passing by address) and the Ada implementation (using parameter passing by value/result). Both work correctly. The essential difference between these two parameter passing modes is manifested if the emergence of aliases is examined. For example, in case of parameter passing by value/result the a and b formal parameters denote independent instances during the execution of the *p(x,x)* procedure call. Calling *s(a)* has no effect on b, so the value of x at calling will be printed out twice.

The former example using a global variable would also function differently, if the parameter passing used would not be by address, but by value/result. In Ada this has the following form:

```
Global: Integer;
procedure R (Local: in out Integer) is
...
```

In this case, the value of the *Global* variable would be 3 after the execution of the code snippet.

Unfortunately, there are also problems with the parameter passing modes by result and by value/result. One of the problems is that the actual parameter which could not only be a variable, but an arbitrary L-value, can be evaluated in two ways. The first possibility is to determine the memory address for the value of the formal parameter to be copied back by the subprogram at the moment of

the call, while the second is that it is determined at the moment of the return. For example, if the former r procedure is called with the actual parameter *t(global)* where t is an array, these two possibilities will deliver different results. If the L-value *t(global)* is evaluated before the execution of the subprogram, such as in Ada, the subprogram will change *t(1)* to *t(1)+2*. However, if the L-value *t(global)* is evaluated at the end of the execution of the subprogram, such as in ALGOL W, the subprogram will change *t(2)* to *t(1)+2*. (Please note that in the second case a different instance delivered the initial value of the formal parameter, as which received its final value.) These same two possibilities would occur also in case of parameter passing by simple result, if within the subprogram the *local := local + global* assignment would be replaced with *local := global*. Parameter passing by address does not set this kind of problem: in that case *local* is just an alias to *t(1)* during the whole execution time of the subprogram, so r would change *t(1)*.

Some sources consider the parameter passing by value/result variant, used in Ada, to be different logically from the one that of ALGOL W, and call it as *parameter passing by copy*. In our book we consider, like most of the sources, parameter passing by copy and by value/result to be logically the same, the difference is only considered to be an implementation anomaly.

However, there is still another problem with the parameter passing by result and by value/result. Let us change the body of p and adopt it to Ada like this:

```
procedure P (A, B: out Integer) is
begin
  A := 1;
  B := 2;
end P;
```

Now let us examine the effect of the call *P(X,X)*. In this example it has certainly not much sense to set X to 1 and also to 2, but in a more complex example a similar call can also occur. Using parameter passing by address the meaning is totally clear: during the execution of P X, A and B denote the same instance, so the value of X after returning from the subprogram is determined by the last assignment to this instance. So X would be 2. Using parameter passing by result, the question arises, if A or B should be copied first into X. The definition of a given programming language must clarify this question, because if this stays implementation dependent, the portability of programs written in the given language will suffer.

Textual substitution

Textual substitution is the ”parameter passing mode” used by macros. This cannot even be called parameter passing physically, since the parameter need not be passed to anywhere: the body of the macro is substituted into the location of

the call, so that the formal parameters of the macro can be replaced textually by the actual parameters.

This parameter passing technique also causes interesting anomalies. As we have already mentioned before (see page 300.),the body of the macro and its formal parameters should be put between parentheses to ensure proper expression evaluation order. An example has also been shown that the actual parameters of a macro could be unexpectedly evaluated multiple times (if that formal parameter appears multiple times within the body of the macro), which causes not only efficiency concerns, but could influence the meaning of the program, if the actual parameters have side effects (see page 300.).

Please note that aliases can also occur the same way as in case of parameter passing by address. The examples shown there can all be transferred to textual substitution also. Just as a negative example, consider the following macro:

```
#define M(a) (((a) <= ++n) ? (a + n) : n)
```

This macro uses the variable n at the location of its call. If it is called in the form `M(n)`, the value of n is incremented by one, and the value of the expression would be not `n+(n+1)`, but `(n+1)+(n+1)`.

Much greater problems would arise than the emergence of aliases if the actual parameters and local variables defined within the macro are mixed up.

```
#define MPRINT(x,n) {int i; for(i=0;i<(n);++i) printf("%d ",(x));}
```

This macro can be used as a statement. What happens if a program unit wishes to print out the value of its i variable ten times? The call `MPRINT(i,10)` would print out the numbers from zero to nine, and not the value of i ten times. This is the case because within the body of the macro i as the actual parameter would replace x, but it would be hidden by the local variable named i of the macro according to the declaration hiding rules.

Finally, please note that for some subprograms there is no way to make macro replacements. For example, consider the two parameter swapping *swap* procedure as a macro.

```
#define SWAP(a,b) { int c = a; a = b; b = c; }
```

Let us assume that t is an array, i is a valid index within this array. If the above macro is called in the form `SWAP(i,t[i])`, it would not swap the two parameters. The usual *swap* would properly function with these parameters using the parameter passing by address and by value/result. But in case of textual substitution, if t contains only ones, i is zero, the following will happen. The `c=i` assignment will set c to zero. Then the `a=b` assignment is expanded to `i=t[i]`, so i will be set to the value of `t[0]` which is one. Finally, the `b=c` assignment is expanded to `t[i]=c`, which will set `t[1]` to zero. The goal would have been, to swap the values of i and `t[i]` (that is `t[0]`), from `(0,1)` to `(1,0)`. Instead i is set to one, but zero was not assigned to `t[0]`, but to `t[1]`.

Parameter passing by name

This parameter passing mode can be used again to implement in- and output parameters, but its functioning is much more complicated than the ones that have been mentioned previously. This parameter passing was spread by the appearance of ALGOL 60, at the beginning of the emergence of programming languages. That time many people programmed in assembly languages which supported the use of macros. So, it seemed reasonable to introduce a parameter passing mode which imitated macro substitution. The parameter passing by name is very similar to the textual substitution, but there are differences between them in some important questions.

One odd feature of the parameter passing by name is that it does not function in a single way, but it mixes diametrically different techniques (such as passing by value and by address). Furthermore, it chooses from the different implementation modes based on the form of the actual parameter. To understand the functioning of a subprogram that uses such a parameter passing is much harder than of the subprograms containing/introducing aliases.

If a 'by name' formal parameter is matched with a literal or a constant expression (for example *4+2*), then the actual value is passed. But if a scalar variable is matched, then it is passed by address. Now comes the next oddity: if the actual parameter is an expression, it will be evaluated every time when a reference is made to the formal parameter within the subprogram body. So, if the actual parameter is an expression which has some components changing during the execution of the subprogram, the formal parameter will have different values according to these changes.

If, for example, the actual parameter is the expression *t[i]*, and the value of *i* has changed at one point during the execution of the subprogram, the formal parameter will denote, from that point on, another array element according to the new value of *i*. The situation is different if the actual parameter is an expression in the form *a+b*, and *b* is modified by the subprogram, so the value of the formal parameter would also change, according to the new value of *b*.

Consider the following ALGOL 60 example famous by the name "Jensen's device" [Sco09] which can compute a sum expression:

```
real procedure sum (expr, i, low, high);
  value low, high;
  real expr;
  integer i, low, high;
begin
  real rtn;
  rtn := 0;
  for i := low step 1 until high do rtn := rtn + expr;
  sum := rtn;
end sum
```

The first two parameters of the function (the real *expr* and the integer i) are passed by name (this is the default parameter passing mode in ALGOL 60), the other two (integer) parameters are passed by value. Now let us compute the value of the following expression:

$$y = \sum_{x=1}^{10} 3x^2 - 5x + 2$$

For this only the above function must be called with the parameters shown in the formula:

*y := sum (3*x*x − 5*x + 2, x, 1, 10);*

x is a scalar variable, so it is passed essentially by address to the formal parameter i. In the subprogram i and x denote the same instance. By running the loop from one to ten (these values are passed into *low* and *high*), the value of x is running through this interval. The expression to sum (the actual parameter in *expr*) is evaluated in the loop body again and again, always using the actual value of x.

Please note the similarity between the parameter passing by name and textual substitution. Independently from the form of the actual parameter, the formal parameter will always behave as if it was replaced in the program source by the actual. If the actual parameter is a constant expression, the value of the formal parameter stays the same: the value of the constant expression. If the actual is a scalar variable, the formal parameter will denote always the same instance; it is like using the actual parameter within the source of the subprogram. Finally, if the actual parameter is an expression containing variables, then every reference to the formal parameter causes the new evaluation of the expression, as if the actual parameter occurred in place of the formal parameter. In Section 7.5.3. we will show that parameter passing by name is not simply a smarter variant of the textual substitution, but it differs in its important aspects.

If using parameter passing by name, a significant part of the problems experienced by textual substitution will not occur. Nonetheless some problems (such as the emergence of the aliases) will remain. The fact that some very simple operations cannot be described by using parameter passing by name, still causes trouble. The *swap* procedure, that has been mentioned several times cannot be implemented with parameter passing by name either.

7.4.2 Comparison of parameter passing modes

The parameter passing by value is the clearest from all the above mentioned. If during programming only by value parameter passing and functions without side-effects are used, the ready program product will be clear, well readable and maintainable, furthermore, it will be easily manageable by mathematical tools

(such as proving of correctness, program transformations). It is not a coincidence that functional languages are usually praised just because of this.

Many programmers think that procedures in imperative languages enable them to write the most programs more efficiently than with just functions. Procedures can pass their computed results to the caller through output parameters. This cannot be modeled with parameter passing by value: it can only be used for input parameters. (For by value parameters the usual *swap* procedure would, of course, not work.) For passing output, and in- and output parameters by address, by result, by value/result and by name modes can be used.[18] Parameter passing by name is a very flexible technique. Its principle is late binding. The actual parameter is not bound to the formal parameter at the calling of the subprogram, but later, during the execution of the subprogram. Furthermore, this binding is done again and again within the body, every time when the formal parameter is referenced. This results in great flexibility, and dynamic adaptability during runtime. The principle of late binding has the same effect on other areas of the programming languages. Consider those object-oriented languages where the implementation of polymorphism is mainly based on the late binding of the operations (methods, messages) to the object in runtime (see Chapter 10.). The lazy evaluated functional languages (see Chapter 15.) or the "short circuiting" logical operators in the imperative languages follow the same principle.

Despite its great flexibility, parameter passing by name could not really become widespread. The primary reason for this is because of its complexity. In case this technique is used the programs often become hard to understand. Its semantic problems (consider, for example, the unimplementabilty of *swap*) are also against it. The implementation of this parameter passing is also not easy: in ALGOL 60, for example, for the by name actual parameters a parameterless procedure (thunk) is compiled which has the job to evaluate the actual parameter. In fact this procedure is passed to the subprogram which will call it every time the formal parameter is referenced. (We will discuss the implementation difficulties of subprograms passed as parameters in more detail in Section 7.7.1.). This short description also reflects that parameter passing by name is not very efficient. In fact, this is the least efficient amongst all the parameter passing modes. Please note that in practice there are only a handful of problems where using parameter passing by name would result in such a brilliant solution, as in the *summarize* function shown previously; and all of these can be usually rewritten elegantly by passing a subprogram as a parameter.

For modeling the output parameters accordingly, parameter passing by address and by result, or by value/result are usually chosen by the designers of the programming language. The last two are chosen primarily to avoid the aliases, and they are semantically clearer, as well as easier to use as the by address

[18] Textual substitution is not even mentioned here: this technique causes so much semantic problems that it is not used to implement parameter passing for subprograms, just for macros.

parameter passing. But as we already mentioned, this is not the only aspect which must be taken into consideration at designing a language. Efficiency is also an important question.

Parameter passing by value, result, and value/result are collectively referred to as *data transferring* passing. This naming indicates that a copy of the actual parameter will be created within the called subprogram, and into/from this data must be copied: the value of the actual parameter, or the computed result. This data transfer can be very costly in memory and in execution time. For big parameters, such as arrays and matrices, parameter passing by address is usually much more efficient: only the address of the actual parameter, which is only a machine word, must be stored into the formal parameter at the time of calling the subprogram.

The situation is further complicated since the efficiency is not only determined by the amount of administration and data transfer needed at calling the subprogram and returning from it. It is also an important aspect how efficiently the used data can be accessed from within the subprogram. In this sense the data transferring parameter passings are better: only the memory location of the copy for the formal parameter must be known. In case of parameter passing by address there is an additional indirection: after looking up the formal parameter in memory, its content must be used as a reference to the actual parameter to work with. If there are many references within the subprogram to the formal parameter, and the actual parameter is not quite big, then data transfer by parameter passings is much more efficient than that by address.

For large parameters parameter passing by address is so much more efficient than data transferring that it can even reach the efficiency of parameter passing by value, in its aspect of modeling a purely input parameter. Pascal programmers are taught that **var** keyword should be used in front of the formal parameter, if the actual one should be changed, *or* the parameter is very large, for example, an array [Sco09]. Of course, this kind of approach can easily result in writing faulty programs: to place efficiency in front of the semantics is always dangerous. At parameter passing by address, if – unwittingly, but – unintentionally the input kind of actual parameter is changed, the compiler cannot warn us for this error. Likewise, if a ”purely output” parameter is implemented with parameter passing by address, the compiler will not give a warning, if the actual parameter is used within the computation!

In the next section some interesting programming languages are chosen to demonstrate how the above mentioned problems are usually solved.

7.4.3 Parameter possibilities in some programming languages

All the parameters in FORTRAN are in- and output parameters. The language does not determine which parameter passing mode must be used by the compiler to implement this, the by address or the by value/result mode. Nearly all the implementations before FORTRAN 77 use exclusively parameter passing by

address, but newer implementation usually use by value/result passing for scalar parameters. If a FORTRAN programmer would like to use an input parameter, it must be manually ensured not to change the formal (and through it the actual) parameter. For example, a local variable could be defined within the subprogram which would receive the value of the formal parameter. Within the subprogram this local variable can be modified without any consequences.

In contrast to other languages, the FORTRAN compilers do not count it as an error if an in- and output actual parameter is a right value (such as a numeric literal, or not an L-value complex expression). If this was considered to be an error, it would be hard to use input parameters in the language. All right, but in case of a right value how will it be passed by address? In case of complex expressions, the compiler will define a temporary variable which will hold the value of the expression. This temporary variable will be passed by address. If the subprogram changes the value of the formal parameter despite the planned input semantics, the temporary variable will be changed. The programmer, of course, will not have access to this temporary variable; the computed result by the subprogram in the formal parameter will be inaccessible.

What happens to the literals, if they are passed by address or by value/result? The compiler stores the literals within the program in the same way, as variables. In case of parameter passing by address for example the memory location will be passed where the literal is stored. If the subprogram again violates the assumed input semantics, and changes the formal parameter, the compile time constant would change, which could be fatal for the further functioning of the program. Many FORTRAN IV implementations were not prepared for this problem. In programs compiled by such a compiler, it could happen that after a `SWAP(0,1)` call, the `0` and `1` constants were really swapped. After this, if on some later points of the program the `X = 1` statement occurred, the `X` variable was not set to one, but to zero.

The next high-impact language, ALGOL 60 uses parameter passing by name as a default, and by value if the formal parameter is introduced by the *value* keyword. There are not many languages which took parameter passing by name from ALGOL 60. One, which did it is the fairly widespread language, SIMULA 67. In this language the default parameter passing mode is already done by value, and the by name method can be chosen.

Because of the problems of parameter passing by name, in the ALGOL W language the in- and output parameters are passed by value/result. The language Ada also uses this parameter passing in certain cases. This will be discussed later.

In ALGOL 68 and Pascal the input parameters are passed by value, in- and output parameters by address. (The default is the by value; in case by address is used it must be indicated separately.) In COBOL the parameter passing by value and by address can also be chosen by the programmer for the nested programs.

In C, basically, there is only one kind of parameter passing: by value. To implement in- and output parameters the programmer usually has to pass an address, i.e. a pointer – by value – explicitly:

```
void swap (int *a, int *b) {
   int c = *a;
   *a = *b;
   *b = c;
}
```

This subprogram can be used in the following way to swap two integer variables:

```
int a = 2, b = 4;
swap (&a,&b);
```

The exceptions to this rule are arrays which are in C very similar to pointers. Arrays are passed by address:

```
void sort (int t[]) { . . . }
int t[] = { 3, 4, 1, 2 };
sort (t);
```

One of the novelties of C++ compared to C was that it introduced a special kind pointer type, the so-called *reference type*. This type also makes the usage of in- and output parameters easier. Parameter passing by address is done through reference types. Explicit marking of pointers are not necessary within the subprograms, nor at calling:

```
void swap (int &a, int &b) {
   int c = a;
   a = b;
   b = c;
}
int a = 2, b = 4;
swap (a, b);
```

Consider another example based on [Sco09]. Reference types can also be used to specify return values. This is especially useful, if a function should return such a value which would not make sense to assign to. Such could be for example a file. Of course, a returned pointer can reference anything, even a file, but by using pointers the reference must be explicitly dereferenced, and this could be pretty uncomfortable. In C++ for example the operator << can be used to write to an output (file, standard error, etc.). This operator returns a reference to its first argument. This allows to ”chain” the << calls to an output.

```
cout « a « b « c;
```

If there were no reference types, the << operator would have to be implemented by pointers. In this case the call would look like this:

```
*(*(cout « a) « b) « c;
```

C++ and the ANSI C contain a very interesting language element: in the definition of the formal parameters **const** keyword can appear. In the example below, this keyword can be used to signal that the formal parameter should be a pointer referencing a constant.[19] This means that the instance referenced by the formal parameter cannot be changed by the subprogram. The actual parameter does not need to be a pointer referencing a constant: the compiler can automatically convert it according to the proper type of the formal parameter. Consider T being a type the following function definition and call.

```
int f (const T *p) { . . . }
T x = . . .;
int i = f(&x);
```

Even if variable x is passed to function f mimicking parameter passing by address, it cannot change the value of x. Semantically x became an input parameter of f. If the values of type T are large, this trick helps avoiding data transferring which would occur at parameter passing by value: instead, only the costs of passing by address, which is usually more efficient, would occur at subprogram call. The advantages of this technique are even better visible, if combined with the reference type of C++. In this case an input parameter is implemented purely with parameter passing by address.

```
int f (const T &p) { . . . }
T x = . . .;
int i = f(x);
```

Please do not think that this technique will "cheaply" ensure the same semantic safety which is provided by parameter passing by value. Because passing by address is used, aliases can still occur. Although the subprogram cannot change the actual parameter through the formal p parameter, within the body of the subprogram it cannot be known if the same has been done through another reference. At parameter passing by value it is always guaranteed that the value of the formal parameter will not change during the execution of the subprogram, unless it is explicitly done by the subprogram. Consider the following example for this: here the resulting output will not be *42* twice, but first *42*, then *43* will appear.

```
void g (const int &p, int &q) {
  cout « p;
  ++q;
  cout « p;
}
int x = 42;
g (x, x);
```

[19] If a declaration should denote variable r to be a constant pointer, it would be written like this: $T*$ **const** r.

At parameter passing by value the formal parameter can be used as a fully functional local variable, even a value could be assigned to it (of course, without having any effects on the actual parameter). This cannot be done in case input parameters are implemented by **const** keyword and passed by value.

In other languages there are also similar constructs to **const**. In Modula-3, for example, the formal parameter can be declared as **READONLY**. The compiler will ensure that the formal parameter will not appear on the left side of an assignment, and will not be passed in an in- and output manner as the actual parameter to another subprogram. The input parameter, declared as **READONLY**, will usually be passed by value, if it is small in size, and by address, if it is large. If a right-side expression is passed as the actual to a **READONLY** parameter by address, the compiler will – just like in FORTRAN – use a temporary variable to store the value of the right-side expression, and its address will be passed to the formal parameter.

In Ada (and other similar languages) a semantically clearer approach is used. The programmer is not specifying parameter passing modes, but semantic modes for the formal parameters. Parameters specified as for **in** mode are input, as for **out** mode are output, and as for **in out** mode are in- and output parameters. The actual parameter passing technique used is not relevant for the programmer: such implementation details are left for the compiler. According to the rules of Ada an **in** mode parameter cannot be changed by the subprogram, not by assignment, nor by passing it to a subprogram as an **out** or **in out** mode parameter. Likewise, **out** mode parameter cannot be read by the subprogram (in Ada 83, not at all, in Ada 95 yes, until it was already assigned a value). For **in out** mode parameters, of course, everything is allowed.

For scalar and pointer typed values Ada uses the appropriate data transfer parameter passing, for composite typed values the language definition does not determine, if data transferring or parameter passing by address should be used; compilers can differ from each other in this respect. (For example, for access types by address mode it is obligatory, but not for records.) This will theoretically not cause any portability problems, because Ada defines a program erroneous, if its result depends on which parameter passing mode the compiler uses. However, compilers are not forced to notice this kind of erroneous conditions and prevent it with a compilation error.

There are some interesting differences between various languages regarding formal parameters which are declared with subtypes [Sco09]. In Pascal, for example, if an interval typed parameter should be passed by address, the type of the actual parameter cannot be narrower than the type of the formal parameter. Otherwise, it could happen that the subprogram executes an operation which delivers a result not passing into the actual parameter.

```
type onetohundred = 1..100;
var a: 1..10;
    b: 1..1000;
```

```
procedure p (var n: onetohundred);
begin
  n := 100;
end;
...
a := 5; b := 5;
p(b);  (* This is O.K. *)
p(a);  (* This causes a compilation error. *)
```

In Ada such a parameter would be passed with data transferring. Both calls would be valid, since within the subprogram the assignment to the formal parameter (managed as a local variable) will not cause any problems. Nonetheless when the subprogram ends, the value of the output parameter must fit into the subtype of the actual parameter. If this is not the case, a runtime error (a dynamic semantic error) is raised in form of a predefined exception.

In Java parameter passing by value is applied. As objects can be accessed through implicit references, and passing an object as a parameter means passing the reference by value, so a subprogram can change the object received as the actual parameter, or to be more precise, the object which is referenced by the reference received as the actual parameter. (This is usually called *parameter passing by sharing*. This name is originated from CLU.) Of course, the value of the actual parameter, that is the reference itself cannot be changed by the subprogram. In the next example the call `f(a,sb)` would only append the string `"hello!"` to the object referenced by *sb*.

```
int a = 5;
StringBuffer sb = new StringBuffer();
void f (int x, StringBuffer s) {
  ++x;
  s.append("hello!");
  s = new StringBuffer();
  s.append("bye!");
}
```

C# uses a much more complex parameter passing model. By default, it functions in the same way as Java: at passing so-called reference types – classes, interfaces, delegate types and arrays – the implicit reference is passed by value, and for so-called value types – primitive types, enumeration types and **struct** types – the values themselves are passed. To deviate from the default mechanism, one can use by address and output semantic parameters. Parameter passing by address must be signaled by the **ref** keyword in front of the formal and of the actual parameter. (This has the same form as the one used in C when parameter passing by address was mimicked.) By using **ref**, reference and value typed actual parameters can be changed by the subprogram. (In case of a reference typed parameter, not only the referenced object, but the reference itself!)

At last, using the **out** keyword (which must also be placed in front of the formal and the actual parameters) parameter passing by address can be achieved where the compiler will ensure the output semantic. (Namely the actual parameter need not be initialized, because the formal parameter will not be initialized automatically, either. The value of the formal parameter cannot be read before a value is assigned to it. The subprogram cannot return without setting a value to the output parameter. After returning from the subprogram, the actual parameter counts as initialized.) Please be aware that passing a reference typed parameter by value is not the same like passing a value typed parameter by address. Consider *C* being a class, with one data member, the integer typed *i*. Consider furthermore *S* being a **struct** type with the same structure.

```
void g (C c, ref S s) {
  c = new C(); c.i = 1;
  s = new S(); s.i = 1;
}
C a = new C(); a.i = 0;
S b = new S(); b.i = 0;
```

The call *g(a,* **ref** *b)* will not modify the reference *a*, but the value of the variable *b* will be changed, the value of its *i* field will be set to one.

In Eiffel the subprogram cannot change its formal parameters, in the same way as in Ada the **in** mode parameters cannot be changed. In Eiffel – like in Java or C# – two kinds of types are differentiated: reference types and so-called expanded types. Every type has an expanded and also a reference variant.

If the formal parameter is of some reference type, the formal parameter cannot be assigned a new value (new reference), but the referenced object (its fields) can be changed within the subprogram. If both the actual and the formal parameters are of some reference type, parameter passing by address is realized: the formal parameter is referencing the same object as the actual. If both the actual and the formal parameters are of expanded types, parameter passing by value is realized: the actual parameter is copied into the formal parameter. If the actual parameter is of reference type, the formal parameter is of expanded type, the actual parameter (its referenced object) is copied into the formal parameter. This is often called dereferencing. In this case, if the actual parameter does not reference any object, i.e. its value is **void**, a runtime exception will occur. The last case is, if the actual parameter is of expanded the formal parameter is of reference type. In this case the actual parameter is copied into the referenced object by the formal parameter.

In CLU the situation is also very similar. Types are categorized as mutable and immutable. Mutable typed values contain object references, immutable typed values contain actual objects. Primitive types are immutable types, but type constructions have their mutable and immutable variants. Mutable objects are passed as parameters by sharing, immutable objects by value.

At the end, the parameter passing mode of the lazy evaluated functional languages must be mentioned. In these languages parameter passing takes place, as if parameter passing occurred by value. The difference is that the evaluation of the actual parameter and the determination of the value of the formal parameter are not carried out at the calling of the function, but later, when the value of the parameter is needed for the first time. This kind of parameter passing is called *by need*.

Parameter passing by need is a bit like by name, because of the principle of late binding. But opposed to parameter passing by name, the value of the actual parameter will only be evaluated once. So, if there are no side effects during the evaluation, there would be also no difference between these two parameter passing modes.

7.5 Environment of the subprograms

In this section it will be examined how subprograms fit into their environment, the whole program. The most important relevant scope and visibility rules will be mentioned, and the issues of nesting and separate compilation will be discussed. For more details please see chapters 4 and 9.3.

7.5.1 Separate compilability

Only those languages are suitable for developing large, long life software systems which enable the separate compilation of different parts of the program without the need of full recompilation [Seb13]. This possibility is really crucial for supporting development in teamwork, and for the efficiency of coding, the reusability of the software and the less costly changeability. Separate compilability of the program parts allow to only recompile those part of the system which were changed in the latest step of software development or maintenance.

Subprograms offer a natural separation layer for those languages which support *compilation units* at all that is the separate compilation of program parts. In ASA FORTRAN, for example, the compilation unit and the subprogram are essentially coinciding terms.[20] In C a subprogram can be a separate compilation unit, but a compilation unit can consist of multiple subprograms (and also variable and type declarations). In Ada it is the same: a subprogram can appear as a compilation unit, and there are also other program units which can occur as compilation units. Nonetheless there are also languages where the subprogram cannot be a separate compilation unit: Modula-2, Clean, Java etc. only support higher structural levels, such as the module or class to form compilation units.

In some languages, such as FORTRAN II or the early variants of Pascal, *compilation units* could only be whole programs. Separate program parts (subprogram or other program units) cannot be compiled on their own. These

[20] Except the main and data segments which are also compilation units.

languages are essentially unusable for developing serious applications. The later versions of FORTRAN and Pascal tried to put an end to this serious deficiency. In ASA FORTRAN (and for example in C) independent compilation is already supported. This means that the application can be assembled from multiple compilation units, but these are completely independent from each other, and do not have information about the others. The disadvantage of this technique is that there is no type checking at cross referencing compilation units. In ASA FORTRAN, for example, the subprograms (procedures and functions) are separate compilation units (called segments in FORTRAN terminology). There is no language feature which could enable communication between two segments. So, when calling a subprogram the number or types of the actual parameters do not match with that of the formal parameters, the compiler cannot catch this error.

In PL/I by calling subprograms in the same compilation unit, the compiler will check the actual and the formal parameters, but when calling (so-called external) subprograms from a different compilation unit it is not able to do this. To increase the safety of the programs, PL/I supports the declaration of an external subprogram that is its specification can be given in the calling compilation unit. This specification, which is given at the declaration can be checked at all calls. Unfortunately, the compiler is still unable to compare the specifications from the subprogram definition and from the declarations in other compilation units.

In C those functions that are defined in other compilation units and returning non-**int** typed values, must be declared. The compiler is still unable – such as in PL/I – to compare these declarations with the definition of the subprogram. For non-declared external subprograms, C assumes a **int** typed return value, and demands it when using the subprogram. The return type from the (external) subprogram definition does not matter for the C compiler. The situation is further complicated since in the original C the type of the parameters were not included in the specification of the subprogram. In ANSI C if no prototype is used, that is the type of the parameters are not given, the number and types of the actual parameters get also not checked. So, if a declaration is written, and it does not contain parameter types, the usage of the subprogram (its calling or passing its address as a parameter) will not be type checked properly. The – nearly perfect – solution is the usage of header files. The declaration of the subprogram is stored in a header file, which will be loaded in the definition, and also in the caller compilation units by the **#include** statement. This method can prevent a lot of errors, only it must be assured that the definition and all caller compilation units load the same header file, and all the functions usable from other compilation units must be declared in the header file.

In newer languages, such as FORTRAN, Modula etc. usually it is possible to define *separate compilation* units. In these languages – in contrast to the independent compilation – the connections between compilation units must be described. In Ada the compilation units specify the elements (objects, types,

subprograms etc.) accessible from the outside, and describe some information about them, which will enable their type correct usage from other compilation units. In Ada, for example, a subprogram can be a separate compilation unit. If this subprogram should be called from another compilation unit, then it must be made accessible for the caller. For this the **with** statement can be used which must be placed at the beginning of the calling compilation unit.

```
with Swap;
procedure Sort (A, B: in out Integer) is
begin
  if A > B then Swap(A,B); end if;
end Sort;
```

The **with** statement signals the compiler that the *Swap* reference within the *Sort* procedure must be matched with the specification of the *Swap* implemented as a separate compilation unit.

7.5.2 Embedding

ALGOL 60 did not support the decomposition of the program into multiple compilation units, but introduced another very important concept: the block structure. This means that program units can be nested in each other. Languages support block structures for different extents. In ASA FORTRAN, for example, there is no possibility to say embed two subprograms in each other. Neither C is called a block structure language, although its nesting block statements show some similarities with the nested subprograms. In the C example below in the block embedded within the body of the *f* subprogram a variable *j* is also declared which hides the variable with the same name from the nesting block.

```
void f (int p) {
    int i = 1, j = 1;
    {
        int j = 2, k = 2;
        printf ("Embedded block. %d %d %d", i, j, k); /* 1 2 2 */
    }
    printf ("Nesting block. %d %d", i, j);     /* 1 1 */
}
```

The more advanced languages support not only subprograms, but also modules and classes for arranging the program source. The subprograms are important building blocks of these language structures that enable the higher level arrangement. In Java, classes can be nested, even into subprograms, but subprograms cannot be nested. In Clean the opposite is the case: modules cannot be nested within modules, nor in subprograms, but subprograms can be nested. Lastly, there are languages which support nesting ”everything into everything”. Such are for example Modula-2 or Ada.

```
procedure Sort (A, B: in out Integer) is
  procedure Swap (X, Y: in out Integer) is
    Z: Integer := X;
  begin
    X := Y; Y := Z;
  end Swap;
begin
  if A > B then Swap(A,B); end if;
end Sort;
```

Nesting of program units is the basis for the scope and visibility rules. For example, if a subprogram is nested into another subprogram, the embedded subprogram is only usable within the nesting subprogram. With this method, the number of connections between program units, and so the program complexity can be reduced.

The embedded subprograms cannot only access the variables (and other entities such as subprograms, types) defined within them *locally*. They can also use the entities defined in the nesting block, or in its nesting block and so forth, recursively. Finally, they also can use the entities (modules, classes, subprograms), which are *global*, that is not nested in anything. The entities accessible from a subprogram, but not defined there locally are usually called *non-local* entities. In the previous Ada example the *Swap* procedure can use the *A* and *B* variables, because these are non-local variables of it.

Through common non-local variables the subprograms can exchange information with each other. This is usually a very efficient possibility but greatly increases the complexity of the code. Modularization, and so the readability and maintainability of the program is supported, if the subprograms communicate with each other through their parameters. By exchanging information through non-local variables, the efficiency of the program can be increased. Parameter passing is costly, mainly in case of data transferring parameter passing modes. By reasonably applying subprogram nesting and communication through non-local variables, the efficiency of the program can be increased without increasing the complexity and violating the principles of modularity. The previous Ada procedure could be modified to let the *Sort* and *Swap* communicate through common variables instead of parameters:

```
procedure Sort (A, B: in out Integer) is
  procedure Swap is
    Z: Integer := A;
  begin
    A := B; B := Z;
  end Swap;
begin
  if A > B then Swap; end if;
end Sort;
```

7.5.3 Static and dynamic scope

The majority of the programming languages use static (or lexical) scope rules. This means that scope is based on the nesting of program units. A declaration is valid exactly only until the end of its embedding program unit: outside of this, it is not valid, but it is valid also within the following nested program units. That is why an embedded subprogram can access the variables of the nesting subprogram.

There are also some languages (many LISP-variants, SNOBOL, APL or Perl) which use dynamic scope rules.[21] This means that the scope of a declaration will not only cover the contained, but also the called program units. If a variable is declared in the subprogram A, then the subprogram B is called, this B subprogram can use the variable, even if B is not nested within A. In the following ALGOL-like example the integer variable I is increased by one, if static scope is applied, or in case of dynamic scope, the real variable I will be increased.

```
begin
  integer I := 1;
  procedure P is
  begin
    I := I+1;
  end P;
  begin
    real I := 1.0;
    P;
  end;
end;
```

It is not a coincidence that most of the languages use static scope rules. A program is much easier to understand with static than with dynamic scope applied. Furthermore, the static type checking also requires the static scope rules. There are still tasks which can be solved very elegantly and efficiently by applying dynamic scope.

Concerning subprograms, the principle of static and dynamic scope can explain the difference between parameter passing by name and textual substitution. In a language applying static scope rules, subprograms using parameter passing by name comply with the rules of static scope, but macros using textual substitution do not. The actual parameter of the subprogram passed to the formal parameter will be evaluated again and again in the environment of the caller when the formal parameter is referenced, but the actual parameter for the macro will be evaluated in the environment of the macro body. (This strange behavior was already shown in the *MPRINT* example on page 308.)The body of the

[21] The newer versions of LISP already use mainly static scope. Originally there was only dynamic scope in Perl, but the recent language lets the programmer choose between dynamic and static scope rules.

subprogram is executed in the place of its definition, in the environment which is determined by the program source lexically (statically) for the subprogram. In contrast to this the body of the macro is evaluated in the environment valid at the place of the call that complies with the dynamic scope.

Before demonstrating the above idea through an example, an alternative definition of the parameter passing by name in ALGOL 60 will be given:

- Actual parameters will be textually substituted into the places of the formal parameters, but without conflicting variable names within the actual parameters and the locally defined variables in the subprogram. If there is a variable within the actual parameter with the same name which exists as a local variable in the subprogram, then this local variable and all the references to it must be renamed.
- The resulting subprogram body must be substituted into the location of the call. If there are non-local variables in the subprogram with the same name as local variables have in the caller, then these local variables (and all the references to them) must be renamed.

Consider the following ALGOL example:

```
begin
  integer n;
  procedure P(x);
  begin
    integer i;
    i := n;
    print i;  print x;
  end;
  n := 1;
  begin
    integer n; integer i;
    n := 2; i := 3;
    P(i + 1);
  end;
end;
```

During the first step of the calling, *i+1* is substituted into the places of x, which will cause the i in the actual, and also declared within P to conflict. The solution is to rename the local variable, to, say, j. Afterwards the body of P will look like this.

```
begin
  integer j;
  j := n;
  print j;
  print (i + 1);
end;
```

During the second step, this body it substituted at the call location. Another naming conflict will occur, since there is an n within the body of P, but the n variable at the location of the call denotes something else. So, at the location of the call n must be renamed to m. After this, the program will become the following:

```
begin
  integer n;
  n := 1;
  begin
    integer m; integer i;
    m := 2; i := 3;
    begin
      integer j;
      j := n;
      print j;
      print (i + 1);
    end;
  end;
end;
```

The program will print out the following: `1 4`.

What would happen if an "equivalent" C program is written, but instead of the above subprogram that uses parameter passing by name, a macro would be used? As usual, the formal parameter of the macro is well bracketed.

```
int n = 1;
#define P(x) { int i = n; printf("%d %d", i, (x)); }
int main () {
  int n = 2;
  int i = 3;
  P(i + 1);
  return 0;
}
```

After substitution, the following code is generated:

```
int n = 1;
int main () {
  int n = 2;
  int i = 3;
  { int i = n; printf("%d %d", i, (i+1)); };
  return 0;
}
```

This program will print out the following: `2 3`.

7.5.4 Lifetime of the variables

The local variables and formal parameters of the subprogram usually live until the end of the subprogram. The local variables and formal parameters of the subprogram are created at every call, and at the end of the subprogram they are destroyed. The C code snippet below demonstrates a typical error. According to the compiler, this is a valid function, but after calling it and the integer referenced by the returned pointer is used, the result might be that the number does no longer exist: it was destroyed at leaving the function.

```
int * do_not_call () {
  int i = 1;
  return &i;
}
```

In some languages there are exceptions to the above rule. In ASA FORTRAN, for example, the local variables of the subprogram are static, that is their lifetime will last for the whole program. In other languages, such as PL/I, CLU, C or C++ static lifetime local variables can be declared with a specific keyword.

```
int next () {
  static int i = 0;
  return ++i;
}
```

The *i* variable declared in the *next* function has a static lifetime. After leaving the function it is not destroyed, and even its value is preserved until the next call. That is why calling the *next* function will return always the next proceeding integer value. The following program has the same effect:

```
int j = 0;
int next () {
  return ++j;
}
```

The difference between the two approaches is that the first one is more modularized. The scope of the *i* variable, which is defined there, will not cover the whole program, as the *j* variable in the second example, since it is limited only to the *next* function. Outside of the function this *i* variable is not accessible.

In Java – using nested classes – not only local variables, but actual parameters can also ”survive” the execution of the subprogram.

```
Object nameIt (final String forename) {
  final String surname = "Smith ";
  return new Object() {
           public String toString() { return surname + forename; }
         };
}
```

If this method is called with an actual parameter, an object is returned which will remember the actual parameter, even long after the return of the method. Having *pete* printed out, its *toString* operation will be called which will look up the actual parameter and local variable of the former method call which created *pete*, and the resulting *String* object will be computed based on these values.

```
Object pete = nameIt("Pete");
System.out.println(pete);
```

7.6 Overloading subprogram names

Some languages allow that at some point of the program the same subprogram name can denote multiple subprograms. If a subprogram is called by that name, the compiler will find out from the context of the call, which subprogram should be executed. If the compiler is unable to determine exactly which subprogram by the same (overloaded) name should be called, it will generate a compilation error. Also, if there is no overloaded version which would match the call, a compilation error will be thrown again.

Accordingly, the selection between the overloaded subprograms is done during compilation time in a language using static type checking. The compiler must make its decision based on the information needed to call the subprogram. What can be this information? This depends on the given language. For example the compiler can select one or the other subprogram based on the type of the actual parameters. Surely, the information for this decision comes from the specification of the subprogram, since the caller program unit only ”knows” the called specification. Usually it is said that the ”signature” of the diverse overloaded variants must be different. But the signature has a different meaning for the various languages.

In Ada the subprograms can be called with qualified name. The name of that package is used for qualification contains the definition of the subprogram If in two different packages subprograms with the same name are defined, the qualified name helps distinguish between them which should be called. If the subprogram should not be called by qualified name, or the subprograms have the same qualified names (that is they are defined within the same package), there must be some other differences at the location of the call. This can be, for example, in the number of actual parameters, or their types. For further differentiation the return value (if any that is if the subprogram is a function), and also the type of the return value can be used. Although the name and mode of the formal parameters are part of the subprogram specification, they cannot be used to differentiate between overloaded subprograms.

```
function A return Integer;
function A return Boolean;
```

```
procedure A;
procedure A (S: String);
procedure A (I: in out Integer);
procedure A (I: in Integer; J: in Boolean := True);
procedure A (V: Natural := 42); -- this is troublesome!
```

The first six subprograms are valid overloadings of the name *A*. The seventh subprogram cannot be defended in addition to the first six, since it does not differ enough from the fifth. Calling one of the first six, the context of the call will nearly always determine which variant should be executed. However, such calls are possible, when this is not the case. If *K* is an integer variable, the call *A(K)* is ambiguous: it can refer to the fifth or the sixth subprogram. Furthermore, the Ada compiler will handle the call *A(2)* also ambiguously, although this would not match the fifth subprogram.

In Java and C++ subprograms with a declared return value can be called by omitting it, so the type of the return value was not included into the signature in these languages: if two subprograms only differ from each other by this, then they cannot be seen as overloaded variants. The subtype relation defined in object-oriented languages makes the rules of overloading even more complex. Because of subtypes, an actual parameter can match multiple types at the same time. Consider the following Java code snippet:

```
void m(Object o) { . . . }
void m(String s) { . . . }
```

The call `m("Hello")` would match both overloaded variants, but the second is "better": so the compiler will see this situation as straightforward, and the second subprogram will be called.[22]

In the Clean language overloading is used differently: overloaded variants can only differ very little, and overloading must be signaled separately. Overloading in Clean is a feature for passing types as parameters to subprograms, so it resembles the notion of the template in Ada or C++. (See Chapter 11.)

Why would two subprograms be named the same way? Primarily this is a good possibility only if the subprograms do, in fact, nearly the same. For example, the subprogram sorting an integer array could be named as *sort*, and also the variant sorting a real array.

According to this, many languages offer predefined overloaded subprograms for the programmers. This is the case for Ada, C++, Java, and even for languages (ALGOL 60, FORTRAN etc.) which do not allow the programmers to write overloaded subprograms. For example, in the standard library of Ada there are procedures with the name *Put*, which print out strings, characters, or integer numbers to the standard output or into a file.

[22] To force the first variant, the following syntax can be used: `m((Object)"Hello")`. With this typecast (see Section 5.1.4.) the static type of the actual parameter is changed to *Object*, so the second subprogram will no longer match the call.

Another field of the application of overloading is the assignment of default values to formal parameters. In Java, for example, the following code would be similar to that discussed on page 293.:

```
void draw_rectangle (int width, int height) {
  draw_rectangle(width, height, 0, 0, 0);
}
void draw_rectangle(int width, int height, int x, int y, int color){. . .}
```

The disciplined usage of overloading has a positive effect on the readability and understandability of the program, because it is a feature of data abstraction and polymorphism (see Chapter 11.). Nonetheless its careless and clumsy usage could easily increase the complexity of the code. Additionally, the interference with subtypes, default parameter values and automatic type conversion causes often trouble.

7.6.1 Operator overloading

Operators are such subprograms which can be called with special syntax (see 7.6). For example, for addition in most of the languages the + operator can be used, which is usually called in contrast to subprograms not in prefix, but in infix form. Many languages offer for their predefined types predefined (often overloaded, such as the + for integers and for reals) operators, but do not allow the programmer to define operators with similar semantics for user types. Such a language is, for example, Java which otherwise allow normal subprogram overloading. In contrast, in C++ or Ada, if a matrix type is written, an infix + operation can also be defined:

```
function "+" (A, B: Matrix) return Matrix is ...
```

If X and Y are matrices, the expressions $X+Y$ and `"+"`(X,Y) are both valid. Operators defined in this way inherit the precedence of the operator defined by the language.

In Ada it is an important limitation that operators defined by the language can be overloaded, but no new operators can be defined. In ML, Clean etc. also this is supported. Please consider the following example to implement the exponentiation (x to the n):

```
infixr 8 tothe;
fun x tothe 0 = 1.0
  | x tothe n = x * (x tothe (n-1));
```

This ML function called `tothe` [Sco09] is an infix operator which is associative from the right (this is signaled by the letter r in the keyword `infixr`), and has a precedence level of 8. FORTRAN 90 also allows the definition of new infix operators, but requires to enclose the operator between dots (such as `A .cross. B`), and every such operator is assigned the same precedence level.

Finally, please note that some languages (such as LISP or Smalltalk) provide a completely uniform notation system for control structures, operators and subprograms. A simple example of searching for the bigger number can be examined for a better comparison:

```
if a > b then max := a else max := b;            (* Pascal *)
(if (> a b) (setf max a) (setf max b))           ; LISP
(a > b) ifTrue: [max <- a] ifFalse: [max <- b].  "Smalltalk"
```

7.7 Implementation of subprograms

To better understand why these languages have the support for subprograms that they have, it is practical to know, at least schematically, the way of the implementation of subprograms. This section will serve this purpose.

During program execution subprograms call each other, so at a given moment multiple subprograms can be active, and in case of recursion even the same subprogram can be active multiple times (”in multiple *instances*”). The calling semantics of subprograms is reminiscent of a stack data structure: always that subprogram will be finished first which was called last. No wonder that calling of subprograms is implemented by a stack, the so-called *runtime stack*.

By activating a subprogram, information about it is pushed to the top of the runtime stack. If the subprogram is calling a newer subprogram, the called one will be stored above the caller within the stack. As long as the called subprogram is active, the caller also stays active. When the callee is finished, its data is removed from the runtime stack, and control is transferred back to the caller. After this, the caller can call another subprogram which will again be placed above the caller within the runtime stack. If the caller has finished all of its statements, its data is removed from the stack, and control is transferred back to its calling program unit.

Accordingly, at any point during program execution the active subprograms can be determined by looking up the runtime stack. The subprogram at the top of the stack shows where the control at the moment is, that is in which of them the actual statement is being executed.

The entries in the runtime stack show how subprograms are activating other subprograms. More precisely, in case recursion is also taken into account: how subprogram instances are activating the instances of other subprograms. Every instance (except the one at the top of the stack) has activated exactly that other one which is stored above it within the stack; and likewise, every instance (except the main program) is activated by exactly one other which is stored underneath it within the stack. The subprogram instances activating each other are also-called the *call chain*.

What kind of information is stored in the runtime stack about a subprogram instance exactly? The memory address must be stored definitely, where to return back after the execution of the subprogram has finished. Furthermore, the original values of the register at call time must be stored. This information is needed, because after returning from the called subprogram, the caller must continue from the same state, as it was in at the time of the call. The data which is stored within the runtime stack about an instance of a subprogram, is called the *activation record* of that instance.

In most of the programming languages the activation record usually contains the local variables and parameters of the subprogram instance as well. During the execution of the subprogram, there is an activation record in the runtime stack assigned to it, which stores its local variables and parameters. After leaving the subprogram, its activation record is removed from the runtime stack, so its local variables and parameters also cease to exist. Exception to this are local variables which are declared with static lifetime. These variables are not stored within the runtime stack, but in a dedicated memory space with the size and allocation determined in compilation time, the so-called *static memory*.

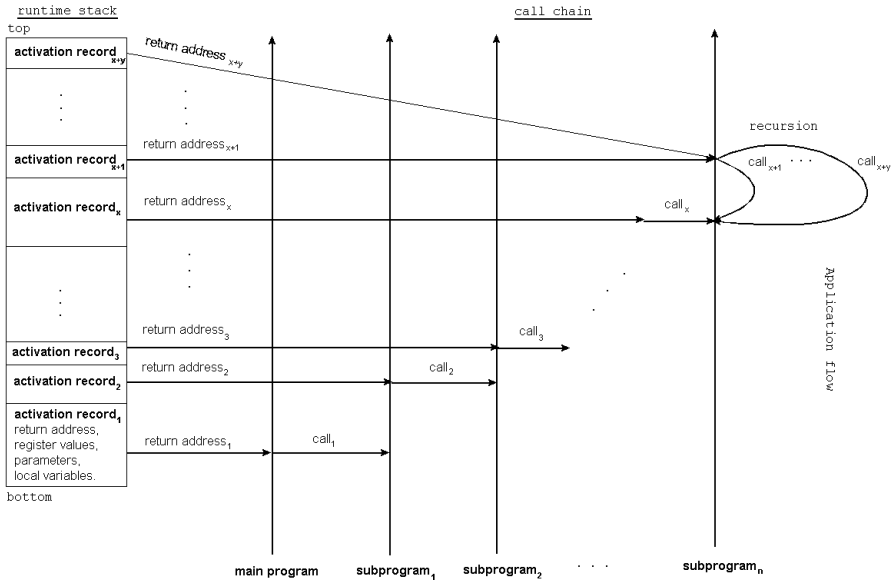


Figure 7.1: Activation records in the runtime stack

Do not forget that subprograms can also be recursive: at one given moment in time there can be multiple instances of them active. In this case every instance has its own activation record within the runtime stack, and these records hold their own set of local variables. The local variables are stored in the activation

record just to enable all called instances of a recursive subprogram to have their own set of local variables.

In FORTRAN (until FORTRAN 90) recursive subprograms were not supported, so (or that is why?) the local variables of subprograms have static lifetime. Static local variables make programs much more efficient, because there is no need to allocate memory at calling of the subprogram, and after leaving to release that memory area, and because the address of the local variables can be determined at program loading (otherwise local variables could only be accessed by indirect addressing) – although in return there is no possibility to write recursive subprograms. In FORTRAN 90 and PL/I a deal can be made: if a subprogram is declared to be able to call it recursively, then its local variables are stored within the runtime stack, the local variables of other subprograms will have a static lifetime. It is also true that using static local variables is only more efficient in relation to running time, but not in terms of allocated memory. Because all the static variables of all used subprograms exist during the whole execution time of the program, and this takes up memory.

The sizes of the activation records of subprograms are different, since it depends on the number and type of the local variables and formal parameters. What is more, in many languages the different instances of the same subprogram can have different sized activation records: just consider that, for example, in Ada the size of a formal parameter or local variable is only determined in runtime, depending on the actual parameters. So managing the calls is not that simple, as it looks for the first time. Within the activation record it must be stored as well where this record ends, that is where the activation record of the calling program unit starts. These pointers are also-called as the *dynamic link*.

The next problem is caused by non-local variables. Global variables have a static lifetime, so they are stored in the static memory. For this reason, their usage is very simple. In contrast to this, those non-global variables which are non-local for a subprogram cause much more trouble. In the non-block structured languages, such as C where every non-local variable is static, the subprograms have a much easier task than in the block structured languages. How can be a non-local, but also non-global variable found within the runtime stack? If dynamic scope rules are in effect, then the non-local variable must be in one of the caller subprograms. The dynamic link must be traversed, and the first activation record must be looked up where the given variable is declared. Nevertheless if the language applies a static scope, then not the callers, but one of the nesting subprograms will have the appropriate variable. Luckily, according to the rules of the static scope, an embedded subprogram can only be active, if the nesting subprogram is also active: since the embedded subprogram cannot be called from the outside of the nesting subprogram. So, within the runtime stack, there must be somewhere an activation record which holds the data for the nesting subprogram. Of course it is possible that the non-local variable cannot be found in the direct nesting, but in one of the indirect nesting subprograms. To support an easier lookup of the activation records of the nesting classes, the compiler

stores the location of the activation record of the nesting subprogram in every subprogram activation record. This is also-called the *static link*. For this reason, in case of static scope rules a non-local variable can be found by traversing the static link.[23]

The compiler can make a lot of optimizations during the compilation of subprograms. As mentioned before, the compiler can decide to substitute the code of the subprogram into the location of the call like with macros. The programmer can even hint this to the compiler in some languages. Another important and well known optimization method can be applied to recursive subprograms. Recursion is a very costly programming technique in terms of running time and memory usage, although it is very elegant and results in an easily readable code. Using a loop is much more efficient. There are recursive subprograms which can be easily converted into a loop by the compiler: these are the so-called tail-recursive subprograms. In these the recursive call is the last statement of the subprogram. This optimization technique will be shown on an example. Consider the following C function implementing a binary search [Set96]:

```
#define yes 1
#define no 0
int X[] = { 0, 11, 22, 33, 44, 55, 66, 77 };
int T;
int search (int lo, int hi)
{
    int k;
    if (lo > hi) return no;
    k = (lo + hi) / 2;
    if (T == X[k]) return yes;
    else if (T < X[k]) return search(lo, k - 1);
    else return search(k + 1, hi);
}
```

This *search* function can be converted to the following form:

```
int search (int lo, int hi) {
    int k;
L: if (lo > hi) return no;
    k = (lo + hi) / 2;
    if (T == X[k]) return yes;
    else if (T < X[k]) hi = k - 1;
    else lo = k + 1;
    goto L;
}
```

[23] There is another widespread method for keeping a record and looking up non-local variables which is based on the usage of so-called *display*. This method will not be discussed here.

The recursive call is located at the end of the subprogram, so when returning from it, the caller will also end immediately. This means that the local variables (including also the formal parameters) are no longer needed. It is unnecessary to create new instances from them for the recursive call, the already given variables should be used instead. Attention must be paid only to assure that the variables implementing the formal parameters are correctly set with the actual parameters of the recursive call.

7.7.1 Implementation of subprograms passed as parameters

The non-local variables can also make the implementation of subprograms, which are passed as parameters, rather complex. The subprogram passed as a parameter can refer to variables which were not declared within it, which means they are not part of the subprogram definition. It can be said that for the execution of the subprogram the knowledge of it is not enough, the environment of the subprogram is also needed – this can be different from the environment of that subprogram which received the subprogram as a parameter.

In non-block structured languages the environment of subprograms is very simple: all non-local variables are global, having a static lifetime. Every reference to a non-local variable can be dereferenced during program loading, that is why in these languages passing subprograms as parameters will not cause much of a problem: simply the starting address of the subprogram code must be passed. So in C and C++, for example, the implementation of the subprograms passed as parameters is done with pointers to the subprograms.

In the block structured Modula-2 the rule is that only most outer level subprograms can be passed as parameters, embedded ones cannot. This means that in this language it is possible to pass parameter subprograms with the help of a memory address: only if there are no references to non-global non-local variables from the passed subprogram.

Modula-3 (and some other languages) also supports passing embedded subprograms as parameters. In this case the subprogram passed as a parameter ”carries” an environment, in which it must be executed. This information is called a *closure*: the encapsulated environment seals the subprogram, so that every outward reference from the subprogram can be dereferenced within this environment. The implementation of passing the closure is quite complex. According to the scope rules there are three possible approaches.

Shallow binding

The environment within the closure is the same as the environment of the caller of the subprogram received as a parameter. This is the trivial solution: dynamic scope is assumed, the subprogram passed as a parameter is executed within the environment of its caller. (This approach is used, for example, by the SNOBOL language.)

Deep binding

The environment within the closure is that of the static scope, so the subprogram passed as a parameter will see during its execution what it can see at the location of its definition. This approach is used by most of the languages.

Ad-hoc binding

The environment within the closure will be of that statement which passes the subprogram as a parameter.

Subprograms as return values of functions cause even more complex problems. For this reason, Modula-3 allows a function to return a subprogram, but it reserves that the returned subprogram must be from the outermost level.

Parameter passing by name is also usually implemented by passing the closure. This is done also in ALGOL 60. Instead of the parameter by name, a subprogram is passed with the proper (deeply linked) reference environment. This (parameterless) subprogram will be called, when the formal parameter by name is referenced. The subprogram will be executed within the environment passed with it together, and the actual parameter will be evaluated. It can be seen that parameter passing by name is very inefficient. Every reference to the formal parameter causes the execution of a subprogram.

7.8 Iterators

In Sections 3.10.7. and 3.10.8. we have already introduced the notation of iterators. Let us compare now iterators with subprograms.

A loop is often written for an operation on all the elements of a value sequence. For example, to compute the factorial of the number n, the following loop can be used:

```
long factorial = 1;
for (int i = 1; i <= n; ++i)
  factorial *= i;
```

So, for all the values $1, 2, \ldots n$ the operation of a multiplication with the *factorial* variable should be carried out. Another example can be to write out all members of a data structure (such as a list, or binary tree) to the screen. The values for this operation are the elements of the data structure, the operation is the writing to the screen. If there is a way to access all the elements of the data structure one after the other, the task can be easily solved by a loop:

```
while ⟨there is something to process⟩
  ⟨take the next one⟩
  ⟨print it out⟩
```

Iterators implement this kind of control structure as a language construct. Instead of implementing all the details of the loop, only the operation should be focused on; handling of the loop variable, checking the end of the loop and advancing the loop can be left for the iterator. Every programmer has already written loops to iterate through the elements of an array at least thousand times. All such loops look essentially the same. Why should this loop be rewritten all the time? Let us hide the details in a reusable program unit, a control abstraction! If such a loop is needed, only that part must be programmed which is unique in it: the operation to execute on the sequence of elements. So, a proper iterator should be parametrized and used!

In the CLU language, for example, there is a predefined iterator, the *from_to* which will iterate through integers within a given `x. .y` interval. The implementation of this iterator could be imagined like the following:

```
from_to = iter (x, y : int) yields (int)
  i: int := x
  while i <= y do
    yield i
    i := i + 1
  end
end
```

The *yield* statement causes the execution of the iterator to be suspended, and returns the value of `i` to the calling program unit. At the next call of the iterator the execution will not start from its beginning, but from the last location where it was last suspended: from the statement following the suspending *yield* statement. The iterator ends if it reaches the end of its statements, in this case, after leaving the *while* loop. The iterator can be used the following way to compute the factorial:

```
factorial: int := 1
for i in from_to(1, n) do
  factorial = factorial * i
end
```

The *for in* structure calls the iterator, as long it can return some value. If the iterator ends, the execution of the *for in* will also end.

For our own data structures, an iterator operation can be written which will enumerate the elements stored within the data structure. Afterwards, the processing of the elements of the data structure, whatever this should mean, can be achieved comfortably with this iterator.

There are several languages which support the writing of iterators. Such languages are, for example, the Sather, Trellis or Alphard. The last one was probably the first language which introduced the iterators. Please note that in Alphard terms the iterators are called generators. This name is also appropriate, because iterators can be seen as a kind of element generators. An important

requirement is that the iterator should not count on the representation or the order of the elements during the generation. In Alphard iterators are not atomic language constructs, but they appear as special classes. These classes have similar characteristics and behavior, more precisely every generator class has the following (or similar) interfaces:

- Generating the first element (`$&$start`)
- Generating the next element (`$&$next`)
- Ending the generation (`$&$finish`)
- Getting the actual element (`$&$value`)
- Checking if the generation has been ended (`$&$done`)

Of course, from the above methods the first three are procedures, the fourth is a getter function, the last is a predicate. The language supports the following iterator structures:

- `upto` (iterating on a closed interval given by integers)
- `stepup` (same as `upto`, but also stepping can be specified)
- `first` (searching for the first element meeting the given requirements)
- `invec` (enumerating the elements of a vector)

The following Alphard example demonstrates the high abstraction power and combining of generators.

```
first i from upto(1,n) suchthat A[i]>max then
   max:=A[i]; maxpos:=i
fi
```

The source line above resembles a complete English sentence thanks to the high expression power. The code searches amongst the elements of an array within a given interval for the first element which is greater than the value stored in the `max` variable. The index of this element will be stored in the `maxpos` variable.

Another interesting fact is that the designers of the language – analyzing the behavior of the iterators and utilizing the fact that the iterators are themselves also classes – defined verification rules to the iterators, and verified them. The significance of this lies in the fact that assuming the predicate (such as the expression after the `suchthat` keyword) and the operation on the elements during the iteration will terminate, the constructions using iterators will also terminate and function properly.

Java introduces the notion of iterators with the *java.util.Enumeration* and *java.util.Iterator* interfaces. Java also uses data abstraction (objects) to implement iterators. The state of the iterator is not represented by an interrupted, suspended "subprogram", but rather as an inner state of an object which has

operations to advance the iterator. The following Java code would print out the elements of a *v* vector:

```
Enumeration e = v.elements();
while (e.hasMoreElements())
  System.out.println (e.nextElement());
```

C# offers a more elegant language structure for traversing with an iterator object. Let us assume that *numbers* is a data structure containing integers, and it has an enumerable type (for example an array).

```
foreach (int x in numbers) {
    Console.WriteLine(x);
}
```

In C++ the iterators [Jos99] from the Standard Template Library can be used to traverse the data structures. They mix the iterator behavior hidden by data abstraction with the possibilities of operator overloading, so their usage is formally the same as that of the pointers. The implementing classes of data structures usually contain an *iterator* member which is a type (the actual representation depends on the data structure), and the member functions *begin* and *end* returning iterators (positions) for accessing the beginning and end of the data structure. Consider, for example, the *list* data structure. The following code prints out the elements of an integer list named *data* to the standard output:

```
list<char>::iterator pos;
for (pos = data.begin(); pos != data.end(); ++pos) {
  cout << *pos << ' ';
}
```

The C++ iterators can be categorized into two groups based on the operations they allow: iterators which only allow the reading of the elements of the data structure (these are the constant iterators, their type is usually contained by the member called *const_iterator*), and those which allow the modification of the elements as well. Another kind of categorization of iterators is based on the order in which they can return the elements of the data structure (forward, for- and backward, or random access), or if they can be used to read (input iterator) or set (output iterator) the data structure.

7.9 Coroutines

Coroutines and subprograms are very similar program units. Their execution is performed according to the symmetric control model, in contrast to the asymmetric (caller–callee) control model of the subprograms. Coroutines are not executed continuously, such as subprograms, but discontinuously. The caller subprogram instance will not receive control back until the called subprogram is

not finished. In the case of coroutines the situation is different. A coroutine can pass control back to another coroutine even before it finishes. Next time when it is called, its operation continues from the point where it was last stopped. Coroutines can pass control between each other, as they like: they are not bound to the rigorous LIFO rules of the runtime stack, as the subprograms. Coroutine calls are often not even called as calls, but as *resuming*.

For what kind of tasks can coroutines be used? One of their most important field of usage is the mimicking of parallelism. A coroutine can be seen as equivalent to a "process", a logically connected sequence of statements. The coroutines are executed parallel to each other in one time, just like processes in a concurrent system. Of course, at one given moment in time only one coroutine is active, but if they pass control to each other quite frequently, they can mimic parallelism.

Iterators and coroutines are related concepts. If a language supports coroutines, it is not problematic to implement iterators in that language [Sco09]. Consider the following CLU code using the `from_to` iterator:

```
for i in from_to(first, last) do ... end
```

This should be converted to be something like the following ALGOL-like code:

```
begin
    i: int;
    isend: bool;
    it: coroutine := new from_to(first, last, i, isend, current_corutine);
    while not isend do
        ...
        transfer it
    end
    destroy it
end
```

Before starting the loop, a coroutine is created which will use the *i* and *isend* variables to return the next number and signal if it runs out of the specified interval. Using **transfer** will pass control again and again to the coroutine, until the *isend* variable becomes true. After the loop the coroutine is destroyed.

The implementation of the iterator can look like the following:

```
coroutine from_to(from,to:int; ref i:int; ref done:bool; caller:coroutine);
    i := from
    done := from <= to
    detach
    loop
        i := i+1
        done := i <= to
        transfer caller
    end
end
```

At the time of calling this coroutine the first data to be returned is set right away into the *i* and *done* parameters passed by address, and the new coroutine is detached from the old one. The next call of the coroutine will enter the loop which will – until the coroutine is destroyed – be again and again evaluated as the coroutine is called. After setting the *i* and *done* variables with the next computed data, the last statement of the loop body transfers the control back to the caller coroutine.

The first high level language supporting coroutines was SIMULA 67. This language was, as its name suggests, designed for making simulations. For simulations the modeling of separate processes is often needed that is why the coroutines came into the language. Other languages also adopted this language feature, such as BLISS, Interlisp or Modula-2. Some languages offer better language elements to implement parallelism, like execution threads in ALGOL 68, Modula-3, Ada, Occam or Java. This will be dealt with in more detail in Chapter 13.

7.10 Summary

In this chapter the control abstraction and the subprograms as language elements for the implementation were discussed. It was shown that using subprograms decreases the complexity of the code, and increases the reusability, readability, changeability and maintainability of the finished software product.

Two kinds of subprograms are distinguished based on the mode of their call. The subprograms used as statements are called procedures, those used in expressions are called functions. It was discussed that side effects for functions should be avoided.

Subprograms receive the required information for their operation through input parameters, and return results through output parameters, or in case of functions as return values to the caller. Subprograms can also use and modify the values of non-local variables. The specification of the subprogram contains all the information needed to call the subprogram. The compiler may check if the calls match this specification.

The languages model the input, output and in- and output subprogram parameters with different parameter passing modes (by value, by address, by result, by value/result and by name, etc.). Subprograms may also be parameters for other subprograms. The implementation of subprograms passed as a parameter in a block structured language following static scope rules, is based on the concept of the closure.

The flexible usage of subprograms may be supported in the different languages by variable length parameter lists, default values of the formal parameters, overloading, indefinite typed parameters and operators defined by the programmer. Recursive subprograms, subprogram types and the Curry-method can further increase the usability of languages.

7.11 Exercises

Exercise 7.1. Answer the following questions briefly!

1. What is the difference between procedures and functions?
2. What does it mean if a function has side-effects?
3. What is the purpose of the subprogram specification, and what does it consist of?
4. When is it necessary to declare subprograms in advance?
5. In which case should inline subprograms be used? How do they differ from macros?
6. What does the overloading of subprogram names mean?
7. Give examples for languages which support operator overloading and definition of new operators!

Exercise 7.2. Refresh your knowledge concerning parameter passing - give short answers for the following questions!

1. What is the formal and what is the actual parameter?
2. What can be a parameter for a subprogram? Give examples from different languages!
3. Why is it good if the compiler checks the types of the actual parameters?
4. Why is it good if the formal parameter can have a default value?
5. How can parameters be categorized based on the direction of information flow?
6. Give examples for languages where it is possible to forbid for a subprogram to modify the value of the actual parameter? Why is this good?
7. What does parameter passing by copy, by sharing and by need mean? Give examples for languages which can support these!

Exercise 7.3. Write a function which determines if some natural numbers stored in an array are relatively primes!

Exercise 7.4. Implement the binary search algorithm!

Exercise 7.5. Make a sorting procedure using the quicksort algorithm!

Exercise 7.6. Write a subprogram for transposing a matrix! Write it as a function and also as a procedure. Which one when should be used?

Exercise 7.7. Write a subprogram to search within a given interval specified by parameters the first zero point of a function also passed as a parameter!

Exercise 7.8. Make a C++ and Ada function to multiply two matrices! Compare the language elements used! (Operator-overloading, number of parameters, definiteness of the types of parameters.)

Exercise 7.9. Write a subprogram to compute the n^{th} Fibonacci number! Solve this task with a loop and with recursion. What differences in the efficiency can be observed?

Exercise 7.10. Make a summarizing procedure in the C++ and Ada languages. To which extent can the solution be made generic in these languages?

Exercise 7.11. Is the output of the following C++ program computing the factorial of a given number correct?

```
#include <iostream>
using namespace std;
int factorial(int & i){
    int fact;
    fact=1;
    while (i > 1){
      fact = fact*i;
      i-;
    }
    return fact;
}
int main() {
    int i;
cout« "Give the serial number of the factorial: ";
cin » i;
cout « i « "! = " « factorial(i) « "\n";
cout « i « "! = " « factorial(i) « "\n";
    return 0;
}
```

Exercise 7.12. The following C++ program compiles without any errors or warnings. Thus, the displayed result is non-deterministic and not expected. Why?

```
#include <iostream>
using namespace std;
int multiply(int a){
    int n;
    return a*n;
}
int main() {
    int v=2;
    cout « "the result is: " « multiply (v);
    return 0;
}
```

Exercise 7.13. Compare the output lines with the values of `n` in the following java code, and explain the differences!

```
public class Hello {
   public static void main(String[] args) {
     Integer n = new Integer(2);
     System.out.println("n=" + n);
     multiply(n,3);
     System.out.println("n=" + n);
    }

    public static void multiply(Integer n, Integer m) {
      n=n*m;
      System.out.println("in multiply: n=" + n);
    }
}
```

7.12 Useful tips

Tip 7.1. We will reference the sections discussing the given questions.

1. As described in Section 7.2, both procedures and functions are common kinds of subprograms, for the main difference between them consider which computes mainly return values and which implement transformations on the state space.
2. Think of the previous question, and about the expected role of the function. Consider the case if some additional effects appear besides this expected role. We have also discussed in Section 7.2.1, what if this side-effect is actually the desired behavior, and otherwise if not expected, why this situation should be avoided (see Section 7.2.2).
3. In Section 7.3.2 we have covered the need of an interface of the subprogram to the outside word to present in some form how it can be called. Consider the possibilities and restrictions of different programming languages, how and how much of this information is required and can be specified, and who (or what) could possibly need this kind of information?
4. In Section 7.3.6 we have mentioned some programming languages, which require that all entities must be declared before usage. How and in which order could subprograms calling each other be declared in such programming languages? Also think of the possibility of implementation hiding.
5. In short, think of the question of efficiency. As we have explained in Section 7.3.7, inline subprograms and macros both are compiled as substitutions, but consider the different levels, where these substitutions are actually applied.

6. Please refer to Section 7.6, where we have explained how some languages can support the same name for multiple different subprograms.
7. In Section 7.6.1 we have identified the operators as special subprograms which can be called with special syntax (see 7.6). Appropriate language support and examples have been also introduced there.

Tip 7.2. Parameter passing have been discussed in Section 7.4. Refresh your knowledge from there.

1. Based on Section 7.3, consider how the parameters are referenced within the subprogram body and how the values are passed to these parameters during subprogram call.
2. Section 7.3.1 has listed the entities that are allowed as parameters for a subprogram. Language support for indefinite types, unconstrained arrays, multidimensional arrays, subprograms and labels as parameters have been discussed with examples, type and module parameters have been mention here and are covered in Chapter 11 in more detail.
3. Consider how the compiler could help program reliability by checking type correctness.
4. In Section 7.3.4 we have seen how default values can help shortening the form and increase the flexibility of subprogram calling.
5. In the beginning of Section 7.4 the three basic types of parameters have been mentioned based on the direction of information flow between the caller and the called subprogram.
6. Think of the **const** or the **READONLY** modifiers of the formal parameters discussed in Section 7.4.3. Consider also how this write protection for the actual parameter can influence program reliability and parameter passing performance.
7. Some sources consider the parameter passing by value/result variant, used in Ada, to be different logically from the one that of ALGOL W, and call it as *parameter passing by copy*. In our book we consider, like most of the sources, parameter passing by copy and by value/result to be logically the same, the difference is only considered to be an implementation anomaly.
   In Java parameter passing by value is applied. As objects can be accessed through implicit references, and passing an object as a parameter means passing the reference by value, so a subprogram can change the object received as the actual parameter, or to be more precise, the object which is referenced by the reference received as the actual parameter. (This is usually called *parameter passing by sharing*.
   In the lazy evaluated functional languages parameter passing takes place, as if parameter passing occurred by value. The difference is that the evaluation of the actual parameter and the determination of the value of the formal parameter is not carried out at the calling of the function, but later, when the value of the parameter is needed for the first time. This kind of parameter passing is called *by need*.

Tip 7.3. Two integers m and n are said to be *relatively prime* if the only positive integer that evenly divides both of them is 1. This means also that their greatest common divisor (denote it by $gcd(m, n)$ is 1. We have to check in the array v that

$$\forall i, j \, in \, v'range \, if \, i \neq j \, gcd(v[i], v[j]) = 1$$

holds.

Tip 7.4. The binary search algorithm finds the position of a specified input value within a sorted array. In each step, the algorithm compares the searched value with the middle element of the array. If they do not match, and the value is less than the middle element, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the value is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the value cannot be found in the array. This binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time.

Tip 7.5. The quicksort algorithm is well known, for details see e.g. the book [Cor09]. Write a generic procedure which solves the problem for unconstrained amount of data stored in a vector, for which an ordering relation exists.

Tip 7.6. The transpose of an $m \times n$ matrix $\mathbf{A}$ is another matrix $\mathbf{A}^{\mathrm{T}}$ of the size $n \times m$ where $[\mathbf{A}^{\mathrm{T}}]_{ij} = [\mathbf{A}]_{ji}$. From this definition it can be seen, that the dimensions of the original matrix are swapped for the resulting matrix, this condition must be checked or ensured. The transposing itself can be implemented as a nested iteration over all the elements of the source or the target matrix, and setting the appropriate element according to the above definition.

Tip 7.7. For an exact solution the domain of the function should be finite, i.e. discrete. So for all the possible values the function may be evaluated and checked if it gives zero or not. This is a simple iteration over all the domain values in the given interval. As we are searching for the first zero point, the type of the domain for the function must support ordering, and the iteration should loop in an ascending order. The type of the value set of the function must have a zero value and support the equality comparison with it.

Tip 7.8. If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix, the result $\mathbf{AB}$ of their multiplication is an $m \times p$ matrix where $(\mathbf{AB})_{ij} = \sum_{k=1}^{m} \mathbf{A}_{ik}\mathbf{B}_{kj}$. From this definition it can be seen, that the number of columns of the first matrix must be equal to the number of rows of the second matrix, so this condition must be checked. The multiplication can be implemented as a nested iteration over all the elements of the product matrix, and computing the appropriate element with another summarizing loop according to the above definition.

Tip 7.9. The sequence of Fibonacci numbers is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0$ and $F_1 = 1$ by definition. As this definition is

already recursive, that kind of solution is pretty straightforward. The iterative approach is practically the enumeration of all the Fibonacci numbers until n, where only the two last result numbers are required to compute the next one.

Tip 7.10. Summarizing requires a data type which can be indexed, the index range should have an ordering (to be able to iterate over it), the value typeset should have a zero value and an addition operation. Generally summarizing works as iterating over the index range and adding up the indexed values, starting from a zero result value. The extent of genericity depends on the generic support of the given language (see Chapter 11) and the generic possibilities of the chosen data types fulfilling the above mentioned requirements for the proper working of summarizing.

Tip 7.11. Run the example program! Please note, that the computation of the factorial is invoked twice, so the result should be also displayed twice. If you see different, incorrect results, check the code of the computation algorithm. Could that function cause some kind of side-effects?

Tip 7.12. Take a closer look, what the expected result of the function call should be. Can you identify and explain it?

Tip 7.13. As we have explained in Section 7.4.3, in Java parameter passing is done by sharing. Within the body of the *multiply* static method, the received object is seemingly modified through its reference, but according to the output of the program, the caller object stays unmodified. So, what has been changed exactly by the assignment within this method?

7.13 Solutions

Solution 7.1.

1. In programming theory there are two common kinds of subprograms: procedures and functions. Procedures implement transformations on the state space defined by their variables or on the environment of the program. Functions on the other hand compute a value. Calling procedures can be considered as a statement, calling a function is more like an expression. It can be seen that procedures extend the statement set of the language, and functions extend the operator set used for expressions.
2. Usually functions compute a value, but do not make transformations, and have no effect either on the values of the program variables, or on the program environment; this is often said the functions have no side-effects. Many programming languages, such as C, C++, Java, C# or CLU do not really differentiate between procedures and functions. In these languages there is no sharp dividing line between statements and expressions either. An expression can stand alone as a statement, and its execution means the evaluation of the expression which often causes some side-effects.

For example, the value of the *c++* expression is the same as the value of the *c* variable, but as a side-effect, the value of the variable is incremented by one. In these languages all subprograms are "functions", but some of them have a return value of an empty "void type". These kinds of functions are practically procedures. Avoiding side-effects also improves program readability, clarity and portability. If functions with side-effects are called in an expression, their value (and the order of the statements causing side-effect) also depends on the evaluation order (precedence) of operators within the expression. This makes the understanding of the code more problematic even if the evaluation order within expressions is precisely defined. If this is not defined clearly, expressions can occur easily with uncertain results undefined by the language, and this error is not even captured by the compiler. In such a case, the program can "accidentally function properly" on a given architecture, but on a different system, or just after switching to another compiler everything goes suddenly wrong.

3. For the caller program units the specification of the subprogram is what they can access. It is sometimes also said that the interface of the subprogram to the outside word is its specification. Callers cannot see anything from the body. So the control abstraction is realized in this way: the callers face only an abstraction of the concrete statement sequence. The specification usually contains the name of the subprogram, whether it is a function or not, and if yes, the type of the return value, and also which parameters are accepted. The more precisely the specification describes how to use the subprogram, the easier it is to write correct programs in the given language. The compiler uses the specification to check whether the caller of a subprogram is using it correctly or not, for example passing the proper count of parameters.
4. In some languages there is a limitation for every entity appearing in the program so their type must be specified before usage. Technically, it is customary to say that all entities must be declared before usage. In such languages this is also true for subprograms. There are situations when subprograms cannot be declared by giving their definition. For example, in case of an indirect recursion, that is, if subprogram `A` calls subprogram `B` which calls subprogram `A`, there is no way to give the definition of the two subprograms so that they both would be defined before usage. In this case a forward declaration can help. It essentially consists of the specification of the subprogram, giving only information about how to use it.

   Another case is if the subprogram should not be declared by its definition because of implementation hiding. In Ada, Modula-2 and other languages that support modules, in the specification part of the module its usage is specified (such as its exported subprograms, types, variables etc. can be declared here), the body of the module contains the implementation of its exported entities. So the specification part of the module only contains

the specification of the subprograms, which is enough to decide if these are called correctly by the user program units of the module. The full definition of the subprogram is contained by the body of the module.

5. Macros can be considered as the ancestors of subprograms. They were introduced in the assembly languages, but can be used in LISP, BLISS, or in the C preprocessor, probably this is the best known. The C macro can be used to arrange the program source, like the subprograms. It is an efficient, but not too safe language construct. It can accept parameters, but the type of actual parameters will not be checked against the types of the formal parameters. Macros are more efficient than subprograms, from the aspect that the extras of subprogram calling and returning are omitted. The body of the macro is substituted at every call into the program source, so no extra administration is required for calling (such as saving registers and return address, etc.). For shorter running time, it is more efficient to use macros but the final size of the program with macros will be greater than with subprograms.
   Some languages (Ada, C++, Euclid, LISP etc.) try to combine the advantages of macros and subprograms by supporting the usage of inline subprograms. In these languages the programmer can usually give hints to the compiler, which subprograms should be compiled as substitutions. Inline programs beside having the efficiency of macros, can offer safety by their subprogram specification, and can also ensure the production of semantically clearer code. The inline subprograms represent only an implementation technique, they do not have an effect on the function of the program.
6. Some languages allow that at some point of the program the same subprogram name can denote multiple subprograms. If a subprogram is called by that name the compiler will find out from the context of the call, which subprogram should be executed. If the compiler is unable to determine exactly which subprogram by the same (overloaded) name should be called, it will generate a compilation error. Also, if there is no overloaded version which would match the call, a compilation error will be thrown again.
7. Operators are such subprograms which can be called with special syntax. For example, for addition in most of the languages the + operator can be used, which is usually called in contrast to subprograms not in prefix, but in infix form. Many languages offer for their predefined types predefined (often overloaded, such as the + for integers and for reals) operators, but do not allow the programmer to define operators with similar semantics for user types. Such a language is, for example, Java which otherwise allow normal subprogram overloading. In contrast, in C++ or Ada, if a matrix type is written, an infix + operation can also be defined. In Ada it is an important limitation that operators defined by the language can

be overloaded, but no new operators can be defined. In ML, Clean etc. also this is supported. FORTRAN 90 also allows the definition of new infix operators, but requires to enclose the operator between dots (such as `A .cross. B`), and every such operator is assigned the same precedence level.

Solution 7.2. 1. Formal parameters are those specified in the definition of the subprogram. They are used throughout its body to describe the operations. Actual parameters are passed by the caller to the subprogram which has to perform its operation with these received parameters. This means that actual parameters must be matched against formal parameters. The formal or actual parameter sequence of a subprogram is commonly called as the parameter list.

2. In a programming language entities are usually vales, types, subprograms and modules. In Ada additional entities are processes or in Clean the type constructors. To use entities as parameters is allowed almost in every programming language. (Except, for example, some early variants of BASIC among others, which did not allow any parameters for subprograms at all.)

   Value entities as parameters are the most basic possibility supported by all modern programming languages.

   Many languages allow formal parameters to have indefinite types. With these, such programs can be made which can handle unconstrained arrays. In Ada, not only the unconstrained, that is variable length array can be indefinite, but also the unconstrained discriminated record type. Even in Standard Pascal it is possible to write a sorting procedure which is able to use arbitrary length arrays. In C, arbitrary length arrays can be passed with the help of pointers as parameters. In Java, multidimensional arrays are such one-dimensional arrays which contain arrays.

   To generalize a possible implementation of an integral function, there must be a way to pass a subprogram as a parameter. This was not allowed at all in some early imperative languages (ASA FORTRAN, COBOL etc.). The possibility appeared in the functional languages (including LISP already born in the end of the 50's), in ALGOL 60 and in numerous languages designed later (PL/I, SIMULA 67, ALGOL 68, Pascal, FORTRAN 77, Modula-2, Modula-3 etc.).

   In ALGOL 60, PL/I, SIMULA 67, ALGOL 68 etc. labels can also be passed to subprograms as a parameter.

   In several current languages subprograms cannot be, but pointers to them can be passed as parameters. Such languages are, for example, C, C++ or Ada.

   The reusability of a subprogram can greatly be improved if types could be passed as parameters. This is supported by so called generics, in languages such as Java, C++ and Ada for example.

3. The compiler ensures program reliability by checking type correctness at parameter passing. The compiler uses the specification to check whether the caller of a subprogram is using it correctly or not, for example passing the proper count and types of parameters. This is done in compile time, so unexplainable runtime errors caused by type mismatch can be avoided in advance.
4. There are languages which support default values for formal input parameters. In this case it is not needed to match all the formal parameters with actual values: formal parameters without actual values will have their default values – if specified. Such subprograms always have the same number of parameters, but not all of them must be explicitly set. This feature is very useful for subprograms with many formal parameters which usually have the same values. So there are parameters which are important to be set, and there are ones, that are less important. According to this, formal parameters with default values must be placed at the end of the parameter list, in a descending order of their importance.
5. Subprogram parameters can be grouped in three groups based on the direction of information flow. The input parameters can be found in the first group. They deliver information from the caller into the called subprogram. The second group is for the output parameters which return information from the subprogram to the caller. The last group of in- and output (inout) parameters ensure a two way information flow.
6. C++ and the ANSI C contain a very interesting language element: in the definition of the formal parameters the **const** keyword can appear. This means that the instance referenced by the formal parameter cannot be changed by the subprogram. If the values of type of the formal parameter are large, this trick helps avoiding data transferring which would occur at parameter passing by value: instead, only the costs of passing by address, which is usually more efficient, would occur at subprogram call. The advantages of this technique are even better visible, if combined with the reference type of C++. In this case an input parameter is implemented purely with parameter passing by address. In other languages there are also similar constructs to const. In Modula-3, for example, the formal parameter can be declared as `READONLY`. The compiler will ensure that the formal parameter will not appear on the left side of an assignment, and will not be passed in an in- and output manner as the actual parameter to another subprogram. The input parameter, declared as `READONLY`, will usually be passed by value, if it is small in size, and by address, if it is large. If a right-side expression is passed as the actual to a `READONLY` parameter by address, the compiler will - just like in FORTRAN - use a temporary variable to store the value of the right-side expression, and its address will be passed to the formal parameter.
7. Some sources consider the parameter passing by value/result variant, used in Ada, to be different logically from the one that of ALGOL W, and call

it as parameter passing by copy. In our book we consider, like most of the sources parameter passing by copy and by value/result to be logically the same, the difference is only considered to be an implementation anomaly. This parameter passing mode is usually used to implement in- and output parameters. Basically it is the joint application of by value and by result parameter passing modes. The formal parameter is a local variable of the subprogram, which at the calling of the subprogram becomes the value of the actual parameter, then at the end its value is copied back into the actual parameter. The actual parameter can only be an L-value, so that it can receive the result computed by the subprogram.

In Java parameter passing by value is applied. As objects can be accessed through implicit references, and passing an object as a parameter means passing the reference by value, so a subprogram can change the object received as the actual parameter, or to be more precise, the object which is referenced by the reference received as the actual parameter. This is usually called parameter passing by sharing. This name is originated from CLU.

In the lazy evaluated functional languages parameter passing takes place, as if parameter passing occurred by value. The difference is that the evaluation of the actual parameter and the determination of the value of the formal parameter is not carried out at the calling of the function, but later, when the value of the parameter is needed for the first time. This kind of parameter passing is called by need. Parameter passing by need is a bit like by name, because of the principle of late binding. But opposed to parameter passing by name, the value of the actual parameter will only be evaluated once. So, if there are no side effects during the evaluation, there would be also no difference between these two parameter passing modes.

Solution 7.3. The solution is in Ada, it is embedded into a very simple test procedure:

```
with Text_IO; use Text_IO;
procedure Rel_Prime_Test is
   package Pos_IO is new Integer_IO(Positive);
   use Pos_IO;
   package Bool_IO is new Enumeration_IO(Boolean);
   use Bool_IO;
   type Myvect is array(Positive range <>) of Positive;
   Vector1 : Myvect(1..n) := (7,8,9,11,5);
   Vector2 : Myvect(1..n) := (7,8,9,11,10);
   Rel_Prime : Boolean;
   I,J : Natural;
   function Greatest_Common_Divisor( A, B: Positive ) return Positive is
     X :  Positive := A;
     Y :  Positive := B;
     begin
     while X /= Y loop
          if X > Y then X := X-Y; else Y := Y-X; end if;
     end loop;
     return X;
```

```
   end Greatest_Common_Divisor;
   function Relative_Prime( A, B : Positive) return Boolean is
   begin
     return  Greatest_Common_Divisor (A,B) = 1;
   end Relative_Prime;
   procedure Rel_Prime_Vect (V : in Myvect;  Rel_Prime : out Boolean;
                             I, J : out Natural) is
     M : Positive := V'First;
     N : Positive := V'Last;
     begin
     I := M-1;
     Rel_Prime := True;
    while I < N and Rel_Prime loop
      I := I+1;
      J := I+1;
       while J <= N and Rel_Prime loop
         Rel_Prime :=  Relative_Prime(V(I),V(J));
         J:=J+1;
      end loop;
    end loop;
    if not Rel_Prime then
      J := J-1;
    end if;
   end Rel_Prime_Vect;
begin
   Rel_Prime_Vect(Vector1, Rel_Prime, I, J);
   if not Rel_Prime then
        Put_line("The first elements in the vector");
        Put_line("which are not relatively prime:");
        Put(Vector1(I)); Put(" "); Put(Vector1(J));
     else
        Put("The elements in the vector are ");
        Put_Line("relatively prime");
     end if;
   Rel_Prime_Vect(Vector2, Rel_Prime, I, J);
   if not Rel_Prime then
       Put_line("The first elements in the vector");
       Put_line("which are not relatively prime:");
       Put(Vector2(I)); Put(" "); Put(Vector2(J);
     else
       Put("The elements in the vector are ");
       Put_Line("relatively prime");
     end if;
end Rel_Prime_Test;
```

Solution 7.4. The solution is in Ada as a generic procedure. The specification of the generic binary search algorithm:

```
generic
   type Item is private;
   type Index is (<>);
   type Vector is array (Index range <>) of Item;
   with function "<"(X, Y : Item) return Boolean is <> ;
 procedure Log_Search(V:in Vector; X:in Item;  T:out Boolean; Ind:out Index);
```

The body of the generic algorithm:

```
procedure Log_Search(V:in Vector; X:in Item;  T:out Boolean; Ind:out Index) is
    M, N, K : Integer;
    L : Boolean;
    begin
        M := Index'Pos(V'First); N := Index'Pos(V'Last); L := False;
         while not L and then M <= N loop
            K := (M + N) / 2;
            if X < V(Index'Val(K)) then
```

```
            N := K - 1;
         elsif X = V(Index'Val(K)) then
            Ind := Index'Val(K);
            L := True;
         else
         -- the function ">" is not supposed between
         -- the generic parameters
            M := K + 1;
         end if;
      end loop;
      T := L;
   end Log_Search;
```

A simple test of the procedure:

```
with Text_IO, Log_Search, Quick_Sort; use Text_IO;
procedure Log_Search_Test is
   type Months is
         (January, February, March,  April, May,  June,
          July, August, September, October, November, December);
   package Int_IO is new Integer_IO(Integer); use Int_IO;
   package Month_IO is new Enumeration_IO(Months); use Month_IO;
   package Bool_IO  is new Enumeration_IO(Boolean); use Bool_IO;

   type My_Vect  is   array (Integer range <>) of Months;

   procedure My_Log_Search  is
      new Log_Search(Months, Integer,  My_Vect);
   procedure My_Sort  is
      new Quick_Sort(Months, Integer,  My_Vect);
      -- we can use the Quick_Sort algorithm from Exercise above
   V : My_Vect (1 .. 5) := (October, March, February, August, September);
   Elem  : Months;
   Found, OK : Boolean;
   Ind   : Integer;

begin
   My_Sort(V);
   Put_Line("Give the month");
   OK:=False;
   while not OK loop
     begin
       Get(Elem); OK:=True;
       exception
          when Data_Error =>
                Put_Line("Please give it once more, it has to be a month: ");
     end;
    end loop;
    My_Log_Search(V,Elem, Found, Ind);
    if Found then
      Put("The month ");
      Put(Elem);
      Put(" is in the  vector at the index: ");
      Put(Ind);
      New_Line;
    else
      Put("The month ");
      Put(Elem);
      Put(" is not in the  vector");
      New_Line;
      Put_Line("The element in the vector are:");
      for I in V'First..V'Last loop
         Put(V(I)); Put(" ");
      end loop;
      New_Line;
```

```
    end if;
end Log_Search_Test;
```

Solution 7.5. The solution is in Ada as a generic procedure. The specification:

```
generic
  type Item is private;
  type Index is (< >);
  type Vector is array(Index range  < >) of Item;
  with function "<"(X, Y : Item) return Boolean is  < >;
procedure Quick_Sort(V : in out Vector);
```

The body:

```
procedure Quick_Sort(V : in out Vector) is
  procedure Sort (Left,Right :  in Index) is
    Pivot : Item :=
     V(Index'Val((Index'Pos(Left)+Index'Pos(Right))/2));
    Leftind : Index := Left;
    Rightind : Index := Right;
     begin
       loop
         while V(Leftind) < Pivot loop
           Leftind := Index'Succ(Leftind);
         end loop;
         while Pivot < V(Rightind) loop
           Rightind := Index'Pred(Rightind);
         end loop;
         if Leftind < =Rightind then
           declare
             Temp:Item := V(Leftind);
           begin
             V(Leftind) := V(Rightind);
             V(Rightind) := Temp;
           end;
           Leftind := Index'Succ(Leftind);
           Rightind := Index'Pred(Rightind);
         end if;
         exit when Leftind>Rightind;
       end loop;
       if Left < Rightind then
         Sort(Left,Rightind);
       end if;
       if Leftind < Right then
         Sort(Leftind,Right);
       end if;
     end Sort;
 begin
    Sort(V'First,V'Last);
end Quick_Sort;
```

A small example demonstrates the instantiation of the generic procedure:

```
with Quick_Sort, Text_IO; use Text_IO;
procedure  Quick_Demo is
   subtype Element is Integer range 0..1000;
   subtype index is Integer;
   type Vect is array (Index range <>) of Element;
   A : Vect(1..5):=(7,3,4,2,0);
   procedure Qsort is new Quick_Sort(Element,Index,Vect);--the actual procedure
   package Int_IO is new Integer_IO(Integer);
   use Int_IO;
 begin
```

```
   Put("The original vector:"); New_Line;
   for I in A'First..A'Last loop
     put(A(i));
   end loop;
   New_Line;
   Qsort(A);
   Put("The sorted vector:"); New_Line;
   for I in A'First..A'Last loop
     put(A(i));
   end loop;
   New_Line;
end Quick_Demo;
```

Solution 7.6. Both solutions are Ada generics. The specification of the generic procedure:

```
generic
   type Elem is private;
   type Index is (<>);
   type Matrix is array (Index range <>, Index range <>) of Elem;
   with function "+"(A, B : Elem) return Elem is <>;
   with function "*"(A, B : Elem) return Elem is <>;
procedure Matrix_Transp_Proc(A :in Matrix; B:out Matrix);
```

The body:

```
procedure Matrix_Transp_Proc(A:in Matrix; B: out Matrix) is
begin
   if not( A'First(1)=B'First(2) and A'Last(1)=B'Last(2)) then
      raise  Constraint_Error ;
   end if;
   for I in A'Range(1) loop
      for J in A'Range(2) loop
        B(J,I) := A(I,J);
       end loop;
    end loop;
end Matrix_Transp_Proc;
```

The specification of the generic function is very similar:

```
generic
   type Elem is private;
   type Index is (<>);
   type Matrix is array (Index range <>, Index range <>) of Elem;
   with function "+"(A, B : Elem) return Elem is <>;
   with function "*"(A, B : Elem) return Elem is <>;
function Matrix_Transp(A :in Matrix) return Matrix;
```

The body:

```
function Matrix_Transp(A:in Matrix) return Matrix is
      C : Matrix(A'Range(2), A'Range(1));
begin
    for I in A'Range(1) loop
      for J in A'Range(2) loop
        C(J,I) := A(I,J);
       end loop;
    end loop;
   return C;
end Matrix_Transp;
```

The next program demonstrates the use of both solutions:

```
with Matrix_Transp, Matrix_Transp_Proc, TEXT_IO;use TEXT_IO;
procedure Matr_transp_trial is
   subtype Element is Integer range 0..1000;
   subtype Index is Integer;
   type Matr is array (Index range <>, Index range <>) of Element;

-- instantiation of the generic function and the procedure:
   function Transp is new Matrix_Transp(Element,Index,Matr);
   procedure Transp is new Matrix_Transp_Proc(Element,Index,Matr);

A : Matr(1..2,1..3) := ((1,2,7),(3,4,9));
C : Matr(1..3,1..2);

package Int_IO is new Integer_IO(Integer); use Int_IO;
procedure Put(A : Matr) is
begin
   for I in A'First(1)..A'Last(1) loop
     for J in A'First(2)..A'Last(2) loop
        put(A(i,j));
     end loop;
     New_Line;
   end loop;
end;

begin
   Put("The original matrix:"); New_Line;
   Put(A);
   C:=Transp(A); -- the call of the function

   Put("The transposed matrix:"); New_Line;
   PUT(C);

   Transp(A,C); -- the call of the procedure

   Put("The transposed matrix:"); New_Line;
   PUT(C);

end;
```

Solution 7.7. Exact solution is possible only if the domain and range of the function are discrete. Thus a general solution can be written e.g. in Ada, as follows.

The specification:

```
generic
   type Domain is (<>);
   type Values is (<>);
   Zero: Values;
   package Zero_Point_Generic is
      type Func_Pointer is access function (D : Domain) return Values;
      procedure Zero_Point (Low, High : in Domain; F : in Func_Pointer;
                            Found : out Boolean; Zero_P : out Domain);
   end Zero_Point_Generic;
```

The body:

```
package body Zero_Point_Generic is

   procedure Zero_Point (Low, High : in Domain;
                         F : in Func_Pointer;
                         Found : out Boolean;
                         Zero_P : out Domain) is
      Element : Domain;
```

```
   begin
      if Low>High then raise Constraint_Error; --these must specify an interval
      end if;
      Element := Low;
      Found := F(Element)=Zero;
      while Element < High and not Found loop
          Element := Domain'Succ(Element);
          Found := F(Element)=Zero;
      end loop;
      if Found then
         Zero_P := Element;
      end if;
   end Zero_Point;

 end Zero_Point_Generic;
```

A simple test, to demonstrate instantiation and call:

```
with Text_IO, Zero_Point_Generic; use  Text_IO;
procedure Zero_Point_Test is
   package Zero_Test is new Zero_Point_Generic (Integer, Integer, 0);
   use Zero_Test;
   package Int_IO is new Integer_IO(Integer);
   use Int_IO;
   function Test (X: Integer) return Integer is
   begin
      return X+1;
   end Test;

   Test_P: Func_Pointer :=  Test'access; -- pointer to function
   Found: Boolean;
   Zero_Place : Integer;
   Low: Integer:= -2;
   High : Integer := 5;
begin
   Zero_Point (Low, High, Test_P, Found, Zero_Place);
   Put(Found);
   New_Line;
   if Found then
      Put("The zero-place of the test function ");
      Put_Line("in the interval:");
      Put(Low); Put(" "); Put(High);
      New_Line;
      Put(Zero_Place);
   else
      Put("The test function has not zero-place ");
      Put_Line("in the interval:");
      Put(Low); Put(" "); Put(High);
      New_Line;
   end if;
end Zero_Point_Test;
```

Solution 7.8. In the Ada programming language generic solutions can be given using unconstrained array type as parameter.

The specification:

```
generic
    type Element is private;
    type Index is (<>);
    type Matrix is array (Index range <>, Index range <>) of  Element;
    with function "+"(A, B :  Element) return  Element is <>;
    with function "*"(A, B :  Element) return  Element is <>;
function Matrix_Product(A, B:in Matrix) return Matrix;
```

The body:

```
function Matrix_Product(A, B:in Matrix) return Matrix is
    L : Index := A'First(2);
    C : Matrix(A'Range(1), B'Range(2));
  begin
   if A'Length(2) /= B'Length(1) then raise Constraint_Error;
      -- not appropriate sizes of matrices
   end if;
    for I in A' Range(1) loop
      for J in B' Range(2) loop
        C(I,J) := A(I,L) * B(L,J);
        for K in Index' Succ(L)..A' Last(2) loop
          C(I,J) := C(I,J) + A(I,K) * B(K,J);
        end loop;
      end loop;
    end loop;
   return C;
  end Matrix_Product;
```

The following procedure demonstrates the instantiation of the generic function:

```
with Matrix_Product, TEXT_IO;use TEXT_IO;
procedure Matr_trial is
   subtype Element is Integer range 0..1000;
   subtype Index is Integer;
   type Matr is array (Index range <>, Index range <>) of Element;

function "*" is new Matrix_Product(Element,Index,Matr);
-- operator overloading is possible at instantiation
   A : Matr(1..2,1..2):=((1,1),(1,1));
   B : Matr(1..2,1..2):=((1,2),(3,4));
   C : Matr(1..2,1..2); -- this will be the result
package Int_IO is new Integer_IO(Integer); use Int_IO;

procedure Put(A : Matr) is
begin
   for I in A'First(1)..A'Last(1) loop
     for J in A'First(2)..A'Last(2) loop
        put(A(i,j));
     end loop;
     New_Line;
   end loop;
end;

begin
   Put("The first matrix:"); New_Line;
   Put(A);
   Put("The second matrix:"); New_Line;
   Put(B);
   C:=A*B;
   Put("The product:"); New_Line;
   Put(C);
end;
```

Operator overloading is not allowed at generic functions, if we would like to create a generic "*" function, we have to put it into a generic package, as in the following Ada programs:

```
generic
  type Elem is private;
  type Index is (<>);
  type Matrix is array (Index range <>, Index range <>) of Elem;
```

```
  with function "+"(A, B : Elem) return Elem is <>;
  with function "*"(A, B : Elem) return Elem is <>;
package Matrix_Product2 is
   function "*"(A, B : Matrix) return Matrix;
end Matrix_Product2;
```

The body:

```
package body Matrix_Product2 is
function "*"(A, B:in Matrix) return Matrix is -- the same body as above
    L : Index := A'First(2);
    C : Matrix(A'Range(1), B'Range(2));
begin
   if A'First(2) /= B'First(1) or A'Last(2) /= B'Last (1) then
      raise Constraint_Error;
   end if;
   -- we suppose that the ranges are the same in A and B
    for I in A' Range(1) loop
      for J in B' Range(2) loop
        C(I,J) := A(I,L) * B(L,J);
        for K in Index'Succ(L)..A'Last(2) loop
          C(I,J) := C(I,J) + A(I,K) * B(K,J);
        end loop;
      end loop;
    end loop;
   return C;
   end "*";
end Matrix_Product2;
```

Using this solution, we have to instantiate the package, and then the overloaded "*" function for matrices will be available.

In the C++ solution we can use a template for our matrix class, operator overloading is possible, the following code snippet gives the result:

```
template <class T>
class matrix{
    public:
    //constructors are needed, etc...
   private:
    // the representation should be private
    int rows;
    int cols;
    T** m;
public:
 matrix operator*(const matrix& b) {
    // check if the dimensions match
    if (cols == b.rows) {
      matrix result(rows, b.cols);
      for (int i = 0; i < rows; i++) {
        for (int j = 0; j < b.cols; j++){
          for (int k = 0; k < cols; k++) {
            result.m[i][j] += m[i][k] * b.m[k][j];
          }
        }
      }
     return result;
    }
    else {
      throw "Dimensions do not match";   }
  };
//...
}
```

Another possibility is in C++ the friend. In this case we have to write the header of the function into the matrix class:

```
friend matrix operator*(const matrix& a,
                        const matrix& b) ;
```

The friend function must have two parameters, thus we have to modify the function's body too:

```
matrix operator*(const matrix& a, const matrix& b) {
    if (a.cols != b.rows)
        throw "Dimensions do not match";
    matrix result(a.rows, b.cols);
    for (int i = 0; i < a.rows; i++)
      for (int j = 0; j < b.cols; j++)
        for (int k = 0; k < a.cols; k++)
          result.m[i][j] += a.m[i][k] * b.m[k][j];
    return result;
  }
```

Solution 7.9. The solution is in Ada, the result of the test gives also the comparison of the two possibilities.

```
with Text_IO;
with Ada.Calendar;
use Text_IO;
use Ada.Calendar;
procedure Fibonacci_Test is
   package Nat_IO is new Integer_IO(Natural);
   use Nat_IO;
   package Fl_IO is new Float_IO(Float);
   use Fl_IO;

    N : Natural;
    N1, N2 :Natural;

 function Fibonacci_Recursive(N:Natural) return Natural is
  begin
   if N=0 or N=1 then
     return N;
   else
     return
      Fibonacci_Recursive(N-1)+Fibonacci_Recursive(N-2);
   end if;
 end Fibonacci_Recursive;

 function Fibonacci_Iterative(N:Natural) return Natural is
    Act_Fib : Natural:=0;
    Prev : Natural:=1;
    Next :  Natural;
  begin
    for I in 0..N-1 loop
      Next := Act_Fib + Prev;
      Prev := Act_Fib;
      Act_Fib := Next;
    end loop;
    return Act_Fib;
```

```
 end Fibonacci_Iterative;

   C1, C2 : Time;

begin
   Put_Line("Give the value of n:");
   Get(N);
   Put("The value of the ");
   Put(N); Put_Line("th Fibonacci number:");

   C1:=Clock;
   N1:=Fibonacci_Iterative(N);
   C2:=Clock;

   Put(N1);
   New_Line;
   Put("The duration of the ");
   Put_Line("iterative Fibonacci algorithm:");
   Put(Float(C2 - C1));

   New_Line;
   Put("The value of the ");
   Put(N); Put_Line("th Fibonacci number:");

   C1:=Clock;
   N2:=Fibonacci_Recursive(N);
   C2:=Clock;

   Put(N2);
   New_Line;
   Put("The duration of the ");
   Put_Line("recursive Fibonacci algorithm:");
   Put(Float(C2 - C1));

end Fibonacci_Test;
```

The result on the screen is:

```
Give the value of n:
        40
The value of the         40th Fibonacci number:
  102334155
The duration of the iterative Fibonacci algorithm:
 1.28500E-05
The value of the         40th Fibonacci number:
  102334155
The duration of the recursive Fibonacci algorithm:
 3.73950E+00
```

Solution 7.10. The Ada solution is a generic procedure (or similarly a function). We can prescribe the needed ”+” operation and Zero value.

A possible specification is the following:

```
generic
   type Item is private;
   type Index is (<>);
   type Vector is array (Index range <>) of Item;
   Zero:Item;
   with function "+"(X, Y : Item) return Item is <> ;
procedure Sum(V:in Vector; X: out Item);
```

The body:

```
procedure Sum(V:in Vector; X: out Item) is
begin
   x:=Zero;
   for I in V'Range loop
      X := X + V(I);
   end loop;
end Sum;
```

A simple example for the instantiation with the Rational type:

```
with Sum, Rational_Numbers, Text_IO;
use Text_IO, Rational_Numbers;
procedure Sum_Trial is
   type Myvect is array(Integer range <>) of Rational;

   procedure MySum is new Sum(Rational, Integer, Myvect, Zero);

   V: Myvect(1..3):=(Rat(1,2),Rat(2),Rat(2,3));
   Result:Rational;
   begin
      Put("The elements of the vector: ");
      for I in V'Range loop
         Put(V(I)); New_Line;
      end loop;

      MySum(V,Result);

      Put("The sum of the elements: ");
      Put(Result);
end Sum_Trial;
```

The C++ template cannot check the existence of the zero-element or the "+" operator, the template can be used only if beside the "+" operation standard conversions exist to the 0 value of the element-type.

A possible solution is:

```
#include <iostream>
using namespace std;

template <class T>
T sum(const T *v, const int size){
  T s=0;
  for (int i=0; i<size; i++){
    s=s+v[i];
  }
  return s;
}

int main() {
  int vect[4]={2,3,4,5};
  float vect2[3]={2.3,3.1,2.5};
  int sum1;
  float sum2;
  sum1=sum<int>(vect,4);
  cout<<sum1<<"\n";
  sum2=sum<float>(vect2,3);
  cout<<sum2<<"\n";
}
```

Solution 7.11. The parameter to the factorial function is passed by reference but is used as a pure in parameter. Within the function body the formal parameter is used in the computation, so this will change also the value of the calling variable.

This is an unintended side-effect of the function. As the involved variable is also printed out together with the result of the function, the evaluation order of the output stream operator is not defined. This is why we let the result display twice, as the second line will definitely have incorrect values, the first time the appearance of the unintended side-effect is implementation dependent.

Solution 7.12. Within the function body an uninitialized local variable is used in the computation, so no deterministic behavior can be expected.

Solution 7.13. *Integer* parameters are immutable wrapper objects, and the arithmetic multiplication is assisted by the so called autoboxing feature, which uses the int value of these objects, executes the multiplication, and stores the result in another new object. That is why after the multiplication within the called method the new value is seen (as the formal parameter now has a new reference), but as the caller still has the reference to the old object, which is left untouched (because it is immutable), the old value is printed again after returning from the *multiply* method.

8 Exception handling

In the software development process it is vital that the end-product is correct and reliable, satisfies the customer's requirements and does not have any unexpected side effects. Many programming languages have introduced exception handling and correctness proving tools to achieve the above goal. In this chapter, we present the means and procedures of handling runtime errors.

The goal of exception handling is to deal with problems that make continuing the normal program execution hard or even impossible. Such problems are division by zero, nullpointer[1] dereference, unsuccessful type cast, fault in the operating system or hardware, or communication problem (the connection breaks or the remote endpoint violates the protocol). Exception handling provides more or less elegant ways to handle these situations by providing language elements that alter the control flow of the program.

8.1 Introduction

Already the very first high level programming language, that is, FORTRAN, provided solutions to handling runtime errors, even if these were limited to input and output errors. The first programming language to incorporate generic exception handling elements was PL/I. It was followed by (amongst others) CLU, Ada, C++, Eiffel, Delphi, Java and C#. Other languages have elements or library functions that can be used to implement exception handling (for example the *goto*[2] statement and the *setjmp* and *longjmp* functions of the standard library in C);[3] still, these cannot be treated as exception handling as they were not created for this purpose or their use is cumbersome.

8.1.1 Basic concepts

Even though exception handling varies from one language to the other, many of them share the same concepts (sometimes with different names).

Exception is a runtime error that breaks the normal execution of the program. Exception may be detected by hardware (or the underlying operating system) or by software (the program itself or its runtime execution environment). When

[1] *NULL* in C, *nullptr* (*0* before C++11) in C++, **null** in Java, *Void* in Eiffel
[2] See Section 3.8.
[3] See Section 3.8.4.

hardware detects an exceptional situation, it usually triggers an interrupt to the operating system, which in turn sends a signal to the application. Examples of the first type are, e.g. division by zero or reading from a file after its end. Examples of the second type are, e.g. indexing out of bounds of an array. The goal of exception handling is to correct these errors if possible, or to halt[4] the program with the least damage (by saving all data) if the error is not correctable.

Most languages provide ways to create user defined exceptions. To that end, either already existing language elements (e.g. classes, objects) are used or new language elements are introduced. In some languages the language itself contains exceptions,[5] while in other languages only the standard library provides predefined exceptions.[6] Exceptions may have a parameter when they are thrown (if the exception is an object, then this parameter could be the attribute of the object), while in other languages the exception cannot have a parameter.

Exceptions are usually thrown, but other languages use keywords such as *raise*, *signal* or *trigger*. In most languages, the language definition describes situations where the runtime environment throws exceptions[7] (especially due to a fault detected by hardware), but usually the programmer can also throw an exception. Thrown exceptions are caught and handled by exception handler language elements. There are many ways to finding the right exception handler for the given exception.

If an exception is not handled at the current logical level of the program, it can be propagated so that it is handled higher in the function call chain. This is a very important utility to achieving that exceptions are handled at the right logical level. For example, let us suppose that our program reads records of data from a file. In the program the "main program"[8] opens the file, then passes the file descriptor to the record-reading subprogram[9] to read the actual data. If the data in the file is invalid, then the exceptional situation is detected in the subprogram, but this subprogram does not have all the information necessary to handle the error (which in this case would be to print the filename and an error message). In this example it is better to let the exception propagate, so that the main program can handle it. In some languages the specification of the subprograms (functions, methods, etc.) contains the set of exceptions that can be thrown by the subprogram. Then either the compiler or the runtime execution environment can check that the subprogram really does not throw a different exception.

Exceptions break the normal execution of a program. However, to guarantee that the program is running after handling the exception, it is best to keep

[4] In this chapter, by halting we mean that the program halts due to an error.
[5] For example *CONSTRAINT_ERROR* in Ada95.
[6] For example *NullPointerException* in Java.
[7] For example *ArrayIndexOutOfBoundsException* in Java is thrown when the program attempts to use an index that is less than zero or greater than or equal to the length of the array.
[8] Function, procedure, etc. based on the language, see 7.3.2.
[9] See Section 7.7.

the invariants of the components (functions, classes).[10] The following levels of exception safety[Abr10] can be provided by components:

- No exception safety: when exception is thrown, "all bets are off", there is no guarantee for the program state;
- Basic: the invariants of the components are preserved (i.e. they can be used after the exception was thrown) and no resources are leaked;
- Strong (or transactional): the operation has either completed successfully or thrown an exception. In the latter case no state change happens (including side effects);
- No throw: the component does not throw any exception;

In recent languages it is usually possible to always execute a block of code, regardless of whether there was an exception or not. We call this the *finally* block. This helps in recovering allocated resources even in exceptional situations.

In this chapter, we will not discuss exception handling in logical languages. Exception handling in logical languages is described in Chapter 16, where the basic concepts are introduced.

8.1.2 Why is exception handling useful

If there is no elegant exception handling, the programmer must check for errors after each "dangerous" statement and handle the possible errors. Let us take a look at the next code segment written in language C [KR89], which copies the `copy.c` file line by line to the `copy.c.bak` file.

```
const int buflen = 512;
void copy(void) {
    FILE* input = NULL;
    FILE* output = NULL;
    char* line = NULL;
    line = (char*)calloc(1, buflen + 1); /* Allocate memory. */
    input = fopen("copy.c", "r");        /* Open files. */
    output = fopen("copy.c.bak", "w");
    fgets(line, buflen, input);          /* Read one line. */
    while (!feof(input)) {               /* Until we reach end of file. */
        fputs(line, output);             /* Write the line. */
        fgets(line, buflen, input);
    }
    fclose(input);                       /* Close the files. */
    fclose(output);
    free(line);                          /* Free allocated memory. */
}
```

[10] See Section 12.3.8.

This code satisfies the requirement, but what happens if, for example, the `copy.c` file cannot be opened? The value of the *input* pointer[11] will be *NULL* and at the first dereference the program will halt with "Segmentation fault"[12] (or its equivalent). This is not something what we consider reliable software.

In language C we can have multiple correct solutions for this requirement. One is to check for errors after every "dangerous" statement and handle the errors there. Another solution is to jump to a common error handling code using the **goto** statement. We could also use the *setjmp* and *longjmp* functions, but these would greatly decrease code readability and understandability.

The first solution is as follows (the *copy* function returns an error code if one of the steps have failed in the algorithm and the global *errno* variable will also contain the error-specific value):

```
const int buflen = 512;
int copy(void) {
    FILE* input = NULL; FILE* output = NULL;
    char* line = NULL; char* read;
    int written;

    line = (char*)calloc(1, buflen + 1);
    if (NULL == line)
        return NO_MEMORY;
    input = fopen("copy.c", "r");
    if (NULL == input) {
        free(line);
        return CANNOT_OPEN_INPUT;
    }
    output = fopen("copy.c.bak", "w");
    if (NULL == output) {
        free(line);
        fclose(input);
        return CANNOT_OPEN_OUTPUT;
    }
    read = fgets(line, buflen, input);
    if (NULL == read) {
        free(line);
        fclose(input);
        fclose(output);
        return READ_ERROR;
    }
```

[11] See Section 5.6.

[12] Segmentation fault is a well known error message of UNIX or UNIX-like operating systems in this situation. On Windows it is equivalent to "General Protection Fault".

```
    while (!feof(input)) {
        written = fputs(line, output);
        if (written < 0) {
            free(line);
            fclose(input);
            fclose(output);
            return WRITE_ERROR;
        }
        read = fgets(line, buflen, input);
        if (NULL == read) {
            free(line);
            fclose(input);
            fclose(output);
            return READ_ERROR;
        }
    }
    fclose(input); fclose(output);
    free(line); return OK;
}
```

In the code example above we can hardly see what the program really does due to the error handling. Moreover, nearly each error handling block does the same: frees the allocated resources and returns an error code. It would be much better to handle the similar errors at the same place.

In C with the **goto** statement we can put the error handling in one place:

```
const int buflen=512;
int copy(void) {
    FILE* input = NULL; FILE* output = NULL;
    char* line = NULL; char* read;
    int written;
    int error;

    line = (char*)calloc(1, buflen + 1);
    if (NULL == line) {
        error = NO_MEMORY;
        goto err;
    }
    input = fopen("copy.c", "r");
    if (NULL == input) {
        error = CANNOT_OPEN_INPUT;
        goto err;
    }

...
```

```
    while (!feof(input)) {
        written = fputs(line, output);
        if (written < 0) {
            error = WRITE_ERROR;
            goto err;
        }
        read = fgets(line, buflen, input);
        if (NULL == read) {
            error = READ_ERROR;
            goto err;
        }
    }
    error = OK;
err:
    if (NULL != input) fclose(input);
    if (NULL != output) fclose(output);
    if (NULL != line) free(line);
    return error;
}
```

In comparison to the previous solution, the statements that restore the state are at the same place now in the code, but we still need to check the return value from each function, and that makes the code hard to read.

The root cause of the problems in the previous solutions is that we have to add code to check for errors right after each "dangerous" statement, because the language does not provide a way to handle the errors at a different place. Another problem is that the different functions report the errors in different ways. Some return *NULL*, others some negative value. Although the C standard library wors in this way, in some cases this method is insufficient.

Let's see how we could implement this function in language C extended with an imaginary exception handling. In this imaginary exception handling exceptions thrown in the **try** block are handled in the following **catch** blocks. The **finally** block after the **catch** is always executed, whether or not an exception occurs (this exception handling is analogous to the Java language, see Section 8.3.6). In this imaginary extension the used library functions also throw *MemoryException* and *IOException* on runtime errors:

```
int copy(void) {
  FILE* input = NULL; FILE* output = NULL;
  char* line = NULL;
  int ret;
  try {
    line = (char*)calloc(1, buflen + 1);
    input = fopen("copy.c", "r");
    output = fopen("copy.c.bak", "w");
```

```
    fgets(line, buflen, input);
    while (!feof(input)) {
      fputs(line, output);
      fgets(line, buflen, input);
    }
  }
  catch (MemoryException e) { ret = NO_MEMORY; }
  catch (IOException e) { ret = e.error_code; }
  finally {
    if (NULL != line) free(line);
    if (NULL != input) fclose(input);
    if (NULL != output) fclose(output);
    return ret;
  }
}
```

It can be seen that the statements in the **try** block are exactly the same statements as in our first example without error handling, so the code is as readable as the first version, but in this case we handle the errors. If we do not want to handle the errors here, we may propagate these exceptions. Without exception handling this could be achieved in a more complicated way by returning error codes, possibly through many function calls. Another advantage of this solution is that the allocated resources are always freed, even if the code is extended later, because the **finally** block always executes.

A third advantage of exception handling is that the return value of a subprogram is differentiated from an error code. Consider the following C example: the parameter string is parsed as an integer and its value increased by one is returned.[13] This example uses the *atoi* function from the C standard library:

```
int func(const char* s) {
  return atoi(s) + 1; /* no error handling! */
}
```

This function works well as long as the parameter content is really an integer. However, when the input cannot be parsed as an integer, the function cannot report an error, because the *atoi* function cannot use the usual convention of the C standard libraries, the *-1* return value.[14] Actually all integer values could be valid return values, so the *atoi* function cannot report an error at all.

There are many ways to solve this problem. One way is to return a struct[15] instead of the integer value, with one attribute holding the return value and the other the error code. This is slightly inconvenient, because the function cannot be executed directly in an expression that expects an integer value.

[13] The possible integer overflow is ignored for this example.
[14] *-1* is a valid return value for the `"-1"` input.
[15] See Section 6.3 as "cartesian product type".

```
struct atoi_result {
  int value;
  int error;
};
struct atoi_result atoi2(const char* s);

int func(const char* s) {
  struct atoi_result result;
  result = atoi2(s);
  if (!result.error) return result.value + 1;
  /* error handling */
}
```

Another solution is to pass the integer value by address[16] as another parameter. This is also a little inconvenient as we need to create a pointer from our integer value, and these kinds of side effects[17] make understanding the program harder. Another solution is to pass the error value in the parameter, but understanding that code would be just as complicated.

```
int atoi3(char* s, int* i);
int func(const char* s) {
  int i;
  if (-1 != atoi3(s, &i)) return i + 1;
  /* error handling */
}
```

Actually the `strtol` function in the C standard library implements a similar solution: a pointer has to be passed by address to the function; in case of an error the pointer will point to the start of the string.

There is a third solution for this problem, which involves a global variable[18] to maintain the success state (there was an error or there was not) of the last function call. However, this creates problems in multithreaded environment[19] where different threads might write the same variable.

```
extern int atoi_error;
int atoi4(const char* s);
int func(const char* s) {
  int i;
  i=atoi4(s);
  if (!atoi_error) return i+1;
  /* error handling */
}
```

[16] See Section 7.4.
[17] See Section 7.3.7.
[18] See Section 4.2.1.
[19] See Chapter 13.

Many functions in the C standard library use the common global `errno` variable for more detailed error description. Apart from the problems in a multithreaded environment, there is an additional problem with this approach: subsequent function calls will overwrite the error description in the common global variable so that the original cause of the problem may get lost, unless special care is taken.

If the *atoi* function threw an exception, these problems would not occur as the return value would be well differentiated from the error.

In object-oriented languages[20] the constructors[21] are similar to the function presented above, since they can only return the constructed object (or in case of error, a null pointer or its equivalent) which in many cases is not sufficient for error handling (there can be many kinds of errors in a complicated constructor). Exception handling is useful in this case too.

8.1.3 The aspects of comparing exception handling

In the following sections we will compare the exception handling mechanisms of various languages. To that end, we first need to agree on the aspects on which to base our comparison.

- What kind of language element is the exception? Is it possible to group exceptions? Is it possible to organize exceptions into hierarchies? Organizing exceptions into groups or into a hierarchy is useful in order to decide what exceptions are ”interesting” (should be handled on the current logical level of the program) and what exceptions are not ”interesting” (should be handled on a different level).
- To which language element is exception handling connected (statement, block, subprogram, other)?
- After handling an exception, at which point does the normal execution of the program continue - is it where the exception was thrown, is it after the exception handler or is the block ”retried” (executed again) where the exception was thrown?
- Are exceptions propagated? What happens with the unhandled exceptions? Can the exception handler throw an exception?
- Can the exception have a parameter? The type of the exception might not have enough information to properly handle the error.
- Is there a way to have a piece of code that *always* executes, both in the exceptional and in the normal situation (in other words, is there a *finally* block)? This only makes sense if exception handling is connected to a larger unit than a single statement.

[20] See Chapter 10.
[21] See Section 10.3.

- Is it possible, and if so, is it mandatory to specify the possibly thrown exceptions? It is important that a language forces the programmer even at compile time to handle the exceptions, or else the exceptions are detected only at runtime, during testing.[22]
- How are exceptions handled in a multithreaded environment? This question only makes sense if the language supports multithreaded execution.
- If an exception is represented by an object, what is its scope and lifetime?

8.2 The beginnings of exception handling

8.2.1 Exception handling of a single statement: FORTRAN

FORTRAN [LV77], developed in the 1950s is considered to be the first high level programming language. It was designed for numerical computations. FORTRAN provided tools for handling runtime errors, though these dealt with errors during input-output operations only. These operations can have an `err` parameter which contains the label of the error handling statement:

```
      program hello
      open(file='testfile', err=100, unit=2)
      write(*,40)
40    format('Hello World!')
      goto 120
c
100   write(*, 110)
110   format('Error!')
120   end
```

If the above example is run and the `testfile` file is readable, the program will print `Hello World!`. If the file is not readable, the program will print `Error!`.

As shown above, this is not "real" exception handling; it only offers some statements for elegant error handling.

8.2.2 Exception handling of multiple statements: COBOL

COBOL [Bak74] was also developed during the 1950s. In contrast to FORTRAN, COBOL was designed to write finance-related software. In the field of error handling, COBOL was more advanced than FORTRAN. The language provides two kinds of language elements for error handling: after some (especially arithmetical) statements it is possible to specify a label where the program execution continues after an error, and there is also a more generic way to handle input-output errors.

[22] Or even worse, by the user.

The following example describes the first kind of error handling:

```
DIVIDE Num1 BY Num2 GIVING Num3 REMAINDER Num4
  ON SIZE ERROR DISPLAY "Error".
```

The statement above divides `Num1` by `Num2` and the quotient is put into `Num3`. If there is an error (e.g. either `Num2` is 0, or the quotient or the remainder does not fit into `Num3` or `Num4`), then the program will print "Error". This is very similar to the statement-level error handling seen in FORTRAN. However, there is a more generic error handling solution for input-output errors. In the section containing the procedures (*PROCEDURE DIVISION*), between the keywords *DECLARATIVES* and *END DECLARATIVES* we can assign an error handler for each file used:

```
PROCEDURE DIVISION.
DECLARATIVES.
Error SECTION.
    USE AFTER EXCEPTION PROCEDURE ON OwnFile.
ErrorHandling2.
    DISPLAY "Error".
END DECLARATIVES.
```

This is substantially more advanced than the solution in FORTRAN, because we do not have to specify the error handler for each input-output statement; but rather it is enough to specify this at one place. So this way the programmer will not forget to handle the errors in new code.

However, this is still not generic exception handling as it can only be used for input-output and there is no way to use user defined exceptions.

8.2.3 Dynamic exception handling: PL/I

PL/I [GI90] was the first programming language which introduced generic exception handling in the 1960s. Thus this mechanism is rather rudimentary.

In PL/I the exceptions have a name (a label), so there is no way to group them or give parameters to them. There are some predefined exceptions in the language itself and the programmer can create new exceptions using the `CONDITION` keyword. The *ON* statement is used to specify which statement to run if the given exception is thrown after the *ON* statement is executed. Exceptions can be thrown by the `SIGNAL` statement:

```
ON CONDITION(OWNEXCEPTION) PUT LIST('A');
SIGNAL CONDITION(OWNEXCEPTION);
```

When an exception is thrown, the currently applicable exception handler is called, so the exception handling is connected to statements. An interesting language feature is that for single statements exception handling can be enabled (or disabled) for certain exceptions. Exception handling can be overridden and with the `REVERT` statement we can rollback to the previous exception handler:

```
P: PROC;
  /* ... */
  ON ZERODIVIDE PUT LIST('A');
  N=0;
  X=X/N;              /* prints 'A'. */
  BEGIN
    ON ZERODIVIDE PUT LIST('B');
    X=X/N;            /* prints 'B'. */
    REVERT ZERODIVIDE;
    X=X/N;            /* prints 'A'. */
  END;
  ON ZERODIVIDE PUT LIST('C');
  X=X/N;              /* prints 'C'. */
  (NOZERODIVIDE):
  X=X/N;              /* No exception thrown. */
END;
```

After the exception handler has finished, the execution continues with the statement after the one that threw the exception, unless there was a `GOTO` statement in the exception handler. Since the execution continues after the statement that threw the exception, there is no exception propagation, and therefore, the exceptions cannot be specified.

As mentioned before, when an exception is thrown, always the currently applicable exception handler is executed and always the exception handler set by the last `ON` statement is the applicable:

```
  ON ZERODIVIDE PUT LIST('A');
  N=0;
  IF X = 10 THEN /* if X is 10, prints 'A'. */
    GOTO LABEL1; /* otherwise prints 'B'. */
  ON ZERODIVIDE PUT LIST('B');
LABEL1:
  X=X/N;
```

8.3 Advanced exception handling

8.3.1 Static exception handling: CLU

CLU [Lis81] was developed long after PL/I,[23] so its exception handling is a lot more advanced, though a little strange compared to contemporary languages.

In CLU the exceptions are not objects, but expressions, so they cannot be grouped. The *`except`* clauses are used to handle exceptions. In these clauses

[23] See Section 8.2.3.

those exception have to be specified that the programmer wants to catch, or the *others* keyword can be used to catch all exceptions. There are two kinds of exceptions in the language that can be thrown by the *signal* or *exit* statements. The first kind of exceptions are connected to subprograms, while the second kind is connected to statements. Exceptions thrown by *signal* return from the subprogram to the caller. Exceptions thrown by *exit* have to be handled in the same subprogram where they were thrown (there has to be an exception handler is the same subprogram that handles the exception). In this case the program continues after the exception handler. Unlike PL/I, in CLU the compiler can decide which exception handler catches which exception.

If an exception is thrown from a subprogram that is not handled in the same subprogram, then a `failure` exception is thrown, so unlike in newer languages, exceptions in CLU do *not* propagate. The reason for this is that in this way the programmer does not have to know the implementation of the subprogram (e.g. what other subprograms are called and what exceptions are thrown) in order to know what kind of exceptions can be thrown in there (see [Lis93]). Of course, the exceptions that we want to propagate can be caught in the exception handler and can be rethrown.

The exceptions thrown by *signal* have to be specified in the subprogram header after *signals* keyword, except the predefined `failure` exception.

Because the exception is an expression, we can pass a parameter to the exception, as shown below:

```
P1=proc() signals(OwnException(string))
  signal OwnException("Trouble!")
end P1

P2=proc()
  P1()
  except
    when OwnException(s:string)
    % At this point the value of s is "Trouble!".
  end
end P2
```

There is no language construct in CLU that is executed both in normal and exceptional situations (i.e. there is no `finally` block). Since the exceptions thrown by *signal* stop the execution of the subprogram, we cannot work around to create such construct.

8.3.2 Exception propagation: Ada

The first version of Ada ([DG80] and [Ada83]) was standardized in 1983. It was then updated in 1995, when many new elements were added to the language. The latest standard was accepted in 2012.

Ada uses the **exception** language element to represent exceptions. There are four predefined exceptions in the language:

- *CONSTRAINT_ERROR*: thrown when a constraint, for example, the range of a type is violated;
- *PROGRAM_ERROR*: thrown when the execution structure of the program is violated, for example, a function tries to return without a return value;
- *STORAGE_ERROR*: thrown when the program runs out of memory;
- *TASKING_ERROR*: thrown due to task-related problems.

As a consequence, Ada does not provide a way to organize the exceptions into hierarchies. Exceptions can be thrown by the **raise** ⟨*exception-name*⟩ statement or by using the *Raise_Exception* subprogram from the **Exception** package.

Exception handling in Ada is connected to blocks. All blocks can throw exceptions, which can be caught by **when** clauses after the **exception** keyword at the end of the block. An exception handler without an explicit block can be put after the body of a subprogram, body of a package, body of a task or an **accept** block. In the **when** clauses of the exception handler, the types of the handled exceptions (more than one can be used and the **others** clause can catch all previously unhandled exceptions, just like in CLU) and an identifier can be specified. The identifier is a variable of *Exception_Occurrence* type and will get the value of the thrown exception. The operations of this type can be used to elicit information about the exception (e.g. its parameters). The execution does not return to the block where the exception was thrown and the language does not provide a tool to retry the block that threw the exception. For a complete example see Section 8.5.3.

One of the interesting properties of Ada is that the declaration parts of subprograms, blocks, packages and tasks are evaluated partly at runtime, meaning the exceptions can be thrown from these parts too. In the first three cases the exception is propagated the same way as the exception thrown from the subprogram, block or package body. If the exception is thrown from the declaration part of a task, then the task becomes complete[24] and the predefined *TASKING_ERROR* is thrown from the activating point of the task.

If a block throws an exception and that block does not handle the exception, the exception is propagated without change to the higher level (this is a new feature compared to CLU). If the exception reaches the main program and that does not handle the exception either, the program halts. If the exception is thrown in a task and it is not handled anywhere, the task becomes complete (its execution stops), but the exception does not propagate to an other task or to the main program.

The exception handler can also throw an exception. This is useful, for example, when we know that the caught *CONSTRAINT_ERROR* signals an indexing

[24] see Section 13.10.1.

error. In such cases we can throw our own exception representing an indexing error from the **when** *CONSTRAINT_ERROR* clause. The caught exception can be thrown again with a **raise** statement without a parameter. This makes sense in the **when others** clause which is executed when an unexpected exception is caught. In this case this clause can clean up as much as possible, and then rethrow the exception.

While exceptions cannot have a parameter in Ada 83, later versions introduced the *Exception_Occurrence* type and its related subprograms. From these subprograms the *Raise_Exception* can be used to set a parameter in the exception, which can then be queried in the exception handler.

There is no explicit *finally* in Ada, but with two nested blocks we can achieve this (the exception handler of the outer block will have the code that has to run always). Consider the following example:

```
procedure P is
  occ:Exception_Occurrence;
  was_exception:boolean:=false;
  begin
  -- Allocating resources
    begin
    -- Executing statements.
    exception
    when Exception1 | Exception2 =>
      -- Expect and handle these exceptions.
    when e:others =>
      -- Rethrow the others. Save the exception,
      -- then rethrow it after the resources are
      -- freed.
      Save_Occurrence(occ,e);
      was_exception:=true;
    end;
  -- Freeing the resources. This would be the finally block in Java.
  if was_exception then
    -- Rethrow the saved exception.
    Reraise_Occurrence(occ);
  end if;
end P;
```

It is not a very elegant solution though, as the programmer has to explicitly save and rethrow the exception. Not even this workaround was available in Ada 83.

In Ada it is not possible to specify which exceptions are thrown by a subprogram. It means that if the programmer forgot to handle an exception, it will be found only at runtime when the exception is thrown, but not caught, and the

program halts. Since only the documentation can tell what kind of exceptions can be thrown by a subprogram, the compiler cannot, this is a drawback.

8.3.3 Exception classes: C++

The development of language C++ [Str00] started in the early 80s, just like that of Ada, and still continues to this date, the latest standard was accepted in 2011. The exception handling of the language improved significantly during its development, partly due to influences from Ada[25] and CLU.[26]

In C++ all variables can be thrown as an exception. It means that we can define classes for the exceptions and can organize them into hierarchies. For example, standard C++ libraries only throw exceptions that are descendants of the *std::exception* class. The **throw** keyword can throw an exception. Since the exception can be an object, it can carry parameters. If the exception is a simple type, its value can be regarded as the exception parameter.

Exception handling is related to blocks in C++ too. An exception thrown in a **try** block can be caught in the following **catch** block. Exception handling looks like the following:

```
try {
    // Exception can be thrown.
}
catch (type [name]) {
    // Exception handling.
}
```

There can be more than one **catch** block for a **try** block, each can only catch the exception of the specified type (bear in mind the automatic type conversions in C++). If a name is specified in the **catch** block, this variable will get the value of the caught exception. If `...` is specified instead of a type and name, then this clause will match all exceptions:

```
try {
    // All kind of exceptions can be thrown.
}
catch (...) {
    // All exceptions caught.
}
```

After handling the exception the execution does not return to the block where the exception was thrown from. This also means that the destructors of the local objects specified in the **try** block are executed before the catch block. As a consequence, the programmer has to be very careful before throwing an exception from a destructor - when the destructor is called, an other exception

[25] See Section 8.3.2.
[26] See Section 8.3.1.

may be in "thrown" state. In this case, the program halts as there can be only one exception thrown at a time. The **throw** statement without parameters rethrows the caught exception.

The unhandled exception is propagated to the caller function. If the *main* function (the main program where the execution starts) does not handle the exception either, the *terminate* function is called which by default calls the *abort* function to halt the program. The *terminate* function can be overridden by calling the *set_terminate* function, but this (overridden) function still has to halt the program (more precisely: it is not defined what happens when the function returns).

Using POSIX threads, the unhandled exceptions thrown in a thread stop only the given thread, not the whole program. Also, these exceptions do not propagate to other threads. However, there are other libraries for C++ that introduce multithreading, and they might handle the exceptions differently.

The language C++ does not provide an explicit tool for *finally*. It is not enough to nest the blocks (just like in Ada), as there is no "exception occurrence" type, meaning we cannot save and rethrow the exception. The pattern used to achieve *finally* is to put this code into the destructor of a local object. If the function returns (either in normal or exceptional way), the local objects are deleted and their destructors are called. This pattern is also known as "resource acquisition is initialization" [Str12]. Consider the following example:

```
class File {
    FILE* f;
public:
    // For simplicity's sake do not care about errors
    File(const char* filename) { f = fopen(filename); };
    ~File() { fclose(f); };
};
void func()
{
    File f("something.txt");
    try {
        // Exception might be thrown
    }
    catch(const FileException& fe) {
        // Handle one type of exception
    }
}   // File is always closed here!
```

As mentioned above, when an object is created on the stack (i.e. it is a local variable of a function), the object is deleted as soon as the function finishes. Thus if this object is thrown, the exception is deleted before it is caught. To avoid this from happening, a temporary copy is made from the exception object (by calling the copy constructor of its class) and this temporary copy is caught later

in the **catch** block. The temporary object is deleted when the execution of the program leaves the **catch** block. When the exception is caught by value instead of reference, an additional copy is made from the temporary object (and the copy constructor with the possible side effects is called again). If the exception object was created with the **new** statement, the exception object is allocated on the heap and does not get freed when the function returns or throws an exception. Therefore the programmer has to explicitly delete this exception object in the **catch** block.

In C++ functions can specify a set of exceptions that can be thrown from the function: in the header of the function after the **throw** keyword between parentheses the thrown exceptions can be listed. If there is no **throw** specification, all types of exception can be thrown from the function (due to C-compatibility). However, if there is a **throw** specification, only types specified there can be thrown. If a different exception is thrown, the *unexpected* function is called which halts the program. This function can also be overridden by using the *set_unexpected* function, but even the overriding function has to halt the program, it cannot return. It is possible to specify an empty set, in which case the function cannot throw any exceptions. The compiler cannot check if the specified exceptions are thrown or not. This feature was deprecated in the C++11 version of the C++ standard, the possibly thrown exceptions cannot be listed anymore; however, the new **noexpect** keyword can be used instead of **throw()** to tell the compiler that the function cannot throw any exceptions. The reason for this change is that the exception specifications in function types are not handled consistently in the language (e.g. they cannot be used in **typedef** constructs) and they also add some runtime overhead which in most cases is unnecessary [Sut09].

8.3.4 Exception handling and correctness proving: Eiffel

The language Eiffel [Mey91] was developed around the "Design by Contract" principle in the second half of the 1980s. This is one of the reasons why its exception handling is drastically different from the contemporary languages.

Exceptions are not objects of the Eiffel language. Even though there is a language element to catch exceptions (**rescue** clause), the only way to throw or get the attributes of the exceptions, is to inherit from the *EXCEPTIONS* class.[27] The reason for this is that according to the designers of the language, there is no point in changing the state of the exceptions, and thus it is not necessary to use objects as exceptions. Exceptions can be differentiated by exception code, which can be queried by the *exception* and *developer_exception_code* (in the case of programmer defined exceptions) methods inherited from the *EXCEPTIONS* class. These two codes are integers, meaning exceptions might be grouped (e.g. exceptions between 10 and 15 mark network errors); however the predefined exceptions of the language cannot be grouped in this way.

[27] Eiffel supports multiple inheritance, see Section 10.7.5.

The philosophy of the exception handling in Eiffel is different from the other languages discussed above: exception is thrown when the "contract" is violated between the subprogram and its caller, meaning exception handling is connected to subprograms. The "contract" is violated when one of the constraints (precondition,[28] loop invariant,[29] etc.) is false. Exceptions are also thrown when a *Void* valued variable is used in a way it should not be used, when the operating system signals an error, when a subprogram fails or when using the **raise** method.

Exception handling is connected to subprograms in Eiffel. It is placed in the **rescue** clause after the body of the subprogram. Its task is to retry the execution (and fulfil the contract) or to restore the class invariant and signal to the caller that the execution failed (this is called "organized panic"). In this case the subprogram fails, and an exception is thrown in the caller.

```
func: INTEGER is
  do
    -- Exception might be thrown here.
  rescue
    -- Exception has to be handled here.
  end;
```

For a complete example see Section 8.5.4.

The interesting characteristic of the exception handling of Eiffel is that it provides a way to restart the execution when there is an exception in the subprogram (e.g. if the subprogram means to read a number from the user, but gets something else instead, it can retry). The **retry** statement is used for this purpose and the fact that when retrying, local variables are not initialized again (this can be used to limit the number of retry attempts by keeping the current number of retry attempts or retry state in a local variable). In Eiffel the exception only "disappears" when the subprogram is successfully executed. If the **rescue** clause does not finish with a **retry** statement, there will be an exception in the caller. The fulfillment of the postcondition[30] of the subprogram is the task of the subprogram and not the task of the **rescue** clause.

Propagation of the exceptions in Eiffel is more akin to CLU than to its contemporary languages (C++, Ada). All classes have a *default_rescue* method. It is defined in the *ANY* class (all classes inherit from *ANY*) and its body is empty there. This method is called if there is an exception in a subprogram without a **rescue** clause, meaning there is no unhandled exception in Eiffel. The class of the subprogram can override the *default_rescue* method (for example to restore the class invariant). On the other hand, if the **rescue** clause cannot execute the subprogram successfully, not even for second (or later) attempt, it finishes, and an exception is thrown in the caller, for the execution of the subprogram was unsuccessful. In this case a new exception with possibly a new

[28] See Section 12.3.4.
[29] See Section 12.3.10
[30] See Section 12.3.5.

code is thrown, but this keeps the code of the original exception. If an exception is propagated in this way to the main program, and that does not handle the exception either, the program halts, and prints the exceptions that have led to this situation.

Exceptions in Eiffel can be thrown by the **raise** method inherited from the *EXCEPTIONS* class. The exception code and a string can be passed, which will later be printed when the program halts due to this exception. Apart from this, a context can be passed to exception too, which can be used for parametrizing the exception.

Since exception handling is connected to subprograms, not to blocks, we cannot use the solution described in Ada to implement a *finally*. The C++ solution cannot be used either, as Eiffel has automatic garbage collection and the programmer cannot write destructors. The *finally* block is usually used to restore the class invariant: in Eiffel the **rescue** clause is used for this purpose. For example, if the class invariant states that a certain file has to be closed after each operation, this file has to be closed in both the subprogram and in the **rescue** clause. It is useful to put these operations into the *default_rescue* method, and call it from the **rescue** clause and from the subprograms too.

In Eiffel exceptions are used for exceptional situations, do not occur during normal execution, they cannot be specified with subprograms. Exceptions are thrown when a constraint (e.g. a precondition) is violated, so good constraints and invariants should be specified instead of exception specifications. These constraints and invariants can be used to prove the correctness of the program.[31]

8.3.5 The *finally* block: Modula-3

Modula-3 [Nel91] was conceived as improvement on Modula-2 from the Pascal family tree. Its exception handling is similar to that of Ada,[32] but contains elements borrowed from C++[33] too.

Exceptions are represented by the **EXCEPTION** language elements (just like in Ada), so they cannot be grouped. Exceptions can be thrown by the **RAISE** statement. The interesting property of the Modula-3 language is that the **EXIT** statement (used to break out from a loop) is handled as if an *ExitException* was thrown and the loops have a predefined exception handler for this exception. Similarly, the **RETURN** statement (used to return from a subprogram) throws a *ReturnException*, its parameter is the return value. This exception is handled by the predefined exception handler of the subprogram.

Exception handling is connected to **TRY** blocks, just like in C++. Exceptions thrown in a **TRY** block can be caught in the following **EXCEPT** clauses. The **ELSE** clause following the **EXCEPT** clauses can catch all previously uncaught exceptions (just like in Ada). After the exception was handled, execution follows

[31] For more details see Chapter 12.
[32] See Section 8.3.2.
[33] See Section 8.3.3.

after the exception handler, there is no simple way to retry the execution. Exception have to be either caught or specified to be thrown. If an unspecified exception is thrown and not caught, the program halts, just like in C++. The serious runtime errors cannot be caught.

Modula-3 supports multithreading. If an exception is thrown in a task and is not handled, the whole program halts, in which sense it is different from Ada. Exceptions can have only one parameter.

Modula-3 introduced the **TRY**–**FINALLY** block: the **FINALLY** block is always executed, regardless of whether there was an exception in the **TRY** block or not. This is very useful, for example, if the programmer wants ensure that an opened file is closed even if there was an exception during processing the file. A **TRY** statement can have only an **EXCEPT** or **FINALLY** statement, not both at the same time. Below is an example:

```
MODULE Main;
IMPORT IO;
EXCEPTION ToBeThrown;
EXCEPTION SomethingElse;
BEGIN
  TRY
    TRY
      IO.Put("Hello!\n");
      RAISE SomethingElse;
    EXCEPT
      ToBeThrown =>
        IO.Put("Caught!\n");
    ELSE
      IO.Put("Something else\n");
    END;
  FINALLY
    IO.Put("Always executed!\n");
  END;
END Main.
```

8.3.6 Checked exceptions: Java

Java [Nyek08] was developed in the 1990s and still continues to this date, the latest standard was accepted in 2011. Much of the language was based on C++,[34] but regarding exception handling, most of the elements were taken from Modula-3 with some improvements.

In Java the exceptions are objects. All exception classes have to extend (inherit from) the *java.lang.Throwable* class. Only those objects can be thrown that are instances of a class extending *Throwable*. The *Throwable* class has

[34] See Section 8.3.3

two direct descendants, the *Error* and the *Exception* class. The descendants of the first class are not checked exceptions, meaning they do not have to be caught (the descendants of the *Error* should not be caught as for example the program cannot handle the *OutOfMemory* exception). These exceptions are similar to the runtime errors in Modula-3. From the descendants of the *Exception*, the *RunTimeException* and its descendants are also unchecked exceptions. These are usually not fatal errors, but errors that can happen in too many places, thus the programmer cannot be expected to always expect the *ArrayIndexOutOfBoundsException* and similar exceptions. Figure 8.1. describes this hierarchy.

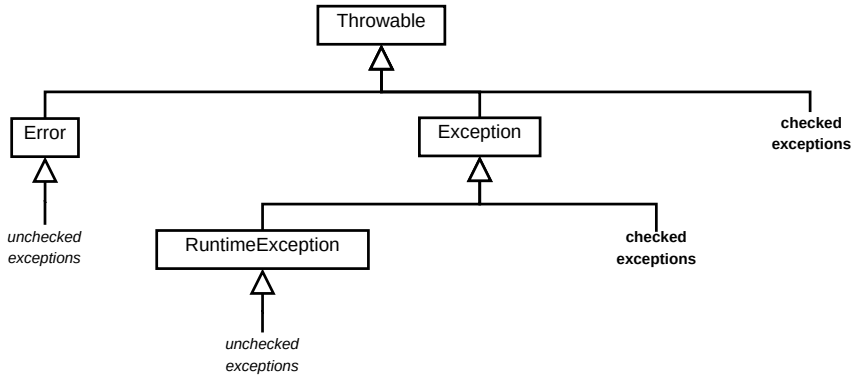


Figure 8.1: Inheritance hierarchy of exceptions in Java

Of course, unchecked exceptions can be caught too, and if an unchecked exception is not caught, the program halts (or if it happens in a thread, only the thread halts, the exception does not propagate to other threads). The other descendants of *Exception* and *Throwable* are all checked exceptions: if a method specifies that it throws a checked exception, that exception *has to* be handled in the calling method (or specified for the calling method to propagate that exception). This is checked by the compiler, not in runtime. This is a very powerful tool, which forces the programmer to handle cases when, for example, an URL object is created and the input to the constructor may be wrong. Exceptions can be thrown using the **throw** statement, just like in C++.

In Java (similarly to C++) exception handling is connected to blocks. Possibly exception throwing statements have to be nested into a **try** block and a **catch** block can catch the exceptions. The difference is that in Java it is mandatory to specify a variable name that will hold the exception value. In Java (unlike in C++) . . . cannot be written into the **catch** clause, but we can take advantage of the exception hierarchy and a **catch** *(Exception e)* { clause can be used. Java version 7 introduced the ability to catch multiple exception types with the same catch clause:

```
catch (NumberFormatException | SQLException ex) {
    /* ... */
}
```

Similarly to C++, the execution does not get back to the block that threw the exception, there is no simple way to implement an Eiffel-like retry mechanism (Java does not have a **goto** statement that might be used for a workaround in C++).

All **try** blocks can be followed by a **finally** block (known from Modula-3) after the **catch** blocks. The **finally** block is executed regardless of whether an exception was thrown or not, caught or not. Java 7 introduced the try-with-resources statement which provides an alternative way to handle resource allocation, using the "resource acquisition is initialization" pattern:

```
static String readFirstLine(String pathName) throws IOException {
  try (BufferedReader reader =
    new BufferedReader(new FileReader(pathName)))
  {
    return reader.readLine();
  }
}
```

In the above example, the opened file will be closed always, even if the *readLine* method throws an exception. This statement can only be used to handle resources wrapped in objects that implement the *AutoClosable* interface.

8.3.7 The exception handling of Delphi

Delphi [Lis00] was developed in the 1990s as an object-oriented expansion of Turbo Pascal. Its exception handling is similar to its "relative", Modula-3.[35]

Like in Java,[36] the exceptions in Delphi are objects, their classes are descendants of the *Exception* class, so they can be grouped by the class hierarchy. Exception handling is also connected to blocks. Exceptions thrown in **try** blocks can be caught in the following **except** blocks. The **except** block handles all exceptions in **on** clauses. In an **on** clause, similarly to C++ a type or a type and an identifier can be specified (when there is an identifier specified, it will get the value of the exception). At the end of the **except** block there can be an **else** clause, which is executed when no **on** clause handles the exception (this is equivalent with the **on** *Exception* clause). After the exception is handled, the execution continues after the exception handler, and there is no Eiffel-like **retry**.[37]

The uncaught exception is thrown again in the caller subprogram, so the exceptions are propagated. Exceptions can be thrown from exception handlers

[35] See Section 8.3.5.
[36] See Section 8.3.6.
[37] See Section 8.3.4.

and the currently handled exception can be rethrown by a parameter-less `raise` statement. If an exception is not handled at all, the program halts. If there is an unhandled exception in one thread of a multithreaded program, only this specific thread halts, like in Java. Unlike Modula-3 and C++,[38] in Delphi there is no way to specify in the function header which exceptions are thrown by a method.

A **try** block can have not only an **except**, but also a **finally** block; however, only one of them at a time (like in Modula-3). This works similarly to Modula-3: the **finally** block is executed regardless of whether an exception was thrown or not, caught or not. The **try**–**catch**–**finally** construct of Java can be implemented by two nested **try** blocks:

```
try
try
  ... { statements }
except
  on e: Exception do
    ... { exception handling }
end
finally
  ... { this is always executed }
end
```

8.3.8 Nested exceptions: C#

C# [Sch02] was developed in the late 1990s as a kind of a response to Java.[39] Subsequently its exception handling is very similar to both C++ and Java.

Similarly to Java, exceptions are objects in C# too. The classes of the exceptions have to inherit from the *System.Exception* class. A remarkable property of this class is that it has an *InnerException* field which contains the exception that led to the actual exception thrown (or **null**, if there was no such exception). For example, if the static constructor of an object throws an exception and it is not caught by the constructor, at the point where the constructor was called, a *System.TypeInitializationException* exception is thrown, and its *InnerException* field contains the original exception thrown by the constructor. This way an arbitrary number of exceptions can be chained together.[40] Two other useful attributes of a C# exception are the *StackTrace* and *TargetSite* fields. The former contains the call chain (in string) at the point when the constructor was thrown, the latter points to the method that threw the exception.

Exceptions can be thrown by the **throw** statement and can be caught by using **try** and **catch** blocks in the same way as in C++ or Java. C# also has a **finally**

[38] See Section 8.3.3.
[39] See Section 8.3.6.
[40] This is similar to the *getClause()* method in Java exceptions.

block which works the same way as its Java counterpart. In C# the compiler emits an error if there is an unreachable **catch** clause, because a previous clause matches a parent (or great-parent, etc.) class, just like in Java or C++.

Unlike in Java, there are no checked exceptions in C#. This was a conscious choice based on experience with checked exceptions in Java programs [EH03]. It turned out that in many cases the caller does not care about the exception type. One of the strengths of exception handling is that the actual error handling can be several layers above (in the function call chain) the place where the error happens - however, with checked exceptions, all intermediate layers have to (at least) specify all possible exceptions coming from the lowest layers. This leads to a meaningless **throws** *Exception* definition in many methods.

8.3.9 Exception handling with functions: Common Lisp

The first version of the Lisp language was created in the 1950s and has been developing ever since. Exception handling was introduced later, for example Common Lisp [Ste90], standardized in 1994, has exception handling too.

Lisp is a functional language and works in the sense of "everything is a function". Thus exception handling is implemented using two special functions and there are no extra language elements. Still, it is worth checking this solution, for the design is somewhat different from the solutions shown above.

In Lisp an exception is represented by an atom (an atom can be thrown or caught). Exceptions cannot be grouped and one exception handler can catch one kind of exceptions only. The exception atom will get the value of the exception and it is returned by the exception handling `catch` function. If an exception is not caught, it is propagated up through the call chain. If the main program does not catch the exception either, the program halts.

Due to the peculiarities of the language syntax, exception handling has to be written before the exception is thrown. A `throw` function inside the `catch` function can throw the exception:

```
(catch 'faultfault
  (print "All is well so far")
  (throw 'fault "There is trouble!")
  (print "Not executed."))
```

While interpreting the above function the Lisp interpreter will print the "All is well so far" string, but not the "Not executed" string. The return value from the `catch` function will be the "There is trouble!" string. After the exception handled, the execution continues after the `catch` and there is no simple way to implement the `retry` from Eiffel.[41]

If an exception is thrown and is not caught in the function, the exception is rethrown in the caller function, the exception is propagated. There is no

[41] See Section 8.3.4.

specialized exception handler in the language, the exception can be handled in the function calls following the *catch* function. It's not easy to decide whether there was an exception at all, because we can only rely on the return value of the *catch* function as a source of help:

```
(setq exception_happened
     (catch 'excexc
       (if (equal i 1)
          (throw 'exc "i=1")
        nil)
       (eval "OK")))
(if (equal exception_happened "OK")
   (print "All is well")
  (progn
    (print "Exception happened, with cause:")
    (print exception_happened)))
```

If there is no exception in the *catch* function, its return value will be the return value of the last function (in the above example it is "OK"). It is unfortunate, because we have to introduce an unnecessary – to meat the requirement – statement due to technical reasons.

8.3.10 Exceptions in concurrent environment: Erlang

The development of Erlang [CT09] was started in the late 1980s. Its background lies in logical and functional languages and was developed to build highly reliable, fault tolerant, soft realtime and concurrent systems. In this section, not only the exception handling, but the general error handling of the language is presented.

Due to the highly concurrent nature of many Erlang programs, the basic building block of a running Erlang system is usually a process, not an object (like in object-oriented languages).[42] In object-oriented programming languages, the tasks are usually executed by objects calling each other's methods; in Erlang it is usually executed by processes sending messages to each other. Error handling is also connected to processes: processes can be linked to each other or can monitor each other. Another basic design feature of Erlang is to "Let it crash!" meaning if an execution fails, the executing process should crash and be restarted.

This design pattern is implemented using process links or monitors. If two processes are linked to each other, and one of them terminates (finishes the execution or crashes, for example), an exit signal is delivered to the other process (a process can be linked to more than one other process). By default if the process terminates with a reason other than *normal* (i.e. due to an error), the exit signal terminates the other process too. However, this later process (usually called supervisor) can choose to trap the exit signals (by setting the **trap_exit** process

[42] See Chapter 10.

flag) and handle the situation, e.g. by restarting the exited process. Links are always bidirectional. If it is not sufficient, monitors can be used. A process can monitor other processes - if the monitored process terminates, the monitoring process receives a message about the event and can act upon it. The monitor is unidirectional. This design pattern is very useful e.g. for implementing network servers: when one process(group) serving one client crashes, only those processes crash, the rest of the server can continue to serve the other clients.

If we treat this inter-process error handling as exception handling, the exception is the content of the message sent when a process terminates (it is a specially formatted tuple).[43] This tuple contains the PID (process identifier) of the crashed process and the reason why it terminates, which is an Erlang term. These exceptions might be grouped by the exit reason; however in practice this is not useful. This exception handling is obviously connected to processes. Unlike in previous cases, exceptions are propagated between processes, since the exception is a process termination. The terminated process can be restarted, but this is fundamentally different from the *retry* in Eiffel,[44] because the restarted process will have different state.

Since having an exception implies process termination, there is no way to have a *finally* block. However, the standard library of Erlang provides *behaviors*: modules (in practice, processes) implementing the *gen_server*[45] or *gen_fsm*[46] behavior have to implement a *terminate* function. This is called when the implementing process terminates, and this is the place to free the allocated resources. This is not a language element, but a function of the standard library.

In addition, Erlang has an in-process (single-threaded) exception handling. Exceptions in Erlang are runtime errors (e.g. division by zero) or generated errors (by the **exit** or **throw** functions). The exceptions are classified based on their origin, so the runtime errors are in the **error** class, the exceptions generated by **exit** are in the **exit** class and the exceptions generated by **throw** are in the **throw** class. Both the **throw** and the **exit** functions can have any terms as parameters. Thus it is possible (but rarely used) to create further grouping of exceptions. However, the exceptions can be trivially parametrized this way. The exceptions are propagated through the call chain and if not handled, eventually they terminate the process. An interesting side-effect is that the **exit** function might not exit the current process, if the thrown **exit** class exception is handled.

Exceptions can be thrown from expressions, and the language does not provide ways to specify them. However, in library functions it is often documented that an exception other than a runtime error can be thrown.

The language provides two constructs to handle these exceptions. Originally only the **catch** expression could be used to catch exceptions and it had to be used before the exception was thrown (example output from the Erlang interpreter):

[43] See 15.3.1.
[44] See Section 8.3.4.
[45] The server part of a generic client-server relation.
[46] A generic finite state machine process.

```
1 > catch 1/0.
{'EXIT',{badarith,...
2> catch throw(x).
x
3> catch exit(problem).
{'EXIT',problem}
```

Erlang 5.4 introduced the **try** expression which is an exception handling more familiar to programmers with a background in C++ or Java:

```
try <expression> [of
  <pattern1> -> <body1>;
  ...]
catch
  [<class>:]<exception-pattern1> -> <exceptionbody1>;
  ...
[after
  <afterbody>]
end.
```

Exceptions thrown in the <*expression*> are caught. If there is no exception thrown and the optional **of** section is specified, – based on the value of the <*expression*> – the body with the matching pattern is executed (i.e. the pattern which matches the value of the expression). Exceptions thrown from these bodies are not caught by this **try**. Also if none of the patterns match, a **try_clause** exception is thrown which is not caught by this **try**. If there was an exception thrown during evaluating the <*expression*>, the body of the matching exception pattern is executed.[47] If no patterns match, the exception is propagated outside the **try** expression. Finally, if the optional **after** section is present, it is executed after all bodies are executed, this is Erlang's solution for the *finally*. Example:

```
f(X) ->
    try g(X) of
        5 = Return ->
            io:format("Normal return: ~p~n", [Return]),
            Return
    catch
        exit:1=Exit ->
            io:format("Exit ~p caught~n", [Exit]);
        error:Error ->
            io:format("Error ~p caught~n", [Error]);
        throw:Throw ->
            io:format("Throw ~p caught~n", [Throw])
```

[47] The exception class is also part of the pattern, so it is possible to handle different classes with the same body. If no class is specified, **throw** is used.

```
    after
        io:format("Always executed~n")
    end.

g(1) ->
        exit(1);
g(2) ->
        2/0; % a runtime error
g(3) ->
        throw(3);
g(4) ->
        exit(4);
g(X) ->
        X.
```

- If f is called with 1, the exit is caught.
- If f is called with 2, the runtime error is caught.
- If f is called with 3, the thrown integer is caught.
- If f is called with 4, the exit is not caught, so the process terminates (unless f is called inside a **catch** or **try**).
- If f is called with 5, no exception is thrown and the function returns 5.
- If f is called with 6, a **try_clause** exception is thrown and not caught.

Unlike in C++ or Java, if there is an **of** section, not all exceptions thrown from between the **try** and **catch** keyword are caught by this construct, which might be confusing at first. It also shows that this **try** construct was invented to replace the often used **case catch** ... expressions:

```
f(X) ->
    case catch g(X) of
        5 = Return ->
                io:format("Normal return: ~p~n", [Return]),
                Return;
        {'EXIT',ExitOrError} ->
            io:format("Exit or error ~p caught~n", [ExitOrError])
    end.
```

This construct did not differentiate between normal return values and values returned from **throw**,[48] it was very hard to check the difference between exits and runtime errors, and of course, there was no simple way to implement *finally*. It is very similar to the exception handling of Lisp.[49]

[48] Actually in the language specification **throw** is defined as non-local returns, i.e. more similar to the *return* of C++ or Java than to the **throw** of the same languages - a false friend in computer languages.

[49] See Section 8.3.9.

When an exception is not handled in the process, the process terminates. If the process was linked to other processes or was monitored by other processes, they get signals or messages about the fate of this process. This way the single-process exception handling is connected to the inter-process exception handling.

8.3.11 New solutions: Perl

Perl [SP01] originates from the late 1980s. Although it does not have a specialized exception handler, using the *eval* and *die* operators the programmers could implement an exception handling that is similar to Lisp. The `Error` library (`package` in Perl terminology) is using these operators to implement an exception handling similar to that of C++[50] and Java.[51] Although this solution is not strictly a language element, it has some unique ideas. In this section the `Error` package ([Bar01] and [Sha02]) will be presented.

The classes of exception objects have to inherit from the `Error` class. Since Perl supports multiple inheritance,[52] this is not a strong drawback. The `Error` class is part of the `Error` package too. As these exceptions are objects, they can be organized by class hierarchy. The `throw` function can be used to throw the exception. Exceptions can be parameterized by setting the fields of the objects. Exceptions are not part of the language, hence the thrown exceptions cannot be specified in function headers.

Exception handling is connected to blocks. Exception handling is implemented by the `try` class-level function of the `Error` class and this function is parameterized by blocks and clauses. For simplicity's sake, we will name the parameter of the `try` block as `try` block, the block after `catch` as `catch` block, etc., even though they are not special blocks in the language.

```
use Error qw(:try);

@Error::Own::ISA = (Error);

sub handler {
  my $e = shift;
  my $do_continue = shift;
  print "I caught the exception again: $e"; #$
  $$do_continue=1;
}
try {
  print "Let's thrown an exception!";
  throw Error::Simple("Trouble!");
}
```

[50] See Section 8.3.3.
[51] See Section 8.3.6
[52] See Section 10.7.5.

```
catch Error::Simple with {
  my $e = shift;
  my $do_continue = shift;
  print "I caught an exception: $e ";
  $$do_continue=1;
}
except {
  return {
    Error::Simple => \&handler%TAMOP% Error::Simple => &handler
  };
}
otherwise {
  my $e = shift;
  print "I did not think about this: $e ";
}
finally {
  print "Always executes.";
};
```

Only exceptions thrown in the `try` block can be caught. Exceptions can be handled in several ways. In the `catch` block, we can catch instances of one class, while in the `except` block the programmer can select an exception handler during runtime.[53] The first parameter of the `catch` function (and of the handler function specified in `expect`) is the exception object. The `otherwise` block is executed when no `catch` or `expect` block handles the exceptions (the same as the `catch(...)` in C++). The first parameter of this function is also the exception object. The `finally` block always executes, as in Java. None of the clauses are mandatory, but only the `catch` and `except` blocks can be specified multiple times. The order of the `catch` clauses are important: if more than one handler can catch an exception, only the first one will execute.

After the exception has been handled, the execution continues after the `try` block, unless the second parameter of `catch` block or the handler function in `expect`[54] is set to a value. In this case, the execution continues as the exception was not handled at all, by trying the next `catch` or function in `expect`. This means that the exception is not propagated to higher level, but on the same level. This feature is not provided by any of the languages presented above.

If an exception is not handled, it is propagated and can be handled by outer `try` blocks. If it is not handled, the program halts, unless the exception is thrown in an *`eval`* operator. This is a side effect of exception handling being implemented by *`die`* and *`eval`* operators. Of course, the exception handler can throw an exception and the exception can be rethrown by its `throw` method.

[53] A hash-reference with keys of exception classes and values of exception handler function references can be returned from the `except` block.

[54] In the above example it is the `$do_continue` variable.

8.3.12 Back to the basics: Go

The development of the Go programming language [Tea09] began in 2007 at Google for systems development with a great focus on concurrency, safety and performance (both compiling and executing). The language actually does not have general exception handling like Java[55] or C++,[56] but contains a more limited approach focusing on error handling. In Go, the functions can have multiple return values which avoid the problems mentioned with the `atoi` function on page 373.Go does not have constructors either. Nevertheless, runtime errors may happen, and thus the language provides features to handle these errors.

With the `defer` statement the programmer can set up a function that will be always called when the function returns on any branch. This has the exact same purpose as the `finally` in e.g. Java: a code that ”cleans up” properly.

The `panic` function can be used to signal runtime error. This is similar to throwing exceptions in other languages: the currently executing function returns, and from the caller's perspective the function is the same as the a call to `panic` (i.e. the panic situation propagates). The `panic` function has one string parameter. The ”panic” can be handled by calling the `recover` function: when recover is called, the function returns normally. If the ”panic” is not handled, this string is printed when the goroutine (thread in other languages) terminates. If a goroutine terminates due to a panic, the whole program terminates, and thus the ”exception” does not propagate to other threads.

```
func f() {
  fmt.Println("f called")
  panic("Let's panic!")
}
func g() {
  fmt.Println("g called")
  f()
  fmt.Println("g finished")
}
func main() {
  defer func() {
    fmt.Println("Panic handled")
    cause := recover()
    fmt.Println("Cause: ", cause)
  }()
  fmt.Println("calling g")
  g()
  fmt.Println("g returned")
}
```

[55] See Section 8.3.6.
[56] See Section 8.3.3.

The above example prints this output:

```
calling g
g called
f called
Panic handled
Cause:  Let's panic!
```

Compare this with a similar Java code:

```
void f() throws PanicException {
  System.out.println("f called");
  throw new PanicException("Let's panic!");
}

void g() throws PanicException {
  System.out.println("g called");
  f();
  System.out.println("g finished");
}

public void main() {
  try {
    System.out.println("calling g");
    g();
    System.out.println("g returned");
  }
  catch (PanicException e) {
    System.out.println("Panic handled");
    System.out.println("Cause: "+e.getMessage());
  }
  // There is no equivalent in Go for code here!
}
```

The big difference is that although the *g* method does not do anything with the *PanicException*, it still has to specify in its definition in Java.

The above shown error handling is connected to subprograms, not to blocks as in e.g. C++. The execution continues after the function call which handled the exception (panic).[57]

One of the problems of the Go authors with exception handling is that it encourages programmers to treat too many ordinary errors as exceptions. This error handling is designed to handle runtime errors, not plain errors (for which the multiple return values can be used). This is why the *defer* statement works more like **finally** than **catch**.

[57] in the above example the execution would continue in the function that called *main*

8.4 Summary

Exception handling is a very useful programming tool, no wonder that the newer languages all provide an implementation. The biggest advantage is that the actual solution of the task is separated from the error handling. Thus it is easier to understand and modify the program, and it does not cause a problem if the error handling code is far (a couple of levels higher in the call chain) from the originating place of the error. Another great advantage is that in object-oriented languages the constructors cannot return an error code due to their nature; rather by throwing an exception the caller does not only get notified that there is an error, but also about the reason of the failure (it would be possible to use an *out* parameter – see section 7.4. –, but that's not too elegant).

Exception handling also has some drawbacks: it slightly spoils the structure of the program, because in a way it works like the **goto** statement. Moreover, in some languages (e.g. C++) it can be downright dangerous to throw an exception when it should not be done and the compiler does not warn. Also it must be borne in mind that exception handling comes with price: the compilers of some languages (e.g. C++) have to generate additional code to handle exceptions which would be unnecessary if we would chosen a different error handling mechanism.[58] The checked exception idea, which seemed to be useful when Java was designed[59] turned out to be less than useful in big software. Some language designers think that exception handling should be used only in really exceptional situations (runtime errors), not for general error handling.

How the standard libraries of the languages use exceptions indicates how desirable are exceptions from the language designers' perspective. Java introduces rich set of exceptions that are organized in a strict hierarchy and the programmer has to use them, but it's fairly simple. By contrast, the standard libraries of C++ does not use exceptions thoroughly, partly due to the many used legacy C functions, which use return codes anyway. Ada only contains 4 predefined exceptions, while in Delphi a separate **unit** has to be used to enable exception handling. In Eiffel the exceptions are mainly due to violated constraints which are often used (e.g. Java's *IndexOutOfBoundsException* would be the violation of the precondition of the indexing method in Eiffel). In Erlang many of the standard applications are using a supervisor hierarchy of linked processes to achieve robustness.

Of all the languages described above Java and C# are may have the most elaborate exception handling, which is not surprising, given that these languages are fairly new, and their designer could look at other languages for ideas. For this reason, new programming languages likely to contain fresh ideas and solutions, as we could see in the Perl module.[60]

[58] Of course, in this case we have to write that different error handling mechanism.
[59] See Section 8.3.6.
[60] See Section 8.3.11.

8.5 Examples for exception handling

8.5.1 C++

The example below implements a stack with two elements. The stack has two operations, *push* puts a new element on top of the stack, while *pop* takes the top element from the stack. Both methods throw an exception on error (pushing to a full stack or popping from an empty stack). The classes of both exceptions are children of the *StackException* class, as an example for grouping exceptions.

```
#include <iostream>
#include <string>

using namespace std;
class StackException {
  protected:
    string error;
  public:
    StackException(string _error): error(_error) {};
    string toString(void) { return error; };
};
class FullStackException: public StackException {
  public:
    FullStackException(void): StackException("Stack is full!") {};
};
class EmptyStackException: public StackException {
  public:
    EmptyStackException(void): StackException("Stack is empty!"){};
};
class Stack {
  int st[2], sp;
  public:
    Stack(void) { sp=0; };
    void push(int i) throw (FullStackException) {
      if (sp >= 2) throw FullStackException();
      st[sp] = i;
      sp++;
    };
    int pop(void) throw (EmptyStackException) {
      if (sp <= 0) throw EmptyStackException();
      sp--;
      return st[sp];
    };
};
```

```
int main(void) {
  Stack st;
  try {
    st.push(1); st.push(2);
    st.push(3); // FullStackException is thrown
  } catch(const StackException& se) {
    cout << "The following problem happened:" << se.toString() << endl;
  };
  try {
    st.pop(); st.pop();
    st.pop();   // EmptyStackException is thrown
  } catch(const StackException& se) {
    cout << "The following problem happened:" << se.toString() << endl;
  };
  return 0;
}
```

8.5.2 Java

The following example implements the same task as the example in Section 8.5.1.: a stack with two elements. The stack has two operations, *push* puts a new element on top of the stack, while *pop* takes the top element from the stack. Both methods throw an exception on error (pushing to a full stack or popping from an empty stack). The classes of both exceptions are children of the *StackException* class, as an example for grouping exceptions. Note that in the C++ example the program itself has to notice that there is an error (there is an **if** clause at the start of the methods), whereas in Java, JVM[61] notices that the state of the stack is wrong and throws an exception.

```
class StackException extends Exception {
  protected String error;
  public StackException(String _error) { error=_error; }
  public String toString() { return error; }
}

class FullStackException extends StackException {
  public FullStackException() { super("Stack is full!"); }
}
class EmptyStackException extends StackException {
  public EmptyStackException() { super("Stack is empty!"); }
}
```

[61] Java Virtual machine: the program that executes the Java program.

```
class Stack {
  int[] st, sp;
  public Stack() {
    st=new int[2];
    sp=0;
  }
  public void push(int i) throws FullStackException {
    try {
      sp++;
      st[sp]=i;
    } catch (ArrayIndexOutOfBoundsException e) {
      sp-;
      throw new FullStackException();
    }
  }
  public int pop() throws EmptyStackException {
    try {
      sp-;
      return st[sp];
    } catch (ArrayIndexOutOfBoundsException e) {
      sp++;
      throw new EmptyStackException();
    }
  }

  public static void main(String args[]) {
    Stack st=new Stack();
    try {
      st.push(1);
      st.push(2);
      st.push(3); // FullStackException is thrown
    } catch (StackException se) {
      System.out.println("The following problem happened: "+se);
    }
    try {
      st.pop();
      st.pop();
      st.pop(); // EmptyStackException is thrown
    } catch (StackException se) {
      System.out.println("The following problem happened: "+se);
    }
  }
}
```

8.5.3 Ada

The following example implements the same task as the example in section 8.5.1.: a stack with two elements. The stack has two operations, *push* puts a new element on top of the stack, while *pop* takes the top element from the stack. Both methods throw an exception on error (pushing to a full stack or popping from an empty stack). The classes of both exceptions are children of the *StackException* class, as an example for grouping exceptions. Notice that similarly to the example in Section 8.5.2., the runtime environment finds the errors.

```
package STACK_E is
    type Stack is limited private;
    procedure Push(st: in out Stack; i: INTEGER);
    procedure Pop(st: in out Stack; i: out INTEGER);
    FullStackException: exception;
    EmptyStackException: exception;
private
    type Data is array(0..1) of INTEGER;
    type Stack is record
        sp: NATURAL := 0;
        st: Data;
    end record;
end STACK_E;

with Ada.Exceptions; use Ada.Exceptions;

package body STACK_E is
    procedure Push(st: in out Stack; i: INTEGER) is
    begin
        st.st(st.sp) := i;
        st.sp := st.sp + 1;
    exception
        when CONSTRAINT_ERROR => Raise_Exception
            (FullStackException'Identity, "Stack is full!");
    end Push;

    procedure Pop(st: in out Stack; i: out INTEGER) is
    begin
        st.sp := st.sp - 1;
        i := st.st(st.sp);
    exception
        when CONSTRAINT_ERROR => Raise_Exception
            (EmptyStackException'Identity, "Stack is empty!");
    end Pop;
end STACK_E;
```

```
with STACK_E,TEXT_IO, Ada.Exceptions;
use STACK_E,TEXT_IO, Ada.Exceptions;

procedure Stack_Example is
    st: Stack;
    i: INTEGER;
begin
    begin
        Push(st, 1); Push(st, 2);
        Push(st, 3);        -- FullStackException is thrown
    exception
        when e: FullStackException =>
            Put("The following problem happened:");
            Put_Line(Exception_Message(e));
    end;
    begin
        Pop(st, i);  Pop(st, i);
        Pop(st, i);         -- EmptyStackException is thrown
    exception
        when e: FullStackException | EmptyStackException =>
            Put("The following problem happened:");
            Put_Line(Exception_Message(e));
    end;
end Stack_Example;
```

8.5.4 Eiffel

The following example implements the same task as the example in Section 8.5.1.: a stack with two elements. The stack has two operations, *push* puts a new element on top of the stack, while *pop* takes the top element from the stack. In both methods the precondition checks for the errors.

```
class
        STACK_EXAMPLE
create
        make
feature -- Initialization
  st: ARRAY[INTEGER];
  sp: INTEGER;
  make is
    do
      !!st.make(0, 1);
      sp:=0;
    end;
```

```
    push(i: integer) is
      require
        StackNotFull: sp < 2
      do
        st.put(i, sp);
        sp := sp + 1;
      end;
    pop: INTEGER is
      require
        StackNotEmpty: sp > 0
      do
        sp := sp − 1;
        Result := st.item(sp);
      end;
end

class
  EXAMPLE
create
  make
feature −− Initialization
  make is
    local
      st: STACK_EXAMPLE;
      i: INTEGER;
      was_exception:BOOLEAN;
    do
      !!st.make;
      if not was_exception then
        st.push(1); st.push(2);
        io.put_string("OK");
        st.push(3); −− StackNotFull precondition is not satisfied here
        io.put_string("NOK");
      else
        i := st.pop; i := st.pop;
        i := st.pop;  −− StackNotEmpty precondition is not satisfied here
      end;
    rescue
      if not was_exception then
        was_exception := true;
        retry;
      end;
    end;
end
```

8.5.5 Erlang

The following example demonstrates the supervisor pattern in Erlang. The code below starts two processes, a *responder* process which prints ”pong” for incoming *ping* messages, and a supervisor process which restarts the *responder* in case it crashes. There is intentionally an error in the *responder* process: when it receives an unexpected message, the process terminates (the recursive call is missing).

```
-module(responder).

-export([start/0]).

responder() ->
  receive
    ping ->
      io:format("pong\n"),
      responder();
    _Unknown ->
      skip
  end.

do_supervise(Responder) ->
  register(responder, Responder),
  M = erlang:monitor(process, Responder),
  sup_loop(M, Responder).

sup_loop(M, Responder) ->
  receive
    {'DOWN', M, process, Responder, Reason} ->
      io:format("Responder stopped with reason ’~p’\n", [Reason]),
      NewResponder = spawn(fun responder/0),
      do_supervise(NewResponder);
    _ ->
      sup_loop(M, Responder)
  end.

start() ->
  Responder = spawn(fun responder/0),
  spawn(fun() -> do_supervise(Responder) end).
```

In order to make testing easier, the supervisor also registers the *responder* process into the process registry.[62] Example testing output from the Erlang shell:

[62] By registering a process into the process registry, we can send messages to the process by using its registered name instead of its PID. When the process is restarted, the PID is

```
3> responder:start().
<0.44.0>
```

The processes are started (the returned pid is actually the pid of the supervisor).

```
4> responder ! ping.
pong
ping
5> responder ! ping.
pong
ping
```

pong is printed for each *ping* (the *ping* printout is the evaluated value of the sent message).

```
6> responder ! crash.
Responder stopped with reason 'normal'
crash
7> responder ! ping.
pong
ping
```

A "wrong" message is sent to the responder, it terminated, but the supervisor restarted it, as the next message is handled.

```
13> exit(whereis(responder), kill).
Responder crashed with reason 'killed'
true
14> responder ! ping, ok.
pong
ok
```

The responder process is stopped with an untrappable exit signal; yet, it handles the next message. This example shows the resiliency achievable using the supervisor pattern. The *supervisor* module and behaviour in the Erlang standard library are more complicated with more features.

8.6 Excercises

Exercise 8.1. List some programming languages where exceptions are not objects!

Exercise 8.2. Why is it a good idea for the specification of a subprogram to contain the kind of exceptions (types, classes, etc.) that can be thrown by the subprogram?

changed, but since we register the same name, we can use the same name.

Exercise 8.3. What is the purpose of the *finally* construct in Java?

Exercise 8.4. Complete the following C++ program:

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <cerrno>

using namespace std;

class NotNumber {
    string wrongInput;
    friend ostream& operator<<(ostream& os,
                               const NotNumber& nn);
public:
    NotNumber(const string& s) : wrongInput(s) { };
};
ostream& operator<<(ostream& os, const NotNumber& nn) {
    return os << "Invalid input: " << nn.wrongInput;
}
long number(const string& s) throw (NotNumber) {
    ⟨missing code⟩
}
int main(void) {
    try {
        cout << number("12") << endl;
        cout << number("12345678901234567890") << endl;
    } catch (const NotNumber& ns) {
        cout << "Error happened: " << ns << endl;
    }
    try {
        cout << number("-34") << endl;
        cout << number("text") << endl;
    } catch (const NotNumber& ns) {
        cout << "Error happened: " << ns << endl;
    }
    return 0;
}
```

The *number* function has to be completed. The task of this function is to create a number (of **long** type) from the parameter string. If the parameter cannot be parsed as a decimal number, or does not fit the **long** type, *NotNumber* exception should be thrown. Use the *strtol* function from the standard library! The result should be the following:

```
12
Error happened: Invalid input: 12345678901234567890
-34
```

```
Error happened: Invalid input: text
```

Exercise 8.5. Complete the following Java program:

```
class WrongDateException extends Exception {
    String invalidInput;

    public WrongDateException(String s) { invalidInput = s; }

    public String toString() {
        return "Invalid input: " + invalidInput;
    }
}

public class MorningDate {
    public static String morningOrAfternoon(String date)
                                        throws WrongDateException {
        ⟨missing code⟩
    }

    public static void main(String[] args) {
        try {
            System.out.println(
                MorningDate.morningOrAfternoon("11:53"));
            System.out.println(
                MorningDate.morningOrAfternoon("11:wrong"));
        } catch(WrongDateException e) {
            System.out.println("Error happened: " + e);
        }
        try {
            System.out.println(
                MorningDate.morningOrAfternoon("13:42"));
            System.out.println(
                MorningDate.morningOrAfternoon("text"));
        } catch(WrongDateException e) {
            System.out.println("Error happened: " + e);
        }
    }
}
```

The *morningOrAfternoon* method should decide if the input refers to a morning or an afternoon hour (i.e. before 12:00 or after) and should throw an exception if the input is not a time. The expected output:

```
11:53 is in the morning
Error happened: Invalid input: 11:wrong
13:42 is in the afternoon
Error happened: Invalid input: text
```

For this exercise it is not necessary to check if the ”minutes” are between 0 and 59.

Exercise 8.6. Complete the following Eiffel program:

```
class
  MORNING

create
  make

feature
    is_morning(time: STRING): BOOLEAN is
      require
          time_is_ok: format_ok(time);
    do
      ⟨missing code⟩
    end

feature
    make is
      local
          second_run: BOOLEAN;
      do
          io.put_boolean(is_morning("11:23"));
          io.put_string("%N");
          io.put_boolean(is_morning("23:11"));
          io.put_string("%N");
          if not second_run then
              io.put_boolean(is_morning("szoveg"));
              io.put_string("%N");
          end;
      rescue
          second_run := true;
          retry;
    end;
end
```

The *is_morning* method should decide if the input is a morning or an afternoon date (i.e. before 12:00 or after). The precondition is that the parameter is a time. The *format_ok* method should take care of it - it part of the exercise to implement it. For this exercise it is not necessary to check that the ”minutes” are between 0 and 59.

8.7 Useful tips

Tip 8.1. See Sections 8.2.3, 8.3.1, 8.3.2, 8.3.4, 8.3.5, 8.3.9 and 8.3.10.

Tip 8.2. See Section 8.1.1 on page 368.

Tip 8.3. See Section 8.1.1 on page 369.

Tip 8.4. The *strtol* function uses two ways to show error: the *errno* variable is set to *ERANGE* (if the parsed number does not fit the **long** type), or the pointer in the second parameter does point to something other than '\0' (if the entire input string was not a number). Do check for both conditions. Do not forget to reset the *errno* variable before using its value!

Tip 8.5. Split the input string on `":"`, then check that the input string was split into exactly two parts. Check that both parts are integers. Check that the first part (the "hour") is between 0 and 11 or 12 and 23 (inclusive).

Tip 8.6. In the *format_ok* method split the input string on *':'*, then check that the input string was split into exactly two parts. Check that both parts are integers. Check that the first part (the "hour") is between 0 and 11 or 12 and 23 (inclusive). In the *is_morning* only check that the "hours" value is before 12.

8.8 Solutions

Solution 8.1. PL/I, CLU, Ada95, Eiffel, Modula-3, Common Lisp, Erlang

Solution 8.2. If the specification for a subprogram contains the list of exceptions that can be thrown by the subprogram, it warns the user of the subprogram that what kind of exceptions can be thrown and what should be handled. In some languages this specification actually forces the programmer to handle these exceptions (the "checked exceptions" concept).

Solution 8.3. The **finally** block is always executed after the **try** block, regardless of exceptions thrown. It is useful to avoid resource leaks: even though the garbage collector frees allocated memory, it might be necessary to e.g. always close a window in a function, whether there was an exception or not.

Solution 8.4.

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <cerrno>

using namespace std;

class NotNumber {
    string wrongInput;
    friend ostream& operator<<(ostream& os, const NotNumber& nn);
public:
```

```
    NotNumber(const string& s) : wrongInput(s) { };
};

ostream& operator<<(ostream& os, const NotNumber& nn) {
    return os << "Invalid input: " << nn.wrongInput;
}

long number(const string& s) throw (NotNumber) {
    long result;
    char* endp;
    errno = 0;

    result=strtol(s.c_str(), &endp, 10);
    if ((*endp != '\0') || (ERANGE == errno)) throw NotNumber(s);
    return result;
}

int main(void) {
    try {
        cout << number("12") << endl;
        cout << number("12345678901234567890") << endl;
    } catch (const NotNumber& ns) { cout << "Error happened: " << ns << endl; }
    try {
        cout << number("-34") << endl;
        cout << number("text") << endl;
    }
    catch (const NotNumber& ns) { cout << "Error happened: " << ns << endl; }
    return 0;
}
```

Solution 8.5.

```
class WrongDateException extends Exception {
    String invalidInput;
    public WrongDateException(String s) {
        invalidInput = s;
    }
    public String toString() {
        return "Invalid input: " + invalidInput;
    }
}

public class MorningDate {
    public static String morningOrAfternoon(String date)
            throws WrongDateException {
        String[] elements = date.split(":");
        if (elements.length != 2) throw new WrongDateException(date);
        try {
            Integer.parseInt(elements[1]);
            int hours=Integer.parseInt(elements[0]);
            if (0 <= hours && hours <= 11) return date+" is in the morning";
            else if (13 <= hours && hours <= 23) return date+
                " is in the afternoon";
            else throw new WrongDateException(date);
        } catch (NumberFormatException e) {
            throw new WrongDateException(date);
        }
    }

    public static void main(String[] args) {
        try {
            System.out.println(MorningDate.morningOrAfternoon("11:53"));
            System.out.println(MorningDate.morningOrAfternoon("11:wrong"));
        } catch(WrongDateException e) {
            System.out.println("Error happened: " + e);
        }
        try {
            System.out.println(MorningDate.morningOrAfternoon("13:42"));
```

```
            System.out.println(MorningDate.morningOrAfternoon("text"));
        } catch(WrongDateException e) {
            System.out.println("Error happened: " + e);
        }
    }
}
```

Solution 8.6.

```
note
    description : "morning application root class"
    date        : "$Date$"
    revision    : "$Revision$"

class
    MORNING

create
    make

feature
    is_morning(time: STRING): BOOLEAN

        require
            time_is_ok: format_ok(time);
        local
            dates: LIST[STRING]
            hours : INTEGER;
        do
            dates := time.split(':');
            hours := dates.at(1).to_integer;
            Result := (hours < 12)
        end;

feature
    format_ok(time: STRING): BOOLEAN
        local
            dates: LIST[STRING];
            hours : INTEGER;
        do
            dates := time.split(':');
            if dates.count = 2 and dates.at(1).is_integer()
                    and dates.at(2).is_integer() then
                hours := dates.at(1).to_integer;
                Result := (0 <= hours) and (hours <= 23)
            else
                Result := false
            end
        end

feature
    make
      local
          second_run: BOOLEAN;
      do
          io.put_boolean(is_morning("11:23"));
          io.put_string("%N");
          io.put_boolean(is_morning("23:11"));
          io.put_string("%N");
          if not second_run then
              io.put_boolean(is_morning("szoveg"));
              io.put_string("%N");
          end;
      rescue
          second_run := true;
          retry;
    end;
end
```

9 Abstract data types

The method of data abstraction *and the abstract data types that follow from it play an important role in modern programming methodology.*[1] *This is why it is crucial to examine what requirements the type abstraction defines against the language, and how these requirements are met. In this chapter we will examine through what features the various programming languages support data abstraction. These features include subprograms, expressions, type constructs, and many other features that are discussed in detail in other chapters of the present book. Thus this chapter will contain a lot of referencing to other parts of the book.*

[1] See [Jac75], [Dij76] and [DDH72]

Every task can be solved in FORTRAN. If something cannot be solved in FORTRAN, it can be solved in Assembly. If something cannot be solved in Assembly, it cannot be solved at all. – The real programmer [Pos83]

This quote is funny, but true. Why are then other programming languages needed? The answer is quite obvious – because most of the tasks are easier to solve in other languages. This has two reasons. Firstly, there are languages designed for solving particular tasks such as the SQL language for querying relational databases. Second reason is the development of programming methodology. Of course, by disciplined programming nearly all programming paradigms can be implemented in any general purpose programming language. However, without language feature support these solutions usually require many mechanical operations on the side of the programmer, and as we know, mechanical operations are the primary sources of errors. *Thus, to avoid this from happening, we need programming languages which actively support the applied programming methodologies.*

9.1 Type constructs and data abstraction

Most programming languages offer a variety of features for the programmer to construct new types from existing ones. Using these type construct features only allow to produce types with fixed semantics. Chapter 6. examines in detail which construct generates what type with what kind of properties from the compound types. These ”pre-constructed” semantics, however, rarely meet all the demands. The objects in the real world to be modeled in the programming language, are rarely Cartesian products, vectors or sets.

What are these constructs then good for? They can help to model real world objects. People are no Cartesian products, but making, for example, a telephone book application, the *relevant* attributes of people *for this application* can be properly described by a Cartesian product. That is, in practice, type constructs are not used on their own, but for representing important new types.

In Section 5.1. we have seen the formal definition of the type. According to that definition, the representation of the type always happens by a sequence of elemental values which – if examining the structure of the program deep enough – is true. In practice, for type representation we use existing types, or new types created with the type construction features of a given language. From the existing types the desired new type is created in two steps. Firstly, an intermediate type, with any of the type construction features, is defined. This type has the desired representation structure, but its semantics – i.e. the operations – differ from that of the desired type. Secondly, using the operations of the type construction and of the original type, the type operations of the new type are implemented.

As an example, let us consider the rational number type. For its representation we may choose a Cartesian product with two integers. We use the Cartesian product type construction of the chosen language, but the result type is not the rational number type yet, just a new type which contains two integers. Likewise, the operations are also not the usual operations of rational numbers, such as addition, subtraction, multiplication and division, but the querying and setting of the individual components only. Nevertheless, with its operations and integer members, this new type is suitable for what the formerly available integer type was not. That is, it supports the operations of the rational numbers.

9.2 Expectations for programming languages

As mentioned in the introduction, it is a legitimate expectation of any programmer that the given programming language actively supports the given program design methodology. Next, we will list the areas where language support is expected to make the job of the programmer more efficient and simple. Our list will concern the following specific language features and syntactic limitations:

- *Modularity* for each type to be implemented in separate compilation units. This ensures the reusability of types, and also efficient program developing, since single modules can be easily transferred to other programs, and different units can be developed by different programmers without them disturbing each others' work.
- *Encapsulation* for handling the related type value set and its operations together.
- *Representation hiding*, which ensures that the user of the given type only uses the operations given by the specification. This limitation enables the modification of the representation and also of the implementation without propagating it up in the program hierarchy.
- *Separation of the specification and the implementation* into separate compilation units. The other modules using the given type can simultaneously be developed with a type specification independent of the actual implementation.

- *Managing of module dependencies.* The compiler should manage the dependencies between modules (one using the other etc.) automatically.
- *Consistent usability.* The user and built-in types are not to be handled differently. Furthermore, types should be manageable in ways that are close to ”real” life methods.
- *Support of generalized program templates.* The programmer should be able to write programs in the most general way. Languages should give the possibility to minimize repetitions, not only on the direct code level, but also on a higher level to avoid the multiple implementations of the same solution structures. This significantly increases the readability and maintainability of the code.

9.3 Breaking down to modules

The complexity of any problem may reach a point, where the size of solving the problem grows beyond control. To respond to the increase in problem complexity, an isolated monolithic approach is no longer a viable solution; instead, we need a software system with a more independent modular design of many subprograms, and with good interconnections. These design units are called modules, and this design approach is called modular design.

In programming languages meant for professional use support modularity. In more modern languages, the basis for breaking down to modules is shifting towards breaking down to types; that is, a module is implementing a type. As will be shown below, modularity has an important role in representation hiding. As in most of the languages the boundaries of the modules and of the various visibility categories are usually the same, modules regulate the visibility of identifiers defined within them. In the following section, we will examine the features of modularity support in great detail.

9.3.1 Modular design

Applying modularity is also helpful in meeting some of the most typical requirements of object-oriented design, e.g. reusability, extensibility and compatibility. This is what the so called modular design is based on. (The module is not a programming language concept, but a unit of design.)

Criteria of modular design

Next, we will consider the main aspects of modular design.

Modular decomposition

Modular decomposition means the breakdown of the problem to more simple tasks which can be solved independently. This is how the complexity of the problem can be decreased.

Usually this method is applied repeatedly, decompositioning also the subtasks. This allows more people to work simultaneously on solving the main problem. The method can be presented as a tree where the nodes represent each decomposition step.

For example, the top-down method of program design is based on this method too. Designers start from the abstract description of the problem, and apply decomposition steps until they reach the level of description which may be implemented on the given programming language.

Modular composition

Modular composition means the application of already existing program units as building blocks to create new programs. A software design method satisfies this criterion if it supports the implementation of software elements which may freely combine. Modular decomposition supports extensibility, while modular composition assists reusability.

The declared goal is to reuse already existing standard elements during programming. For example, for numerical algorithms there is a huge number of program libraries available ready to be used.

A word of remark: these first two criteria are independent of each other, still sometimes contradictory, as repeated decomposition steps may result in very specialized modules which cannot be easily reused.

Modular intelligibility

This means that a module should be meaningful also on its own, and there should be no other ”supporting” modules. This is significant for code maintainability.

Modular continuity

This comes from the notion of a continuous function: a ”minor” change in the specification should mean only a ”minor” change in the program. This also implies that these ”minor” changes should apply only for a few modules, not for the whole structure of the program.

The extent of these ”minor” changes in the specification and in the code is not precisely defined, yet the concept is clear.

A simple example: in source code numeric and textual literals should be replaced by symbolic constants – in case of a change only the constant definition must be updated.

Modular protection

The protection of the whole program must be ensured also in abnormal circumstances. The effect of an error should be confined only to one – or maximum a few – module(s). The method of specifying for every input module the responsibility to check data integrity is a good practice and fulfills modular protection.

Basics of modularity

To fulfill all the criteria above, certain principles are essential to bear in mind.

Language support of modules

Modular design can only be efficient if the modules fit the syntactical units of the given programming (or design, or specification) language. This principle also implies that modules, supported by the given programming language must be compilable on their own.

This is a technical requirement, but if not met, the implementation of the above listed criteria may get troublesome. For example, if a system is partitioned into well separated subtasks (decomposition), they should also be managed as separate syntactical and compilation units. On the other hand, if a module is not a language unit, but is composed of more units, modular composition becomes impossible.

Without proper programming language support, the best methods are only like a ”bird without wings”. Programming language support can only be partly substituted by programming style or workarounds.

Few interconnections

Modules should communicate with as few other modules as possible.

Between modules various kinds of communications are possible: modules can call services of other modules, can share their data structures, etc. This principle is about limiting the number of such connections.

In a system consisting of n modules the number of interconnections should be close to the minimal amount of $n - 1$, and should be much less than the maximal possible $n * (n - 1)/2$.

This actually originates from the principles of continuity and protection: if there are too many interconnections, the effects of a tiny change or the propagation of an error would affect too many modules.

Weak interconnections

If there is communication between two modules, the minimum amount of information should be transferred.

This is also based on the principles of continuity and protection, as less interconnections can transfer fewer changes and errors to the other modules.

Explicit interfaces

This principle states that if two modules communicate with each other, it should be clearly signaled by at least one of them.

This concerns several criteria. Firstly, the criteria of composition and decomposition are relevant as they both require the clear notion of all interconnections. Secondly, the principle on continuity is vital as it is important to know exactly from the interface which other program units could be affected by change. Thirdly, modular intelligibility is also on important factor as interconnections cannot be clearly detected without it.

Information hiding

No (internal) information can be accessed or used from a module, only those pieces of information which are explicitly declared as public.

Applying this assumes the knowledge of all the modules only through their official interface. This means that every module basically consists of two parts: an interface and a private implementation which is not reachable from other modules. A programming language must support this restriction. Of course, it is possible to have the whole module in the interface part, but this should be more the ”exception”, than the norm.

Hiding information is the basis for continuity. Changing the hidden, private part of a module only, and keeping the interface unchanged will have absolutely no effect on the other modules communicating with it.

Open and closed modules

The above rules are supported by the majority of ”traditional” programming languages. Good module structure and information hiding can be implemented in Modula or Ada. Reusability demands more support – thus, let us introduce the following two definitions:

- A module is closed if it is reachable for other modules through a well defined interface only, and can only be used in an unchanged form. The implications are that a closed module is a separately compiled program unit, which may be stored in a program library, and may be linked on demand to the program.
- Open modules are extendable, meaning their services can be extended; for example, new fields can be appended to their data structures, which in turn lead to changes in their services.

In the traditional approach, open and closed modules are clearly conflicting requirements. ”Traditional” programming languages generally support closed modules, and the implementation of abstract user data types (module, package, etc.). As regards open module support, the hierarchical library structure of Ada 95 is a good example (for a detailed discussion, see Section 9.3.2).

The so called object-oriented approach offers a new possibility: the starting point here is the class – corresponding to the abstract data type –, and to create new classes from already existing and closed classes, a new concept – namely, inheritance is introduced.

Reusability

During software development very often similar (but not identical) program elements must be implemented. For example, most of the programs use ordering, searching, etc. Programmers would rather like to avoid implementing these again and again, and hope for reusable modules. Most of the problems emerge due to the fact the tasks are very similar, but not exactly the same. A module can be reused only if it provides general enough services.

To implement generalized code is actually more expensive in the short term (it requires more specific design and robust implementation) than implementing a concrete solution. Investing into it is worth only for professional programmers as they may later want to reuse the results for multiple programs.

What criteria must the module structure meet to support reusable code?

Variety of types

Modules must be implemented to work with various types. For example, by using a module declaring a stack type, the program should be able to support a stack of integers or strings. This requirement is usually supported by so called generic or template language constructs, but can also be implemented with inheritance.

Variety of data structures and algorithms

Attention must be paid to support all kinds of data structures a program may require – e.g. a tree, a list, an array, a sequential file, etc. All of these can be traversed sequentially, but of course, each with a different algorithm. A linear search algorithm is general enough only if all of these types are handled properly.

One type – one module

Operations of only one type should be packaged into one module so that the user of the module can better understand all the supported operations of a given type.

Representation independence

Changes in the representation of a data type should cause no effects outside the module. This can be achieved by using hidden representation and implementation.

A more general demand is that different representations of the same type should be supported at the same time. For example, addition and multiplication

of complex numbers are implemented more easily if using different representations. However, in traditional programming languages, it is not easy to meet this demand. What needs to be remembered is that objects should be usable without knowing their representation or implementation of their operations.

This requirement is also important for software extensibility. The problem can be formulated so that when applying a certain operation, the specific algorithm must be chosen according to the type. If this choice is implemented in the supplier module, it must know all possible implementations – yet, this would lead to oversized and unmanageable modules, and would not offer an easy way to handle new type variants.

Consider, for example, a graphical system which manages lists of points, lines, triangles, tetragons and circles. A list must contain elements of the same type; thus, this in the original source code of the module, data type is traditionally represented by a record with some variant part. Drawing, transforming etc. each of these shapes require different algorithms. This is handled in traditional code as a `case` statement with branches for all possibilities of the variant record.

Let us assume that we have purchased a program package which handles these shapes. If it later turns out that, for example, for drawing traffic signs we need hexagons, there are two options: we may contact the designer of the original package, and ask for a modification, or we may implement the new shape ourselves, with all the operations, including those which need no modification of the original package. In our program most likely a big `case` statement will handle the new shape. Both of these possibilities – changing the original source code, or changing the user code – are tedious, hardly manageable and costly.

Real solutions for these kinds of problems are offered by the features of object-oriented programming.

9.3.2 Language support for modules

After reviewing theory, we will now turn to evaluating programming language support for implementing modular programming.

Low level programming languages

In the beginning, there was only one programming language, the machine code based assembly. From assembly source code a special compiler, the assembler created the executable (machine code) form. The task of the assembler was relatively simple: to compile the assembly statements (so called mnemonics) and compute address references. The goals of such address references could also be symbols which were not defined in the same source file. This technique allowed programmers to produce separate compiling units and to introduce an early implementation of module composition. However, to produce the final executable, an additional step was needed right after compilation: during linkage the still unresolved references of separate compiled code units had to be computed.

The source of the compilation unit had to be compiled only once with the compiler, the newly created object code could be linked to different programs many times. To help the storage of the compiled code units, they were packaged in so called program libraries. This concept and methodology are valid for higher level languages too, but usually all this is done hidden by the compiler.

The closest language to assembly is the C programming language [KR89] designed especially for system programming. Thus, this language is still simple and "low level", but manages compilation units much more efficiently. It uses a precompiler which supports modular decomposition. Every compilation unit consists of two source files: public declarations (as the interface of the module is stored in so called header files), and actual implementations (they go in the source file). Where the module is used, its header file gets included on the source level (**#include**). As the same module may be included more than once through different modules, to avoid multiple (re)declarations, the conditional compilation feature of the precompiler must be used, like this:

```
#ifndef MODUL_H /* Module identifier */
#define MODUL_H
/* This part will be compiled only once */
. . . /* Interface declarations of the module come here */
#endif
```

A compilation unit can be successfully compiled if every referenced header file is reachable for the compiler. Skipping the linkage stage must be signaled with a special flag, however for executable code generation, listing already compiled units is sufficient. By using header files, the visibility of declarations gets also adjustable at some level, though this feature is not supported by the C programming language. The content of the header files are completely public, as every module can use it by including it. The declarations of the implementing module can (normally) not be accessed from the outside, so they can be labeled as local and private to the module.

Procedural programming languages

One of the many high level programming languages FORTRAN, [LV77] has introduced compilation units. FORTRAN programs are made of segments, one of which can be the main segment, the others can be a function (*FUNCTION*), a subroutine (*SUBROUTINE*) or a block data (*BLOCK DATA*).

Identifiers (except for the segment name and the *COMMON* data fields) are local to the segment. At the beginning of a segment, declarations are listed, followed by the executable part. The evaluation of the declaration part is static, assignment of the variables to memory is done in compile time. These segments are the compilation units of the language. There is no language support to describe interconnections between the segments, and in compile time there is no way to check the validity of these connections.

These restrictions were eliminated by the version FORTRAN 90 of the language. A new compilation unit, the module (*MODUL*) allowed the definition of an interface (*INTERFACE*) which helps to check cross-module connections already during compile time.

```
module fn_def
! This module defines the interface

  interface fn
    ! The interface declares function signatures

    function real_fn(param1, param2) result(return_value)
      real param1, param2, return_value
    end function real_fn

    function double_fn(param1, param2) result(return_value)
      double precision param1, param2, return_value
    end function double_fn

  end interface
end module
```

```
! A separate compilation unit contains two implementations
! with different signatures

function real_fn(param1, param2) result(return_value)
  real param1, param2, return_value
  return_value = param1*param2
end function real_fn

function double_fn(param1, param2) result(return_value)
  double precision param1, param2, return_value
  return_value = param1*param2
end function double_fn
```

```
program main   ! Main program module
 use fn_def    ! Referencing the interface definition

  real r1, r2, r3
  double precision d1, d2, d3
  data r1/2401.0/, r2/9.81245/
  data d1/2401.0d0/, d2/9.81245d0/
! Using through interface the two different function implementations
  r3 = fn(r1, r2)
  d3 = fn(d1, d2)
  write(*,'(1x,1p,e16.9,t25,d16.9)') r3, d3

end program
```

Modularity is even more thoroughly supported by the Modula programming languages, the development of which has reached its third generation already. Originally Niklaus Wirth developed it by drawing on his experience with Pascal. One of Modula-2's most successful characteristic is the support for explicit description for interfaces between modules. Modula-3 has practically fully inherited interface and module description [Nel91]. Modules are basic building blocks in Modula-3. Modules are named collections of declarations, such as constant-, type-, variable- and subroutine declarations. A module can make its own declarations accessible for other – client – modules. This is described by interfaces, but modules usually also have some hidden implementation parts.

Modules import the interfaces which they use, and export those which they implement. By using interfaces, modules can be safely compiled separately. There is no way for a reference to become invalid in an already compiled module. Each module is visible to the outside through their interface(s) which helps our program to become transparent without knowing all the details. This ensures a much simpler maintainability and easier learning.

Compilation units of Modula-3 are modules and interfaces. A module is like a block, except for the visibility of identifiers: entities are visible in a block if they are declared in the given block, or in one of the containing blocks; by contrast, in a unit only if they are declared in the given interface, or in one of the imported or exported interfaces.

Modules export an interface to assign a body to procedures declared in this exported interface. Modules or interfaces import an interface to make the declared entities from these imported interfaces visible. Module definition looks like this:

MODULE ⟨*identifier*⟩ **EXPORTS** ⟨*list of exported interfaces*⟩;
⟨*import statements*⟩;
⟨*declarations*⟩;
⟨*module body -- block*⟩ ⟨*identifier*⟩;

```
(* Simple example *)
MODULE Hello EXPORTS Main;(* Main: predefined, empty interface *)
    FROM SIO IMPORT Put; (* import statement *)
BEGIN
    Put("Hello world!");          (* body block *)
END Hello.
```

The program is a collection of modules and interfaces. It contains all the interfaces which are imported or exported by its modules or interfaces, and where all the procedures, interfaces and modules are defined only once at the most. Executing the program means executing the module bodies based on the following rules:

- If a module (M) depends on another module (N), the body of N will be executed prior to the body of M;
- M depends on N, if M uses at least one interface which is exported by N, or if M depends on such a module which depends on N;
- M uses an interface I, if M imports or exports I, or if M uses such an interface which imports (direct or indirect) I;
- in any other cases not covered by the above rules, the order of execution is implementation dependent.

Modula-3 also supports generics. Given the nature of modularity, only a module, or rather its interface part can be made generic. Generics can have parameters, but only interfaces can be used for these parameters. These must be enumerated in the **GENERIC INTERFACE** clause and also in the **GENERIC MODULE** description.

Open modules support in Ada

The Ada language also evolved from Pascal for system programming purposes. The most important aspect of its development was to guarantee program safety and validity, so particular attention was paid to exploiting the benefits of the modular approach.

One of the greatest strengths of Ada lies in the so-called *packages* where – with the help of the specification and body part – the user interface and the implementation of an abstract type can be clearly separated. So the implementation and all the clients using the services of the package can be compiled independently, as long as the specifications stay unchanged. This may work for smaller programs, but for implementing more complex tasks it could become cumbersome.

Consider the case when two logically independent packages want to use a *private* type together. Ada 83 supports two different methods to implement this. Firstly, if the given common type used in both packages is not declared as private, both packages may access it. But in this case, all packages using the given

type will see the type representation which seriously violates data abstraction. The other possible solution in Ada 83 is to implement the solution of the two logically independent tasks in one common package where in the private part the commonly used elements are declared. Abstraction is now not violated, but the bigger size of the package will substantially increase the costs of compilation. It is also not a good idea to place two logically independent programming elements in one common package.

The other significant problem with the packages of Ada 83 arise as soon as the services of an already existing abstract type are extended by adding new operations. After extending the specification part, not only the whole package must be recompiled, but also all of those applications which use the modified package, even if they never use the newly added functionality.

To avoid such problems, Ada 95 [Nyek98] introduced the hierarchical library structure and the notions of child package and child-subprograms. Features of a child package involve the following:

- From the point of view of visibility, it can be public or private;
- Logically it is subordinate to the parent package and can therefore access both its public and private parts;
- The name of the parent package is prefixed to the name of the child package;
- Any package can have a generic child, but a generic package can have only generic children.

In the remaining part of this section, we will discuss public child packages. Let us start with an example. The task is to implement a package managing complex numbers in a canonical form:

```
package Complex_Number is
  type Complex is private;
  function Constructor(Real, I: Float) return Complex;
  function "+" (arg1, arg2: Complex) return Complex;
  -- "-", "*" and "/" are likewise;
  function Real_Part(X: Complex) return Float;
  function I_Part(X: Complex) return Float;
private
...
end Complex_Number;
```

In a later stage, the task is extended – the handling of the trigonometrical form of complex numbers must be supported. In Ada 83 the new operations must

be inserted into the above package, whereas in Ada 95 the following solution is applicable:

```
package Complex_Number.Polar is
  function Polar_2_Complex(R, Theta: Float) return Complex;
  function "abs" (arg: Complex) return Float;
  function Arg(arg: Complex) return Float;
end Complex_Number.Polar;
```

From the point of view of visibility, the child package is treated as if its public part was appended after the public part of its parent, and the private part after the parent's private part. This implies that the child package does not only see the public part of its parent, but also the private part, meaning the package body may also access the parent's private part. Nevertheless, the newly declared operations are no primitive operations of the type, as they are not declared in the same package.

Thus, the private part of the child package connects to the private part of its parent. In the public part of a public child package, the private parts of the parent are hidden not to allow renaming, which could make the hidden parts visible to everyone. This would lead to serious violation of data abstraction. (For private child packages other visibility rules imply, as will be discussed later.)

Based on the previous example consider the following applications of visibility rules in practice:

```
with Complex_Number.Polar;
-- Calling syntax here is:
-- Complex_Number.Real_Part(),
-- Complex_Number.Polar.Arg()

with Complex_Number.Polar;
use Complex_Number;
-- Calling syntax here is:
-- Real_Part(),
-- Polar.Arg()

with Complex_Number.Polar;
use Complex_Number;
use Complex_Number.Polar;
-- Calling syntax here is:
-- Real_Part(),
-- Arg()
```

The body of the parent (using a **with** statement) can reach and use all of its descendants. The child package can use all entities of its parent automatically (without **with** or **use**). With **with** the child can also use all services of its

(compiled) siblings. The private part and the body of the specification of the child package can access the private part of the parent. Using **with** on a child package implies **with** on its parent. After executing **use** on a package, the entities of its children can be reached with the ⟨*parent name*⟩.⟨*child name*⟩ syntax.

All of these visibility rules should be implied recursively everywhere, so a multilevel hierarchy of parent and child packages can be built where the private part and the body of a child package can access the private part of all its parents, but the parents cannot see the child's private parts.

A package can, of course, have more children. With regard to our previous example, the package of complex numbers is best implemented in three parts: firstly, the base defines the private type and the four basic operations; secondly, one child defines the canonical form and implements its operations; and finally, another child defines the trigonometrical form and implements its special operations. This way as small portion of the code as possible must be recompiled if needed, and the number of possible errors is also reduced.

Thus, using a public child package can make a private type visible in more packages, and enables to extend a package without recompilation. This is another alternative – in addition to the **tagged** type in Ada 95 – for using extension in programming, since child packages can also have private parts, and implemented types from the parent can be extended there.

With regard to the naming convention (⟨*parent name*⟩.⟨*child name*⟩), note that this is not only a formal rule introduced by the designers of the language, thus, it is also crucial to identify functionally coherent elements, which, however, is not manifested so clearly in other object-oriented programming languages.

Developing more complex systems may require design decisions to break down the task without letting the clients even notice the change. Additionally, the package structure of Ada 83 has another weakness: partitioning a package body is only possible with the help of subprograms (procedures and functions). Although these subprograms were compilable as separate subunits (by specifying the **separate** keyword), any changes of the external (outside of the subprograms) declarations within the body caused the recompilation of all subunits.

As a solution for this problem, Ada 95 introduced the so called *private child packages*. The private child package differs from public child packages in terms of visibility. Hence, in the remaining part of this section we will focus on these differences. A private child package can be declared at any point of the hierarchy, using the **private** keyword at the beginning of its specification. Every part of the private child package – including the public part of the specification – can access the private part of the parent. A private child package is only seen from that part of the hierarchy where the parent is the root; but it is not seen in the public specification of its siblings, or from those packages which are not ancestors of its parent. These rules ensure that private child packages cannot violate data abstraction. As the private child packages themselves are not visible either from the outside, or from the public parts of their public siblings, which may eventually access them, it is impossible to make the private part visible.

Consider the following implementation snippet of an operating system:

```
package OS is
  type File_Descriptor is private;
...
private
  type File_Descriptor is new Integer;
end OS;

package OS.Exceptions is
  File_Descriptor_Error, File_Name_Error, Permission_Error: exception;
end OS.Exceptions;

with OS.Exceptions;
package OS.File_Manager is ...
  type File_Mode is (Read_Only, Write_Only, Read_Write);
  function Open(File_Name: String; Mode: File_Mode)
                                              return File_Descriptor;
  procedure Close(File: in File_Descriptor);
end OS.File_Manager;

procedure OS.Interpret(Command: String);

private package OS.Internals is ...
end OS.Internals;

private package OS.Internals_Debug is ...
end OS.Internals_Debug;
```

In the above example, the parent package contains the type declarations for the system, and the three public children (two packages and one subprogram) ensure the functional decomposition of the system. Consider the *OS.Interpret()* procedure which is a child-subprogram of the *OS* package, so within its body even the **private** part of the package is accessible. As regards the visibility of the **private** packages, they are both children of the *OS* package, meaning they are also visible in the body of all other children of *OS*. On the other hand, all parts of the *OS.Internals* and *OS.Internals_Debug* packages see the whole specification of *OS*, that is, both the public and **private** part. This is, however, safe, as the specification of these two **private** child packages are not visible from the specification of any other package, meaning the only definition in the **private** part of *OS* cannot become visible through them.

Object-oriented programming languages

Object-oriented languages address the problem of open and closed modules with the help of inheritance. They are also used in other language constructs as they support the implementation of information hiding and encapsulation, and modularity too.

The package support of the Java language is a good example. Every package defines a separate *namespace* which makes the names of all types within it unique. This help avoid mixing up the types with the same name in different packages.

A package is declared on the language level by compilation units. Compilation units of a package contain the code of the package; that is, the declarations of classes and interfaces, referred to as types.

The package structure of Java is hierarchical, meaning between the packages subordinate relationships may exist. A package can have any number of subpackages. A subpackage is also a package – it only states to which package it belongs. Therefore, a subpackage can also have subpackages, and the structure of packages compares to a tree.

The subpackage has no stronger connection with the parent package above it in the hierarchy as any other package. The goal is to define an organized structure of packages and help programmers follow the development process. For example, visibility is not better or worse for subpackages; it remains the same as for any other package. However, with the help of packages, visibility can be adjusted in a more subtle way: without explicitly setting visibility to **private**, **protected** or **public**, a new visibility mode, the so called **package protected** takes effect. Elements in this visibility mode are public within the same package, but private from outside of the package.

At the beginning of every compilation unit, one can specify to which package the given compilation unit belongs to. Without explicitly declaring it, the compilation unit will belong to an unnamed package.

After the package declarations, the import declarations follow. Import declarations allow using the simple name instead of the extended name for public types declared in other packages. Import declarations have an effect only on the compilation unit they belong to. Thus, in the other compilation units of the same package, the imported types must be referenced with full names, or they must be imported there too.

Importing does not use import declarations from imported packages. Hence, the types imported by imported packages cannot be referenced by simple names, but only if they get explicitly imported. It follows that importing is not transitive. Importing does not concern the subpackages of the imported package, meaning they cannot be referenced by a simple name. Referencing by a simple name is possible after an extra import only.

To implement a package only the compilation units must be made, and in each unit, it must be declared to which package it belongs. This is followed by the import and type declarations in every compilation unit.

As an example, to illustrate all the above mentioned, consider a package called *fruit.apple*, which consists of two compilation units, namely *Jonatan.java* and *Starking.java*:

```
// Jonatan.java
package fruit.apple;      // Compilation unit of a package
public class Jonatan {    // Type from the package
   public Jonatan(){ System.out.println("I am Jonatan!"); }
};
```

```
// Starking.java
package fruit.apple;      // Compilation unit of the same package
public class Starking {   // Type from the same package
   public Starking() { System.out.println("I am Starking!"); }
};
```

The following test program demonstrates the use of the import methods:

```
// Test.java Test program within an unnamed package
import fruit.apple.*;           // On-demand import
import fruit.apple.Starking;    // Single-type import

public class Test {
   public static void main(String[] args) {
      Object j1, j2;
      j1 = new Jonatan();
      j2 = new Starking();
   }
}
```

Functional programming languages

As a reminder, functional programming languages are characterized by the following features: there are no variables (only so called *bindings*), no assignments, object values cannot be changed during program execution (this is called transparency), and instead of the regular sequence, branches and loops it uses function composition, pattern matching and recursion. These features make such languages easy to analyze. Functional programming languages are regarded as declarative, as opposed to those of the third generation, so called imperative programming languages.

Next, we will study modularity support in the ML language. Here the basic elements of modules are signatures and structures. Signatures specify structures. In terms of their roles, they correspond to interfaces, types of a class or package specifications in other languages. With regard to structures implementing signatures, they correspond to implementations, classes or packages.

The signature is the description or specification of a program module or structure. A structure can contain any type of declarations: type constructors, exceptions, bindings of objects to symbolic names, and also substructures and shares of substructures explicitly aiding modularization. The signature is the element-wise specification for the declarations of a structure. A structure fits in with or implements a signature if it contains all the exact, or more generic declarations specified by the given signature.

Consider the following simple example where `a'` is a type variable. By using it we can make this specification of the queue polymorphic regarding its element type:

```
signature QUEUE =
  sig
    type a' queue
    exception Empty
    val empty : a' queue
    val insert : a' * a' queue -> a' queue
    val remove : a' queue -> a' * a' queue
  end
```

An existing signature can be reused in two ways to define new signatures: by embedding and by specialization. Consider the following examples demonstrating the two methods:

```
signature QUEUE_WITH_TEST =
  sig
    include QUEUE
    val is_empty : a' queue -> bool
  end
signature QUEUE_AS_LISTS =
  QUEUE where type a' queue = a' list * a' list
```

Embedding is used to expand a signature, whereas specialization compares to implementation as it specifies the representations of the abstract types. The two signatures are equivalent if their specifications are pair wise type-equivalent.

The structure is a program unit consisting of declarations. Consider the following polymorphic implementation of the above `QUEUE` signature (the reason for the double list representation is that in ML the built in list type, in terms of its operations, functions more like a stack):

```
structure Queue =
  struct
    type a' queue = a' list * a' list
    exception Empty
    val empty = (nil, nil)
    fun insert (x, (h,t)) = (x::h,t)
    fun remove (nil,nil) = raise Empty
      | remove (h,nil) = remove (nil, rev h)
      | remove (h, x::t) = (x, (h,t))
  end
```

9.4 Encapsulation

The essence of data abstracting program design is that types are not only some sets of elements, but they are used together with their operations as one unit. This unity is expressed as the notion of *encapsulation*.

The principle of encapsulation is closely related to representation hiding. Only those subprograms are entitled to access the inner structure of a type which are implementing the operations of the type, in other words, which are part of the encapsulation. For every other subprogram the type is opaque.

In object-oriented languages the role of encapsulation is especially important since during inheritance the derived type inherits the *primitive* operations of the ancestor type, that is, those operations which are encapsulated with that type. In the Java or C++ languages these operations are defined in the class namespace, but for example in Ada 95 these operations are introduced after the type declaration, and contain the given type in their signature.

9.5 Representation hiding

One of the most important element of data abstraction is representation hiding. Representation hiding guarantees that the levels of the program architecture can be modified independently of each other. Modification of the implementation on a lower level will not propagate upwards in the structure since higher levels are only affected by the type specification.

For the support of representation hiding, the programming language – usually through its syntactic features – prevents the users of the type from accessing its representation. Since the representation of the type is hidden, it is called *opaque*. In data abstracting program design, of course, all types must be opaque. It goes without saying that, after all, every type is represented by a sequence of bits if digging deep enough into the structure of the program. Hence, this notion is usually used for types where the user of the type has no access to structures at a higher level than the sequence of bits.

9.5.1 Opaque type in C

Not every language has features to implement opaque types. In the C language, for example, there is no such feature. The following example is a code snippet of the implementation of a complex number in C:

```
/* complex.h */ ...
typedef struct __complex_struct {
    double re;
    double im;
} Complex;
extern Complex init(double re, double im);
```

```
/* complex.c */ ...
public Complex init(double re, double im) {
    Complex z;
    z.re = re;
    z.im = im;
    return z;
}
```

In the above solution the representation is accessible for the user of the type. By declaring a complex number z, the reference $z.re$ is syntactically correct. This also means that if later, instead of the canonical form of the complex numbers, the trigonometrical representation is used, the former properly functioning programs will possibly not work since the *Complex* type will not have the *re* field anymore. As there is no language support for this, the programmer must be disciplined not to abuse this possibility.

Another representation, which seemingly solves this problem, is as follows:

```
/* complex.h */ ...
struct __complex_struct;
typedef struct __complex_struct* Complex;
#define COMPLEX_ERROR NULL
extern Complex init(double re, double im);
```

```
/* complex.c */ ...
struct __complex_struct {
    double re, im;
};
Complex init(double re, double im) {
    Complex z = malloc(sizeof(struct __complex_struct));
    if (z != NULL) {
        z->re = re; z->im = im;
    }
    return z;
}
```

In the above solution the programmer cannot access the *re* or *im* components. The representation is not hidden, now it is not the *__complex_struct* structure, but a pointer type to it. And this remains "visible", that is, the usual pointer operations can be applied to it. In some respects the result is even worse, than in the former version, since now the programmer also has to destroy the complex number objects, as creating a *Complex* object causes dynamic memory allocation. On the other hand, pointer types support the `==`, `+`, `-` operators, but their meaning is far from the expected complex number operators equality, addition and subtraction. Likewise, the functioning of the assignment operation also differs from the expected. If used incorrectly, this difference may lead to errors which are hard to find (the compiler cannot offer any help).

9.5.2 Private view of Ada types

In the Ada language the complete hiding of the representation is possible. The specification part of the package, which defines the type, contains the description of the representation. Importantly, though, it is separated from the declaration of the public identifiers; it is located in the private part (after the **private** keyword) of the specification. The content of the private part can logically "not be seen"; that is, the user of the type cannot reference anything from there.

```
package Complex_Numbers is
    type Complex is private;
    function NewComplex(RE: Float; IM: Float := 0.0)
                                                    return Complex;
    ...
private
    type Complex is record
        RE : Float := 0.0;
        IM : Float := 0.0;
    end record;
end Complex_Numbers;
```

```
package body Complex_Numbers is
    function NewComplex(RE: Float; IM: Float := 0.0)
                                                  return Complex is
    begin
        Return Complex'(RE,IM);
    end New;
    ...
begin
end Complex;
```

9.5.3 CLU abstract data types

The CLU language emphasizes support for data abstraction, and thus it offers good features also for representation hiding too. The following example is a code snippet from the implementing module of the complex type:

```
complex = cluster is newcomplex, re, im . . .

    rep = struct[ re : real, im : real ]

    newcomplex = proc (r: real, i: real) returns ( complex )
        return ( up(rep$_(re: r, im: i)) )
    end newcomplex
    re = proc ( c : cvt ) returns ( real )
      return ( rep$get_r( c ) )
    end re
    im = proc ( c : complex ) returns ( real )
      return ( rep$get_i( down(c) ) )
    end im
. . .
```

The representation is defined by the `rep` type. Representation hiding by separating the concrete (`rep`) and the abstract (`complex`) types is so strong that the language requires the usage of explicit type conversion between them. Unlike in Ada, where the representation is a hidden internal structure of the type, here it is a truly different type which happens to meet the given type specification complying to the notion defined in Section 5.1.

In the above example, it is shown that the type conversion between abstract and concrete types can be done in two ways: by using the automatically created `up` and `down` operations which make the conversion from concrete to abstract, or vice versa. These conversions are used in the `newcomplex` constructor and in the implementation of the `im` operation.

Most of the operations during their function first convert the abstract typed parameters to concrete types, perform the operation and convert the result back to abstract type, so that a simplified notion is introduced. If within the

signature of the operation the `cvt` keyword appears, it signals that the above conversation must be done automatically. For example, in the parameter list of the `re` operation, the actual parameter must be of the abstract type, while the formal parameter must be a concrete type.

9.5.4 Visibility levels

To support representation hiding, languages often introduce levels for the visibility of the components of compound types which levels determine who exactly can access the given component. There are languages, such as in C, Pascal or Modula-2, where no such restrictions exist.

In Ada 83, the previously introduced two levels of visibility are used. The specification of the type is public, thus every user module can access it; however, the representation and the implementation are hidden, and may be accessed only in the implementation module of the type.

In object-oriented languages, the following three visibility levels are typically distinguished:

- *public* – the given component is visible for everyone;
- *protected* – the given component is visible only for the descendants (only in languages supporting inheritance);
- *private* – this is the completely hidden part of the representation, its components can only be used in the implementation of the operations

These three levels of visibility are used in, for example, Ada 95 or C++. Java uses an additional level for the containing package of a class.

In some languages (such as Eiffel) visibility is regulated in a more sophisticated way, that is the type implementation prescribes which classes and descendants of the classes can access the given component. Visibility levels are discussed in more detail in Chapter 10.

9.6 Separation of specification and implementation

In those languages which support the ”one module – one type” principle, usually type specification can be described in a compilation unit separate from the implementation. This separation serves three purposes:

- It enables the modules to be developed independently from each other;
- It supports the unobserved modification of representation and implementation;
- All the information needed to use the type is separately available, meaning, the implementation can be delivered in a compiled state too.

The full hiding of the representation is (unfortunately) impossible. For the user module it is unnecessary – and even inadvisable – to know representation details, whereas the compiler must know how much memory has to be allocated to the object of the given type. Thus, if the specification and the implementation are in two separate units, the compiler must be able to determine the required amount of memory allocation from the specification part.

C and C++ header files

In the C/C++ languages the specification must be physically copied into every compilation unit which wishes to use the given type. Thus, a real separation of the specification and the implementation is not possible here, at least not on the level of the language.

To enable the copying of the specification, it must be stored in a separate, special source file – the header file – which is included by a so called precompiler in an appropriate compilation unit. If a type specification is given multiple times, it causes an error; thus, it is best avoided by the programmer.

The header file must contain the specification of the operation and also of the representation which will be fully accessible since it is copied into the user module source code. In the C++ language the usage of the visibility classes can prevent access to the inner structure of the representation, but since the header file and the implementation module of the type have no connection – defined by the language – modifying the header file can grant access to the inner details of the type without even changing the source code of the implementation.

Mapping to pointers

To solve the problem we may alternatively create a situation in which the abstract type is always represented by a pointer which points to the actual representation byte sequence. In cases like this, allocation can be exactly determined at the location of the usage, without revealing anything from the actual representation. However, due to the newly introduced indirection and to dynamic memory management, the usage is made more complex now. This solution has been shown in the C language in Section 9.5.1, and it is also shown in the following Modula-2 example:

```
DEFINITION MODULE Complex_Numbers;

    TYPE Complex;

    PROCEDURE NewComplex(R, I: REAL): Complex;
    PROCEDURE Add(Z1, Z2: Complex);
...
END Complex_Numbers.
```

```
IMPLEMENTATION MODULE Complex_Numbers;

    CONST GUARD = 12345

    TYPE Complex = POINTER TO ComplexStr;
    TYPE ComplexStr = RECORD
        R: REAL;
        I: REAL;
        G: CARDINAL;
    END;

    PROCEDURE NewComplex(R, I: REAL): Complex
    VAR Z: Complex;
    BEGIN
        NEW(Z);
        IF Z#NIL THEN
            Z^.R := R; Z^.I := I; Z^.G := GUARD;
        END;
        RETURN Z;
    END;

    PROCEDURE Add(Z1, Z2: Complex);
    BEGIN
      IF (Z1#NIL) AND (Z2#NIL) AND
         (Z1^.G = GUARD) AND (Z2^.G = GUARD) THEN
          Z1^.R := Z1^.R + Z2^.R;
          Z1^.I := Z1^.I + Z2^.I;
      END;
    END Add;
...
BEGIN
END Complex_Numbers.
```

This solution essentially mimics the functioning of languages which work with object references – such as Java, Eiffel, or CLU. However, without automatic constructors and destructors, reference counting and garbage collecting memory management, it is much harder and more dangerous to use. The advantage of this solution is that it enables the modification of the actual representation – that of the pointed data structure – without the need to recompile the user program parts of the module.

Visibility areas

Ada has chosen a totally different approach than the ones presented above. Here the specification and the representation are held in one compilation unit, meaning the representation is physically not hidden. However, with its syntactical features, the language ensures that this information cannot be utilized by the developer of other program units using the type. The compiler can exactly determine the memory needs of the given type. Seemingly this solution does not differ at all from the header files of the C and C++ languages, but as the language handles the specification and implementation parts of a package as one unit, changing any of these requires the recompilation of the package.

Languages not supporting physical separation

In languages (such as Eiffel) the physical separation – in separate source files – is not possible, meaning the allocation problem will not occur. However, the development tool supports multilevel "views" of the code, and thus the irrelevant details regarding the usage can be hidden.

The Java language lacks this feature – in fact, it merges the specification and the implementation. The structure of the Java bytecode reveals the interface of the given class, the public methods can be selected, and the inheritance information can be recovered, so it is possible to deliver the implementation also in compiled state.

9.7 Management of module dependency

The highest level building blocks of the programs are the modules. The functioning of the program is an interaction of these modules. Every module requires services, and based on them implements new services. Thus, there is a kind of dependency between the modules since, for example, changing one module may require the recompilation of its dependent modules etc.

Some of the programming languages (such as C or C++) require the programmer to handle these dependencies, and manage every compilation unit separately. This is why there are special tools (such as make) which server this purpose specifically. Since, however, the language does not offer any kind of support, these solutions are not perfect, e.g. something may be recompiled too many times, or the necessity of recompilation may not be recognized etc.

Thus, in other languages, the compiler is responsible for the management of these dependencies. For example in the Ada language the **with** statement must specify on which other compilation units the given module depends. This guarantees that before the compilation of the module the specification parts of all those modules are compiled (if necessary) on which the given module depends. For more details, refer to Chapter 9.3.

9.8 Consistent usage

Perhaps one of the most important requirements against a programming language is that it must not distinguish between built-in and user defined types in their usage. Compound types (arrays, records, etc.) and variables must be defined, and as with built-in types, the expressions must be built from them etc.

Further implication of the unified usage is that the new type must fit in with the logic of the language. If, for example, in a given language there is a convention for a type to be read by the *Read* operation, it is important that for the user type the programmer must be able to define a *Read* operation, meaning the overloading of the *Read* identifier must be supported by the given programming language.[2] *Overloading* an identifier means that at a given point of the program source, the identifier may have multiple definitions in effect. In this case, on the basis of the referencing environment, the compiler decides which of the multiple possible interpretation will be applied for the given identifier. Overloading is found in almost all of the programming languages since, for example, the + operator in the 3+4 and in the 3.13+4.2 expressions denote separate operations. In cases like this, on the basis of the type of the parameters (operands), the compiler determines which interpretation of the given identifier will be chosen. This is called *static linking* (for more on overloading, refer to Section 7.6.). For the present purposes, the most important question is if this can be caused by the programmer. That is, if it is possible for an identifier defined by the programmer to have multiple valid definitions at some point in the program source, from which the compiler may choose the actual instance to be executed based on the type of the arguments.

The features in the block structured languages are not considered overloading if variable names overlap e.g. if there is an x variable defined within block A, and there is another variable with the same name defined in the embedded block B; in this case, x from B will mask the x from A, which may be referenced only by qualified name ($A.x$). The features in object-oriented languages are not considered overloading either, if the derived class changes the definition of an operation declared within the ancestor class. This is called *overriding*, and the actual definition corresponding to the reference is chosen not in compile time – statically –, but in runtime – dynamically. This behavior is called *dynamic linking* (for more details, see Chapter 10).

A special case of overloading is the operator overloading (see Chapter 7.6.1) which occurs when the operators of the language (+, −, ∗, /, =, <, >, **sizeof**, **new** etc.) are overloaded. This has a great significance in using the user types naturally. For example, if a matrix type is defined, for addition the +, for multiplication the ∗ infix operators must be used; furthermore, the element of

[2] Overloading is not always necessary since in object-oriented languages each class typically defines a separate name space. Thus, the *Read* operation can be defined in every class without the use of overloading.

matrix A from row i and column j should be accessible in the form $A[i,j]$, because this is how it is dealt with in mathematics. The question arising is if the language supports changing the arity or precedence of the operator or not. In most of the cases, this is not possible. There are languages, such as Java which generally support overloading, but not operator overloading.

In the languages supporting operator overloading, it is a crucial question if the assignment is defined as a statement or as an operator. If the assignment is an operator (such as in C++), its overloading is allowed; however, if it is a statement (such as in Ada) it is not allowed, of course.

Another question concerning operators is if it is possible to define additional to the existing ones, that is, if there are "free operators". Eiffel, for example, supports the definition of new unary prefix an binary infix operators.

9.9 Generalized program schemes

The goal of the programmer is to develop and maintain programs as quickly and efficiently as possible. In terms of efficiency, general types and subprograms should be developed to be reused in the actual and in future programming tasks on the widest possible scale. Programming methodology teaches solutions which are applicable widely. This intention is also reflected in the various programming languages. This section will introduce language features which serve the purpose of reusability. Some of these features are so common and ordinary that we do not even expect them to belong to this category. These will not be discussed in detail, but appear only for the sake of completeness in the bellow summary.

Subprograms

Some details of the program code – used multiple times at different locations, decreasing the unnecessary repetitions of the code and increasing maintainability – must be developed in the form of subprograms. Essentially all programming languages support subprograms.

Parametrization of subprograms

The most important parameters of a subprogram are referenced by formal names when calling the subprogram. This way the subprogram can be used on a wider scale, and by choosing the right parameters it becomes more portable. It may be surprising to categorize parametrizable subprograms as a separate group, but not every programming language supports the parametrization of subprograms. Assembly languages or BASIC support subprogram calling, but communication between the caller and callee subprogram must be implemented by hand, e.g. through global variables. For further details about subprograms and their parametrization, see Chapter 7.

Parametrization of types

Parametrizing the types make them more general and widely usable. Such parametrized types are, for example, the unconstrained discriminant records in Ada, or the unconstrained array types (see sections 6.4.3. and 6.6.4.). Since the same result can be achieved by dynamic memory management and explicit type construction methods, this solution is supported by relatively few languages.

Subprograms as parameters

Subprograms can receive other subprograms as parameters, so for example a programming thesis put in general terms can be implemented on a wider scale for a variety of purposes.

A common solution is the usage of pointer types (see Section 5.6.), but some programming languages – e.g. Ada 83, BASIC – do not support this feature. In object-oriented languages, dynamic linking and special, so called function objects can be used for the same purpose (see Chapter 10).

In the dBase, Clipper and FoxPro languages, similar can be achieved by using macros. These languages, utilizing their interpreted nature, allow textual variables to be executed as program code, thus, for example, by passing the name of a subprogram, the appropriate macro may call the code.

Types as parameters

Programming theses, basic data structures and data types use abstract concepts also for their definitions, and put only the most necessary restrictions to them. As a result, not only one, but many types can typically fulfill these constraints. The implementation of the theses, data structures and types must be specified once, and may be used with all types that fulfill the requirements.

To satisfy these needs there are different possibilities. In the C language, for example, there is no such feature on the language level, but similar effects – to a limited extent – can be achieved by using the macros of the precompiler.

In object-oriented languages, this effect could be achieved theoretically by using only inheritance and polymorphism since the necessary common behavior may be merged into one abstract class – or interface – which will then be the common ancestor of all the classes fulfilling the requirements. The only problem with this solution is that the combination of all the possible requirements must be known in advance – already at the design time of the basic class hierarchy.

Consequently, languages supporting type parameters introduce some kind of separate construction. In these the types can be specified as formal parameters which can be used like any other type within the structure. When passing an actual value to this formal parameter, a variant of the structure will be created for the given types, called the instantiation of the structure.

In the C++ language, type parameters are supported by templates. A template is similar to a macro substitution where the compiler substitutes all occurrences of the formal parameter with the actual type. Certain minimal syntax checking is performed by the compiler, and the formal parameter is marked to denote a type. Thus, important properties of the parameter type (such as certain operations) needed by the subprogram or type to be implemented, cannot be specified as a requirement. When developing a template, the compiler cannot verify if the operations on the parameters exist, and it cannot avoid instantiation with an inappropriate type. Errors caused by this will emerge only at the instantiation. However, an advantage of the language is that the instantiation of the template happens completely automatically.

In the Eiffel programming language the template is called generic, and as throughout the whole language, a strong object-oriented approach prevails here too. The formal parameters of the generic can be classes, additional behavior requirements may be specified by naming the expected ancestor class of the actual parameter. This solution is better than that of the C++ since the operations used within the generic must be the operations of the given ancestor, and thus correctness can be verified when developing the generic. A disadvantage of the solution is that the required common behavior must be known in advance to define an appropriate common ancestor. However, by now most of the problems can be solved by applying inheritance and polymorphism.

In the Ada 95 language, the parametrization of the templates is much broader and more flexible than in the previously mentioned Eiffel. Here on the one hand, an Eiffel-like ancestor type can be specified, but the actual parameter may also be required to be of a given type class (e.g. discrete, enumeration, any type). An additional operation may be specified too and thus, a mandatory common ancestor can be avoided. The operations specified by the programmer enable an even more flexible usage of the template. For example, if a template using an order relation is instantiated with integer numbers, an order relation different than the usual one, such as the partial order relation of divisibility, can be specified. All things considered, however, Ada lacks the flexibility of C++, which is ensured by the automatic instantiation.

Notice that in the case of parametrizing with types in C++, the notion ”type” is used more from the viewpoint of the programming language (see 5.1.). That is, here only the type value set (that is the set of the representing elemental value sequences) matters, whereas in Ada and the Eiffel the type is handled as a unit of type values and their operations. For more on templates, see Chapter 11.

Higher level structures as parameters

The above outlined solutions do not cover all the possible requirements. There may be problems which should be parametrized with more complex, higher level structures. In Ada 95, for example, a template can have another template as a

parameter, or to be more precise, the actual parameter may be required to be a package or subprogram created by instantiating a given template.

Additionally, there may arise the need for parametrizing by an array of types where the number of types to be passed as a parameters is determined at instantiation, but all the passed types are of the same type class.

What requirements the programmers will face in the future is far from clear yet, but it is very likely that the programming languages and methodologies will respond to the needs of the market.

9.10 Summary

Let us summarize again which questions should be answered when examining what data abstraction support a programming language offers:

- *Does the language support representation hiding?*
  The examples of languages have shown that some languages support very complex visibility management (e.g. Java, Eiffel or C++), yet others have no features to hide the internal structure of the types (e.g. C, Pascal or FORTRAN).
- *If the language does not support representation hiding, which features can be utilized to achieve similar effects?*
  In the C and Modula-2 languages, if the data type is hidden behind a pointer, the advantages of representation hiding are partly maintained since the representation of the type may be modified without recompiling the user modules. There are languages (e.g. the BASIC language) where this is not possible, though.
- *For representation hiding what features are supported?*
  For hiding the internal structure of the data type, a large number of variants and features are available in the different languages. It is common for the identifiers declared within the type to be categorized into visibility classes which determine the modules which can reference the given identifier. The Ada 83 language only distinguishes two visibility classes: the public class, which is accessible for everyone, and the hidden *private* class which can be used only by the implementing module of the type. Ada 95, C++ and many other object-oriented languages support the classic three-class model where the previous two categories are extended by a third, *protected* visibility class, the identifiers of which are accessible within the implementing module of the type, and also in the modules implementing the derived types. Java introduces four classes, defining a separate category for the types of the same package. The other extreme is represented by Eiffel where for each identifier the accessing classes can be individually controlled.

- *Does the language support modularization?*
  In the BASIC or Pascal languages, the ”one program – one module” principle is applied, meaning every program is essentially one compilation unit. This limitation prevents two or more developers to work on a program. Thus, most of the programming languages – aimed at professional use – support some form of modularity. As a basis for modularization, data abstraction methodology uses breakdown on types. Thus, languages which support this feature – such as CLU – usually follow the ”one type – one module” principle.
- *What features are available for separating specification and implementation?*
  In languages such as Java or Eiffel, this separation is only logical. In Eiffel, in particular due to the flexibility of the visibility classes, the outer interface of the type may change from user to user. In the Modula-2 and Ada languages, the specification and implementation parts are also separated physically, into two separate compilation units. In the CLU language the abstract type – specification – and the concrete type – implementation – fall under different type categories.
- *How does the language treat the dependencies between modules?*
  Some languages do not handle dependencies at all. Such are, for example, the C and C++, where dependencies are described by a separately introduced language with its own interpreter (`make`). In the Ada and Modula-2 languages, dependencies are also strongly supported. They are defined with the features of the language and the compiler also handles them.
- *Is there a difference between built-in and user defined types?*
  The equivalent usage of user and built-in types is the most important sign of supporting data abstraction methodology. For a consistent use, manifold features are used. Such usage depends on the nature of the language on the one hand, and on its conventions, on the other.
- *Can user types be used in type constructions or in the representation of newer types?*
  Most of the languages support type constructions, except for those languages which do not support the creation of user types (such as BASIC), and certain script languages. In Perl, for example, the hash tables can only contain scalar values and strings.
- *Can user types be passed as parameters?*
  To represent user types, usually some kind of type construction is used. In some languages – such as ALGOL 60 – composite types cannot be passed to subprograms as parameters, and neither can user types using such representation. These kind of languages are rare; in most languages user types can freely be passed as parameters.

- *Is there overloading in the language?*
  One of the basic requirements of consistent usability is to name the same kind of operations with the same name. In the Ada language, for example, a type can usually be read from a text file with the *Get* operation. Consistent usability requires that user types define *Get* operations which will need the overloading of the *Get* identifier. Nevertheless, in object-oriented languages (such as Java) classes often define separate name spaces, meaning their methods will not overload methods of other classes with the same name. In this way, the principle of consistent usability is not violated. In the C programming language, overloading of identifiers is not possible, meaning the same operations of different types must be named differently.
- *Is operator overloading allowed?*
  Operator overloading is a special case of overloading which can of course add much to consistent usability since mathematical formulas may be inserted into program source directly, which in turn make them clearer and more readable. Despite this, there are languages (e.g. Java), which support overloading, but do not support operator overloading.
- *Can a new operator be defined in the language?*
  Additionally to the above mentioned, the Eiffel language also enables the definition of new prefix or infix operators for the types. However, this possibility is very rare.
- *Can procedure subprograms be made in the language?*
  The simplest kind of subprograms is the procedure which is supported nearly in all languages. The C language is a special case where all subprograms were originally functions, but the return value could be omitted.
- *Can function subprograms be made in the language?*
  Most of the programming languages support function subprograms, but the BASIC language is an exception as it only supports procedural subroutine calls.
- *Can objects be passed as parameters to the subprograms?*
  In most of the cases, the answer to this question is yes, but there are exceptions such as the BASIC or assembly languages.
- *Can parametrized types by defined?*
  Parametrized types are not supported in many programming languages. In object-oriented languages a similar effect can be achieved, if properly parametrized constructors and dynamic memory management is used. We have seen examples of unconstrained types in Ada where the value of the parameter given at the instantiation of the object influences the size and the internal structure of the object.
- *Can subprograms be passed as parameters to subprograms?*
  In most of the languages passing subprograms as parameters is possible. The common solution is the passing of a pointer as parameter referencing

the subprogram. An exception to this is, for example, the Ada 83 language where no such pointers may be defined, or the BASIC language. In object-oriented languages, the so called function objects and dynamic binding can be used to achieve similar effects.

- *Is there a way to use types as parameters?*
  In many languages, including the modern ones, this is not possible. Nevertheless, many languages offer completely different solutions, such as the templates of C++, Eiffel or Ada. It is worth considering what features can be used to specify the formal parameter types what regulations can be made, and how flexible these regulations can be.

9.11 Exercises

Exercise 9.1. Implement the rational number type in multiple languages. Make sure that the implementation should fit in with the type system of the language. Compare the solutions.

Exercise 9.2. Implement the stack data type in multiple languages. What is the difference in the use between the procedural and object based approach of the language tools?

9.12 Useful tips

Tip 9.1. In this chapter we have introduced and examined the realization of the complex number type in some programming languages, such as the Ada solution with a package managing complex numbers in a canonical form (9.3.2) and the C code (9.5.1) when opaque types have been discussed. Analog to these implementations, the rational number type can be easily represented by two integer numbers, the numerator and the denominator, where the represented rational number is computed as the division of the numerator by the denominator. By this definition it is clear, that the denominator must be never zero (this condition should be always asserted to be true as part of the type invariant), and there are endless possible representation of the same rational number. For the representation parts getter functions should be implemented, and a constructor should be given accepting the numerator and the denominator. For a minimal set of the standard mathematical operators (such as addition, subtraction, division, multiplication, comparison of equality and ordering) an internal normal form can be used, where the numerator and denominator have the smallest possible values to represent the given rational number.

Tip 9.2. The stack is a Last-In-First-Out (LIFO) data structure, where the last element added to the structure must be the first one to be removed. This is

equivalent to the requirement that the push and pop operations (for adding and removing an element) occur only at one end of the representing sequential collection, referred to as the top of the stack. A stack may be implemented to have a bounded capacity. If the stack is full, it cannot accept any new entities to be pushed. If the stack is empty, a try to remove any items will fail. The representing data structure should support sequential access for the push and pop operations and the storage of an arbitrary type, so a generic approach should be followed.

9.13 Solutions

Solution 9.1. In Ada user defined types can be encapsulated into a package. The specification can be the following:

```
package Rational_Numbers is
   -- a rational number has a numerator and a denominator
   type Rational is private;
   Zero_Denominator : exception;
   Zero : constant Rational;

-- creates a new rational number
   function Rat(Num: Integer:=0; Den : Integer:=1) return Rational;

-- Numerator of the rational number
   function Numer(R : Rational) return Integer;

-- Denominator of the rational number
   function Denom (R : Rational) return Integer;
   -- operations:
   -- addition
   function "+"(A,B : Rational) return Rational;
   -- subtraction: minuend - subtrahend  = difference.
   function "-"(Minuend,Subtrahend : Rational) return Rational;
   -- multiplication
   function "*"(A,B : Rational) return Rational;
   -- division: dividend/divisor=quotient
   function "/"(Dividend, Divisor : Rational) return Rational;
   -- comparisons:
   -- less
   function "<"(A,B : Rational) return Boolean;
   -- less or equal
   function "<="(A,B : Rational) return Boolean;
    -- greater
   function ">"(A,B : Rational) return Boolean;
   -- greater or equal
   function ">="(A,B : Rational) return Boolean;
   procedure GET(Number : out Rational);
   procedure PUT(Number : in Rational);

   private
-- the representation
      type Rational is record
        Numerator : Integer;
        Denominator   : Integer;
      end record;

      Zero : constant Rational := (0,1);
end Rational_Numbers;
```

The body is the following:

```
with Text_IO; use Text_IO;
package body Rational_Numbers is
   package Int_IO is new Integer_IO(Integer); use Int_IO;

--  normal form
function Norm(Inp : Rational) return Rational is
   Outp : Rational;
   Divisor : Integer:= 2;
   Sign : Integer;
  begin
    if Inp.Denominator=0 then raise Zero_Denominator ;
    end if;
    Outp:=Inp;
   if (Outp.Denominator*Outp.Numerator < 0) then Sign := -1;
   else Sign := 1;
   end if;

   Outp.Denominator:=abs(Outp.Denominator);
   Outp.Numerator:= abs(Outp.Numerator);

   while (Divisor  <= Outp.Numerator) and
         (Divisor  <=  Outp.Denominator) loop
      if (Outp.Numerator mod Divisor =0) and
         (Outp.Denominator  mod Divisor  = 0) then
           Outp.Numerator:=Outp.Numerator/Divisor;
           Outp.Denominator:=Outp.Denominator/Divisor;
      else Divisor := Divisor +1;
      end if;
   end loop;
   Outp.Numerator:=Outp.Numerator*Sign;
   return Outp;
end Norm;

procedure Common_Denominator(A,B : in out Rational) is
C,D : Rational;
begin
    A:=Norm(A);
    B:=Norm(B);
    C.Denominator:=A.Denominator*B.Denominator;
    D.Denominator:=C.Denominator;
    C.Numerator:=A.Numerator*B.Denominator;
    D.Numerator:=B.Numerator*A.Denominator;
    A := C;
    B := D;
end Common_Denominator;

function Rat(Num: Integer:=0; Den : Integer:=1) return Rational is
     New_Rat : Rational := (num, den);
 begin
    return New_Rat;
 end;

 -- Numerator of the rational number
 function Numer(R : Rational) return Integer is
 begin
    return R.Numerator;
 end Numer;

 -- Denominator of the rational number
 function Denom (R : Rational) return Integer is
    begin
    return R.Denominator;
 end Denom;
```

```
 -- operations:
   -- addition
 function "+"(A,B : Rational) return Rational is
       C,D : Rational;
   begin
      C := A;
      D := B;
      Common_Denominator(C,D);
      C.Numerator:=C.Numerator+D.Numerator;
      return Norm(C);
end "+";

   -- subtraction: minuend - subtrahend  = difference.
   function "-"(Minuend,Subtrahend : Rational) return Rational is
      C,D : Rational;
  begin
     C := Minuend;
     D := Subtrahend;
     Common_Denominator(C,D);
     C.Numerator:=C.Numerator-D.Numerator;
     return Norm(C);
  end "-";

   -- multiplication
 function "*"(A,B : Rational) return Rational is
    C : Rational;
  begin
      C.Numerator:=A.Numerator*B.Numerator;
      C.Denominator := A.Denominator*B.Denominator;
      return Norm(C);
end "*";


   -- division: dividend/divisor=quotient
function "/"(Dividend, Divisor : Rational) return Rational is
   quotient : Rational;
begin
    quotient.Numerator:=
               Dividend.Numerator*Divisor.Denominator;
    quotient.Denominator :=
               Dividend.Denominator*Divisor.Numerator;
    return Norm(quotient);
end "/";

   -- less
function "<"(A,B : Rational) return Boolean is
   C,D : Rational;
begin
   C := A;
   D := B;
   Common_Denominator(C,D);
   return C.Numerator < D.Numerator;
end "<";
   -- less or equal
  function "<="(A,B : Rational) return Boolean is
     C,D : Rational;
  begin
     C := A;
     D := B;
     Common_Denominator(C,D);
     return C.Numerator <= D.Numerator;
end "<=";
  -- greater
function ">"(A,B : Rational) return Boolean is
```

```
   C,D : Rational;
begin
   C := A;
   D := B;
   Common_Denominator(C,D);
   return C.Numerator > D.Numerator;
end ">";
 -- greater or equal
function ">="(A,B : Rational) return Boolean is
   C,D : Rational;
begin
   C := A;
   D := B;
   Common_Denominator(C,D);
   return C.Numerator >= D.Numerator;
end ">=";

procedure Get(Number : out Rational) is
   OK : Boolean:= False;
begin
   Put("Give the numerator:");
   Get(Number.Numerator);
   New_Line;
   while not OK loop
      Put_Line("Give the denominator, it is not 0!:");
      Get(Number.Denominator);
      if Number.Denominator =0 then
        Put_Line("PLease give it once more, it s not 0!");
       ELSE
         OK:= True;
       end if;
   end loop;
   end Get;

   procedure Put(Number : in Rational) is
   begin
       New_Line;
       Put("The numerator:");
      Put(Number.Numerator);
      New_Line;
      Put("The denominator:");
      Put(Number.Denominator);
      New_Line;
      end put;

end Rational_Numbers;
```

A small trial:

```
with Text_IO, Rational_Numbers;
use Text_IO, Rational_Numbers;
procedure Rational_Trial is
   R1, R2, R3: Rational;
begin
   R1:=Rat(3);
   R2:= Rat(8,4);
   Put(R1);
   Put(R1+R2);
   R3:= R1*R2;
   Put(R3);
   Get(R3);
   -- etc.
 end Rational_Trial;
```

In C++ user defined types can be written encapsulated in a class. A possible solution:

```
#include <iostream>
using namespace std;
class Rational {
  public:
    Rational(int n = 0, int d = 1) {
      numerator = n;
      denominator = d;
    }
  void operator = (Rational c)  {
     numerator = c.numerator;
     denominator = c.denominator;
    }
//norm  transforms its parameter to normal, simplified form
 Rational norm(Rational Inp){
      Rational Outp;
      int Divisor = 2;
      int Sign;
      if(Inp.denominator==0) throw "Zero denominator";
      Outp=Inp;
      if (Outp.denominator*Outp.numerator < 0) Sign=-1;
      else Sign=1;
      if (Outp.denominator<0) Outp.denominator=-Outp.denominator;
      if (Outp.numerator<0)
         Outp.numerator=-Outp.numerator;
      while ((Divisor <= Outp.numerator) &&
             (Divisor <= Outp.denominator)) {
        while ((Outp.numerator%Divisor==0) &&
               (Outp.denominator%Divisor==0)) {
              Outp.denominator=Outp.denominator/Divisor;
              Outp.numerator=Outp.numerator/Divisor;
        }
       Divisor++;
      }
      Outp.numerator=Outp.numerator*Sign;
      if (Outp.numerator==0) Outp.denominator=1;
      return Outp;
    }

void Common_Denominator(Rational& r1, Rational& r2) {
   Rational r11, r21;
   r11= norm(r1);
   r21=norm(r2);
   r11.denominator=r1.denominator*r2.denominator;
   r21.denominator=r11.denominator;
   r11.numerator=r1.numerator*r2.denominator;
   r21.numerator=r2.numerator*r1.denominator;
   r1=r11;
   r2=r21;
}

// addition of  rational numbers
 Rational operator + ( Rational r2 ){
    Rational r1;
    r1.numerator = numerator;
    r1.denominator = denominator;
    r1 = norm(r1);
    r2 = norm(r2);
    Common_Denominator(r1,r2);
    r1.numerator = r1.numerator + r2.numerator;
    return norm(r1);
   }
```

```
// subtraction
Rational operator-(Rational r2 ){
    Rational r1;
    r1.numerator = numerator;
    r1.denominator = denominator;
    r1 = norm(r1);
    r2 = norm(r2);
    Common_Denominator(r1,r2);
    r1.numerator = r1.numerator - r2.numerator;
    return norm(r1);
   }

// multiplication
Rational operator*(Rational r2 ){
    Rational r1;
    r1.numerator = numerator*r2.numerator;
    r1.denominator = denominator*r2.denominator;
    return norm(r1);
   }

//division
Rational operator/(Rational r2 ){
    Rational r1;
    r1.numerator = numerator*r2.denominator;
    r1.denominator = denominator*r2.numerator;
    return norm(r1);
   } // etc.

// the << operator can only be given using the friend construct:
friend ostream & operator<<(ostream& out, Rational& number);

private:
      int numerator;
      int denominator;
}; // end class Rational

ostream & operator<<(ostream& out, Rational& number){
       out << number.numerator << '/' << number.denominator;
       return out;
    }

int main() {
   Rational a(2,6),b(8,12);
   cout <<"the a:"<< a<<"\n";
   cout <<"the b:"<< b<<"\n";
   Rational c;
   // c is initialized to zero
   cout<<"the c:"<<c<<"\n";
   c = a + b;
   cout<<"the sum of a and b:"<<c<<"\n" ;
   c = a - b;
   cout<<"a - b = "<<c<<"\n" ;
   c = a*b;
   cout<<"a * b = "<<c<<"\n" ;
   c = a/b;
   cout<<"a / b = "<<c<<"\n" ;

return 0;
}
```

Solution 9.2. In the programming language Ada the package construct can serve both for encapsulation of user-defined data types with hidden details of representation of the type value set and the implementation of the operations of the type and for creating standalone objects. In this latter case, if we use the generic

possibility of the language and write templates for these objects, the result is very close to the concept of "class" in the object oriented languages. We give a possible solution for both, and the main program demonstrates the differences in instantiation and use.

```
generic
  Max_Size: Integer;
  type Elem_Type is private;
package  Gstackt is
   Empty, Full : exception;
   type Stack_Type is private;
   procedure Push( S : in out Stack_Type;  Elem : in Elem_Type );
   procedure Pop ( S : in out Stack_Type;  Elem : out Elem_Type );
   function Is_Empty( S : in Stack_Type ) return Boolean;
   function Is_Full ( S : in Stack_Type ) return Boolean;

   private
    subtype Index is Integer range 1..Max_Size+1;
    type  Elements_Array is array ( Index ) of Elem_Type;
    type Stack_Type is
    record
      Elements : Elements_Array ;
      First_Free : Index := 1;
    end record;
end Gstackt;

package body Gstackt is
   procedure Push( S : in out Stack_Type; Elem : in Elem_Type ) is
   begin
      if S.First_Free < Index'LAST then
         S.Elements(S.First_Free):=Elem;
         S.First_Free := Index'Succ( S.First_Free);
      else
         raise Full;
      end if;
   end Push;
   procedure  Pop ( S : in out Stack_Type;  Elem : out Elem_Type ) is
   begin
      if S.First_Free > Index'First then
         S.First_Free := Index'Pred( S.First_Free);
         Elem := S.Elements(S.First_Free);
      else
         raise Empty;
      end if;
   end Pop;
   function Is_Empty ( S : in Stack_Type ) return Boolean is
   begin
      return(S.First_Free=Index'FIRST);
   end Is_Empty;
   function Is_Full  ( S : in Stack_Type ) return Boolean is
   begin
      return(S.First_Free=Index'Last);
   end Is_Full;
end Gstackt;

--this package will be a template for objects
generic
  Max_Size: Integer;
  type Elem_Type is private;
package  Gstobj is
   Empty, Full : exception;
   procedure Push( Elem : in Elem_Type );
   function Pop return   Elem_Type ;
   function Is_Empty  return Boolean;
```

```
   function Is_Full   return Boolean;
end Gstobj;

package body Gstobj is
   --the representation is also hidden in the package body
   subtype Index is Integer range 1..Max_Size+1;
   type  Elements_Array  is array ( Index ) of Elem_Type;
   Elements : Elements_Array;
   First_Free : Index := 1;
   procedure Push( Elem : in Elem_Type ) is
   begin
      if First_Free < Index'Last then
         Elements(First_Free):=Elem;
         First_Free := Index'Succ( First_Free);
      else
         raise Full;
      end if;
   end Push;
   function  Pop return Elem_Type  is
   begin
      if First_Free > Index'First then
         First_Free := Index'Pred( First_Free);
         return  Elements(First_Free);
      else
         raise Empty;
      end if;
   end Pop;
   function Is_Empty  return Boolean is
   begin
      return(First_Free=Index'First);
   end Is_Empty;
   function Is_Full  return Boolean is
   begin
      return(First_Free=Index'Last);
   end Is_Full;
end Gstobj;

with Gstobj,Gstackt, Text_IO; use Text_IO;
procedure Gstdemo is
--instantation:
package st1 is new Gstobj(10,Integer); --this is a new stack object
package st2 is new Gstobj(20, Float); --this is also a new stack object
package Intst is new Gstackt(5,Integer); use Intst;

My_stack:stack_type; --this is an object of the type from the package

st1el,stel : Integer;
st2el : Float;

begin
--small examples for possibilities of the use:
  if not st1.Is_Full then
    st1.Push(5);
  end if;
  st1el:=st1.Pop;
  st2.Push(5.0);
  st2el:=st2.Pop;
  Push(My_Stack,2);
  Pop(My_Stack,stel);
 exception
  when st1.Empty=>Put_Line("st1 empty");
  when st1.Full=>Put_Line("st1 full");
  when Full=>Put_Line("My_Stack is full");
  when Empty=>Put_Line("My_Stack empty");
  when st2.Empty=>Put_Line("st2 empty");
  when st2.Full=>Put_Line("st2 full");
```

```
end Gstdemo;
```

A possible C++ solution will also be a template, it is a good practice to write its own exceptions too:

```
#ifndef STACK_EXCEPTIONS_HPP_INCLUDED
#define STACK_EXCEPTIONS_HPP_INCLUDED
#include <exception>
class Stack_Exception : public std::exception {
    public:
        const char* what() const throw() { return "Stack error"; }
};
class Empty_Stack : public Stack_Exception {
    public:
        const char* what() const throw() { return "Stack is empty!"; }
};
class Full_Stack : public Stack_Exception {
    public:
        const char* what() const throw() { return "Stack is full!"; }
};
#endif // STACK_EXCEPTIONS_HPP_INCLUDED
```

A solution for the Stack template can be:

```
#ifndef Stack_H
#define Stack_H

#include <iostream>
#include <string>
#include "solution_stack_exceptions.hpp" //the previous exceptions
template<typename T>
class Stack {
    private:
        unsigned int length;
        unsigned int head;
        T* arr;
    public:
        Stack(unsigned int length);
        ~Stack();
        void push(const T& new_elem);
        T  pop();
        T& top() const;
        bool isEmpty() const;
        void print() const;
        //prints the stack to the screen
};
// The implementations of the operations:
// Constructor, the size of the Stack will be the value of the parameter
template<typename T>
Stack<T>::Stack(unsigned int length) {
    this->length = length;
    this->head = 0;
    this->arr = new T[length];
}
//destructor, deletes the dynamically used memory
template<typename T>
Stack<T>::~Stack() {
    delete [] this->arr;
}
//  new_elem will be put to the top of the stack
template<typename T>
void Stack<T>::push(const T& new_elem) {
      if(head != length) {
           arr[head++] = new_elem;
      } else {
```

```
            // a greater array is needed!!
            T* arr2 = new T[this->length*2];
            for(unsigned i=0; i<this->length; i++)
                 arr2[i] = arr[i];
            this->length *= 2;
            delete [] arr;
            arr = arr2;
            arr[head++] = new_elem;
       }
}
// returns the element on the top of the stack if it exists
template<typename T>
T& Stack<T>::top() const {
     if(isEmpty()) {
         throw Empty_Stack();
     } else {
         return this->arr[head-1];
     }
}
//deletes the top element from the stack  if it exists
template<typename T>
T Stack<T>::pop() {
     if(isEmpty()) {
         throw Empty_Stack();
     } else {
         head--;
         return arr[head];
     }
}
template<typename T>
bool Stack<T>::isEmpty() const {
     return head == 0;
}
template<typename T>
void Stack<T>::print() const {
     std::cout << "content: ";
     for(unsigned i = 0; i < head; i++) {
         std::cout << arr[i] << " ";
     }
     std::cout << "\n";
}
#endif // Stack_H
```

An example for the instantiation and the use of the stack type:

```
#include <cstdlib>
#include <iostream>
#include <string>
#include "solution_stack.hpp"
using namespace std;
int main() {
   try  {
      Stack<int> v(1);
      v.push(3); v.push(2); v.push(1);
      v.print(); // the values will be printed to the screen
      cout << "top(): " << v.top() << endl;
      cout << "pop(): " << v.pop() << endl;
      v.print();
      cout << "Empty? " << (v.isEmpty() ? "yes" : "no") << "\n";
      //the answer will be 'no'
//etc.
   } catch(const exception& e) {
      cout << "Error! " << e.what();
   }
   return 0;
```

```
}
```

10 Object-oriented programming

In object-oriented programming we encapsulate the data and the functions that work on the data. The different parts of the class have different scopes. One of the most important attributes of object-oriented programming is that the classes can be settled to a hierarchy due to inheritance. Inheritance makes that the subclasses of a class contain the member variables and methods of the superclass. The implementation of inheritance in different programming languages raises a lots of interesting questions.
In this chapter we will discuss the theory of object-oriented programming and it's implementation in different languages.

We try to model reality with our programs, and if we look at the history of Computer Science we can find plenty of solutions to make this process easier.

The object-oriented paradigm is a method to model reality (see [Mey00], [Ang97] and [Bud91]) through view the existing elements of the world as objects. Every object is represented by its inner state (with their content, with their variables) and methods. Thus we can say, that object-oriented programming is an approach of programming, and there are system design methodologies based on this approach. These methodologies are about the full development process, they have solutions about the feasibility studies, analysis, design, implementation, and they also give alternatives for testing and maintenance.

Special object models have special attributes to make it possible to behave as the real world object that they mimic. During the *analysis* process we consider the system as a collection of collaborating objects. During design and implementation processes we create these objects. According to the object-oriented approach we can understand the mechanisms of the real world by getting the model closer to reality. By deeper understanding we can build a clearer and more flexible program.

We call a program object-oriented if it is a sum of collaborating objects, where every object has its own responsibility.

10.1 The class and the object

People notice the things of reality, simplify them, distinguish them, and also organize them. The final goal is to make it possible to discover and understand the complex movements of the world. To achieve this we do modeling. For modeling we use fundamental processes (algorithms) such as *abstraction*, *distinction*, *classification*, *generalization*, or even *reduction*, *partition* and *connection*.

Abstraction is a way to narrow the world to deal only with the parts that are needed for solving the problem. This means that we forget about the currently irrelevant parts of the world, and we highlight the essentials.

What we consider as an *object*? In object-oriented programming the objects are independent units of the reality that we want to model. Each object can be defined by its *inner state* and its responses to *the possible messages sent to the object*. This "respond" can mean that the inner state of the object changes, or the objects execute some "operations", tell some value to the outer world, send a message to other objects, instantiate, remove, copy, move other objects, etc.

By the selection of the objects attributes that are important to us, we can classify them, we can put them under some categories, by putting the objects with the similar attributes to the same categories or classes, and the objects with different attributes will be put to other classes. The method of classification is done by *generalization* and *specialization*. We look for permanent similarities or differences to put them into tighter or wider categories or classes.

Classification is part of our natural way of thinking. We put objects with the same type of attributes (variables), and with the same behavior description(*methods*) to the same *class*. The classes of objects hold the attributes of their objects. Every *object* is an *instance* of a *class*, and it has all the specialties of its class, it takes the specifications of the class for its data and its methods as well.

We can say that an object holds information, and can execute tasks by requests. This means that every object is a sum of *data* and *methods*. Methods can fulfill the object specific functions or can describe the behavior of the object.

Every object always has an *inner state*, which is described by the actual values of its data. After calling a method the state of the object can be changed. Objects know and remember their inner state, and they start to execute their methods from an *initial state*, and they can enter to an other state. The execution of the following methods starts from the final state of the previous method.

It is important to *identify* the objects. In real life everything can be identified. It is possible to have two objects with the exactly same inner state, but it does not mean that they are identical.

All these notions mean the *static attributes* of objects. The *dynamic model* describes the temporal behavior of the system. (By this, we mean the influences of the objects, the events of the system, their chronological order, the schedule of actions and methods, the states and the changes of the states.) In the dynamic model, it is obvious that we investigate the connections of objects and the environmental impacts on them. Objects are not alone, they are connected to each other.

The connected objects communicate with each other. During communication they *send messages*. These messages are usually represented by functions that can be called from the outside of the object. We indicate the methods by the identifier of the called object, and we can also pass arguments to the methods. *Obj.message(parameters)* If we expect some kind of response from the object

that we can receive through the passed arguments or by a return value of the method.

The most suitable feature to describe classes is the *abstract data types* . (More in Chapter 9). We have to describe the possible *set of values* of the objects, and we have to determine the *set of available methods* with their abstract definition. This way we give a type specification, which ignores the representation, and only determines the services of the object towards the world. We hide the representation and the corresponding implementation of the methods.

10.1.1 Classes and objects in different languages

Let's see how we can create and use some classes and objects in some programming languages.

SIMULA 67

The innovation and central concept of SIMULA 67 is the object which has its own variables and activities by its class declaration. Class declaration is a pattern and objects which fit this pattern belong to the same class.

```
Class Rectangle(RectangleName, Width, Height);! Class with 3 parameters;
  Text RectangleName; Real Width, Height; ! Specification of parameters;
Begin
  Real Area, Circumference;                    ! Attributes;

  Procedure Refresh;                           ! Method;
  Begin
    Area          := Width * Height;
    Circumference := 2*(Width + Height);
  End of Refresh;

  Procedure WriteOut;                          ! Method;
  Begin
    OutText(" I am a Rectangle "); OutText(RectangleName); OutImage;
    OutText(" Width: ");         OutFix(Width,2,6);
    OutText(" Height: ");        OutFix(Height,2,6);
    OutText(" Area: ");          OutFix(Area,2,7);
    OutText(" Circumference: "); OutFix(Circumference,2,6); OutImage
  end of WriteOut;

  Refresh;                                     ! Life cycle of Rectangle
  OutText("Rectangle was created"); OutImage;
  WriteOut;
End of Rectangle;
```

The instance of the *Rectangle* class can be created with the *new Rectangle(* `"Small"`*, 2, 3)* statement, it has the methods: *Refresh* and *WriteOut*.

Smalltalk

In Smalltalk everything is an object even classes. At first sight this means that classes have variables and methods. This consistency of Smalltalk makes it possible for classes to instantiate their objects through the own methods of the class.

In terminology of Smalltalk methods are procedures which execute due to the received messages. We can create a new class by the following steps:

```
Object subclass: #Account
  instanceVariableNames: 'balance'
  classVariableNames: ''
  poolDictionaries: ''
  category: nil !
```

The `Account` will have one variable called `balance`. We can define methods like this:

```
!Account class methodsFor: 'instance creation'!
  new
    |r|
    r := super new.
    r init.
    ^r
  !!

!Account methodsFor: 'instance initialization'!
init
  balance := 0
!!
```

The `new` method is a method of the class, a constructor, which instantiate a new object. The `init` method belongs to the objects of `Account` and sets an initial value of the variables of Account objects.

An instance of the class can be created with the `Account new` statement.

We should mention that due to the idea that everything is an object, the control statements are also objects. Let's see how we can write a fixed loop in Smalltalk. It was implemented as a method of the `Number` class with the following syntax:

⟨*lower boundary*⟩ `to:` ⟨*upper boundary*⟩ `do:`
`[:`⟨*loop variable*⟩ `|` ⟨*statements of loop*⟩`]`

For example:

```
a := Array new: 10.
1 to: 10 do: [ :i | a at: i put: 0 ]
```

According to Smalltalk we send the `to: do:` messages to `1` where the `10` after `to:` is the upper boundary of the loop and the block after `do:` is executed.

C++

We can declare a new class in C++ like this:

```
class Rectangle {
    int x, y;
  public:
    void setup(int, int);
    int area() { return (x * y); }
};
```

The *Rectangle* class has two hidden variables x and y. The class has two public methods the setup and the one that calculates the area. We have declared the area calculation method in the definition of the class, and we can define the setup method outside of the class definition in the following way:

```
void Rectangle::setup(int x1, int y1) {
   x = x1;    y = y1;
}
```

We can instantiate a *Rectangle* as a static or a dynamic variable:

```
Rectangle house;                  // statically allocated instance.
house.setup(5,3);
int houseArea = house.area();     // houseArea = 15

Rectangle* yard = new Rectangle; // Dynamically allocated variable
yard->setup(20,17);
int yardArea = yard->area();      // yardArea = 340
```

Object Pascal

In Object Pascal [Can00] we can make a new class with the **class** keyword:[1]

[1] To be compatible with earlier Turbo Pascal versions they have left the **object** version as well.

```
type
  TDate = class
    private
      Year, Month, Day: Integer;
    public
      procedure Setup(y,m,d: Integer);
      function LeapYear: Boolean;
      ...
  end;
```

Objects are instantiated dynamically,[2] after the variable declaration we have to call the constructor of the class (See at: 10.3) to allocate and initialize the required memory partition (*Create*). After using the object the programmer has to free the memory (*Free*).

```
var
  ADay: TDate;
begin
  ADay := TDate.Create(...);      // Instantiation.
  ADay.Setup(2003, 4, 6);         // Usage.
  ADay.Free;                      // Free the memory.
end;
```

Class types have data fields, methods, and *properties* (**property**). A property is a name which can reach the data fields of the object by given reading and writing methods.

```
property Month: Integer
  read Month write Month_Write;
```

If the property is in a statement, then it can get a value by the data or method after the *read* directive. If it is in the left hand side of the statement it will pass its value to the variable or method which was given after the *write* directive. In the example above the objects of *TDate* class has a *Month* property which gives back the *Month* field when it is read, and calls the *Month_Write* method on write.

Eiffel

It is hard to call Eiffel [Mey91] object-oriented, it would be better to call Class-oriented. You cannot instantiate an object independently from a class, and classes cannot behave as objects. In Eiffel, every class is a compilation unit.

As a type a class defines a set of objects and their behavior, that can exist during the execution of the system. It is even true backwards: every object that can exist, is an instance of one of the classes of this system.

[2] In spite of Turbo Pascal.

In Eiffel we only have one modularization feature, which is the class. In this case, building a system means that we analyze what type of objects the system will need for its work, and we write a class for each one of them. Sometimes we only need a class for one purpose: to collect some methods. In this case, that class does not need to have any attributes. A class in Eiffel looks like this:

```
class COMPLEX
  feature
    realV, imagV : REAL
  feature
    setup(r, i : REAL) is
      do
        realV := r;
        imagV := i
      end;       .....
end -- class COMPLEX
```

The *COMPLEX* class has two attributes (*realV* and *imagV*) and one method (*setup*). To create a new object we have to declare a variable. (For example: *z : COMPLEX*.) After declaration we have to allocate memory for the object with the *!!* operator or with the**create** command. The environment frees the memory automatically.

Java

Java [Nyek08] is a highly object-oriented language, which means every Java program is a set of related objects and classes. The execution of the program is nothing else than calling the methods of objects and classes.

Writing a program means to define one or more new classes. Writing the control-flow of the program is nothing else, than writing, overriding, and calling methods. (Or defining the suitable event handler.)

The smallest independent unit of the language is the class. Class is the model of logically coupled objects which has the same type. This model is wholly defined unit, and it appears to be unified. It is described by data field definitions and method declarations. During the execution the program instantiates the classes and by this it creates objects. In Java objects are handled dynamically, every variable is a reference of an object. The next example shows the definition of a new class [Nyek08] with two data fields and one method.

```
public class Employee {
  public String name;
  public int salary;
  public void raiseSalary(int incrementValue) {
    salary += incrementValue;
  }
}
```

The following code defines an *Employee* type local variable, instantiates a new object, sets the name and salary of the object, and raises the employee's salary as well.

```
Employee e;
e = new Employee();
e.name = "John Doe";
e.salary = 50000;
a.raiseSalary(6300);
```

C#

In C#, we can see the impact of C++, Java and Visual Basic combined with new features. The syntax of C# classes is close to the syntax of C++ classes, as we can see in the following example made after the [Csref03] page:

```
using System;
class Person {
    private string name ="N/A";
    private int age = 0;
    public string Name {
        get { return name; }
        set { name = value; }
    }
    public int Age {
        get { return age; }
        set { age = value; }
    }
    public override string ToString() {
        return "Name = " + Name + ", Age = " + Age;
    }
    ....
}
```

C# classes can hold member variables, methods and *properties* (*property*). These attributes do not have special keywords like in Visual Basic or in Object Pascal. In the first example, we can use the *Age* and *Name* properties of the *Person* class by **get** and **set** *accessors*. A **set** accessor of a property has a

special *value* variable which holds the value given by the user in the assignment statement. In the *ToString* method the **get** accessor is called automatically.

```
Person somebody = new Person();
...
somebody.Name = "Kate";
somebody.Age = 99;
Console.WriteLine("Personal data:", somebody);
```

Through the accessors we can write cleaner code, with more readable code syntax even with the proper handling of the hidden variables. (It is much more straightforward than the traditional get/set methods.) For example if we want to increase with one the age of the *somebody* object we can simply write:

```
somebody.Age += 1;
```

If we had written separate methods for setting and getting the *age* variable of the object, than our code would look like this:

```
somebody.setAge(somebody.getAge() + 1);
```

Ada

Ada 95 is an extension of an existing language. Since one of the most important criteria was to remain compatible with Ada 83, we can declare classes and traditional complex data types too. For this the programming structure of Ada 95 is not strictly object-oriented (See: [Ada95]). In Ada 95, the designers have not introduced a whole new concept to define classes but they have improved the concept of the existing record. Thus the classes (which are special records) hold only the attributes, the methods of the class can be defined outside in a special way. As a result, we cannot talk about traditional classes in case of Ada.

If we want to encapsulate the attributes and methods of a class we can put the record (which holds the attributes) and the methods (which handle the attributes) to the same **package**.

```
package Person_Type is
   type Person is tagged private;
   procedure Write_Name (P : in out Person; Name : String );
   procedure Write_Address (P : in out Person; Address : String );
   procedure Write_Age (P : in out Person; Age : Integer );
   function Name (P : Person ) return String;
   function Address (P : Person ) return String;
   function Age (P : Person ) return Integer;
   procedure Show (P : in Person );
private
   type Person is tagged record
      Name : String (1 .. 30);
      Address : String (1 .. 30);
      Age : Integer;
   end record;
end Person_Type;
```

The representation of *Person* type is hidden. We have specified methods to set and get Name, Address and Age, and also the *Show* method. The body of the methods should be given in the body of the package.

Python

Python's class and object concept is in the half way between C++ and Modula-3. Every class is implemented as a Python dictionary. It has a mapping to its variables and methods. Objects can be used through references. Multiple names can point to the same object, which is considered as aliasing. Since Python has reference semantics, if an object is changed through a name, then every name would point to the same changed object.

Every object has a *__dict__* object, which holds the names (methods and variables as well) that were defined in the class. It means that when we say *foo.bar = 42* then a *bar* attribute will be looked up in the dictionary of the object, and it will be set to the specified value. We can add new attributes to an object in run time, and we can also remove an attribute (with the *delattr* method) from an already instantiated object. This solution also means that we cannot have an object which has a variable and a method with the same name.

Methods of an object are also objects. They can be referenced, stored in a variable, or assigned to an other reference. However, variables just can come to life when we access them, but methods must exists when we first invoke them.

```
class Foo(object):
    num = 42;
    def f(self):
        return 'num is: '+num;

f = Foo();

print f.num;              # Prints 42
f.num=0
print f.f();              # Prints num is 0
f.other_num=24;           # Valid call
print f.other_num;        # Prints 24
print f.g();              # Invalid call of method g,
                          # because it does not exist
```

Scala

Scala is a functional programming language, or a multi-paradigm language which has first-class functions, and also class definitions. Scala was designed to run on JVM thus it is highly compatible with Java. Although Scala is often mentioned as a Function extension of Java it has a totally different class and object concept.

First, we can make new objects without any class definition. We can just simply introduce its name and values. When we make a new instance (which should be done with the `new` operator, and by the name of the object) from that object, it is initialized, and fully functional.

```
object Complex {
    var real : Double = 3.0
    var imag : Double = 0.0
    def length : Double => Math.sqrt(real * real + imag * imag)
}

object Test {
    def main() {
        Complex.imag = 4.0
        println(Complex.length())
    }
}
```

In the example, we have introduced `Complex` as a new object; it has a type, which we do not know, and it has an instance: `Complex` itself. If we run this code, this will print `5.0` as an output. But we can also introduce `Complex` as a class.

```
class Complex {
   var real : Double = 0.0
   var imag : Double = 0.0
   def length : Double => Math.sqrt(real * real + imag * imag)
}

object Test {
   def main() {
      var c1 = new Complex()
      var c2 = new Complex()
      c2.imag = 4.0
      println(c1.length()) //3.0
      println(c2.length()) //5.0
   }
}
```

Now we have a class which describes a complex number and we also have two separate instances (`c1` and `c2`). The main difference is that in the first case we could not make any new instance of the class, and in the second case we had to call `new` to instantiate a `Complex` object.

10.2 Notations and diagrams

It is important to have a common notation. Previously, we have met the concept of *class*, *object*, data members, methods and state. We say that we view class as the sum of data and methods, and objects are instances of classes with momentary state. Now we try to show these concepts with diagrams.

10.2.1 Class diagram

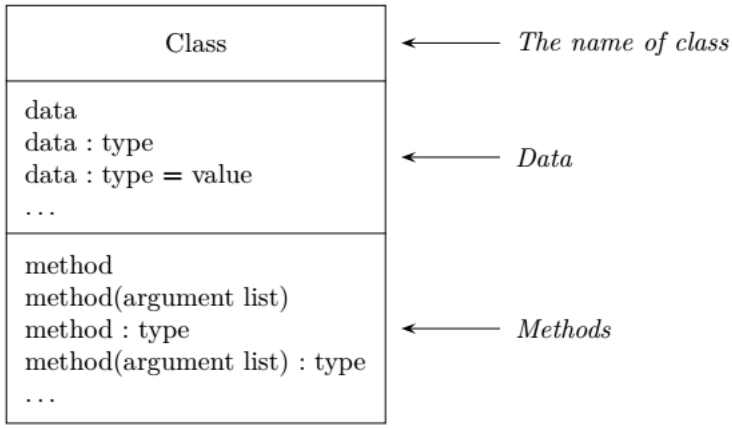


Figure 10.1: Class diagram

The class diagram holds the name of the class, the data used to instantiate the objects, and the methods.

Data are the representation of the state. Methods stand for the messages. Every method can be followed by an argument list.

10.2.2 Object diagram

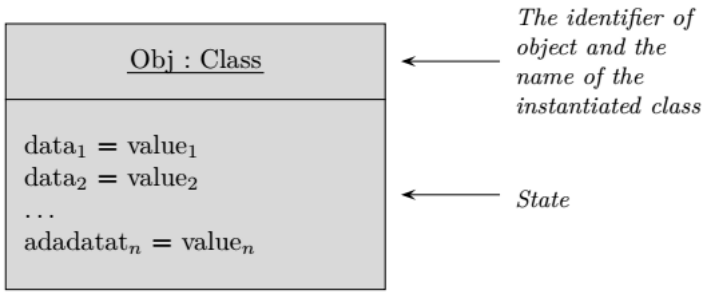


Figure 10.2: Object diagram

An object diagram holds the name of the class that we have instantiated this momentary state.

10.2.3 The representation of instantiation

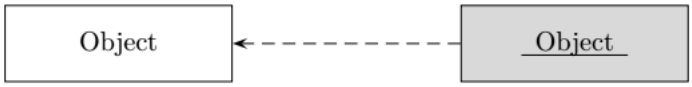


Figure 10.3: The instantiation

Figure 10.3. shows the instantiation of an object. If we instantiate more than one object we have to show the number of instances (Like on Figure 10.4.)

Figure 10.4: Instantiation of multiple objects

10.3 Constructing and destructing objects

The typical life cycle of objects: they are ”born”, they ”live” and they ”die”.

When we create abstract data types it is important for the system to set the type invariant of the concrete object, which means it should set an initial state that fulfills the type invariant. This is the function of the *constructor* of the object. This is when the initialization of the data members happens, and this is when we execute the necessary routines of the initialization.[3] The constructors of public classes are usually public as well, to let the program that uses the class to access the constructor. Sometimes we would like to ensure that only one instance of the class can exist at the same time, then we have to create a hidden constructor [Gam95].

Some of the languages have the paradigm of *destructor* too, which is called when the programmer or the system terminates an object. It's duty is to release the resources of the object: such as free the allocated memory cells, release files, close sockets, etc.

The different programming languages have different solutions for constructors and destructors.

C++

In C++, the constructor and the class have the same name. We can define multiple constructors for the same class due to the method overload capabilities.

```
class Complex {
    double re, im;
  public:
    Complex() { re = 0; im = 0; }
    Complex(double a, double b) { re = a; re = b; }
};

Complex z1;
Complex z2(1, 1);
```

The fields (*re* and *im*) of *z1* object are 0, while the fields (*re* and *im*) of *z2* are 1.

Freeing of object is done by the destructor. The name of the destructor is the name of the class with a ∼ (tilde) prefix.

In C++ the destructor is called automatically when we leave the scope of a static object or when we free the memory of a dynamic object. We cannot give any parameter to a destructor, to make automation possible.[4]

[3] It is the typical duty of constructor to bind the virtual methods of object to the actual methods in the Virtual Method Table (VMT) to tell what to call in the case of dynamic binding. See Section 10.7.2.

[4] The rules of constructing and destructing objects are far beyond the capabilities of this book. If you want to know more about this read Bjarne Stroustrup's book about C++ [Str00].

Object Pascal

In Object Pascal, we can create multiple constructors for a class with any kind of names and parameters. When we declare one, we have to use the *constructor* keyword. The name of constructor of *TObject* class (which is the super class of every class) is *Create*. It is usually a convention to use this name.

In Object Pascal, we have to call the constructor prior to the use of an object. Let's amend our previously mentioned Date class with a constructor. (More on: 10.1.1)

```
type
 TDate = class
   private
     Year, Month, Day: Integer;
   public
     constructor Create(y, m, d : Integer);
     ...
 end;
constructor TDate.Create(y, m, d : Integer);
...
```

If an object allocates resources then we have to free those resources in case we don't need them anymore. The *Destroy* virtual method is the default destructor, it is strongly advised to redefine this method to fulfill our requirements.

With the *Free* method we can avoid the method call on empty references, because *Free* calls *Destroy* only on existing objects.

```
procedure TObject.Free;
begin
  if Self <> nil then
    Destroy;
end;
```

We have to set a variable that pointed to the destroyed object to **nil** manually.

Java

In Java, the name of the constructor is the name of the class (just like in C++). A constructor can have multiple parameters, and cannot have a return value. It is automatically called on the creation of the object. The variables of the super class can be initialized by calling the constructor of the direct ancestor class, or by calling any constructor of the derived class. If we don't write any of them, then it will call the parameterless constructor of the ancestor. If a constructor calls the ancestor or any of its own constructors, this call must be the very first line of the constructor. We cannot access any of the instance variables before calling the constructor.

If a class does not have a constructor, then the compiler will generate the following:

```
class AClass extends BClass {
  AClass() {
    super();
  }
}
```

A use of the above:

```
AClass var = new AClass();
```

There is no such thing as destructor in Java. The allocated memory will be freed automatically. The Garbage Collector which checks and collects the unnecessary objects of the program and frees them on a timely basis runs concurrently with the Java Interpreter. We cannot define a destructor but we can define a *finalize()* method in every class. Unfortunately we cannot know or control when the *finalize()* method should run, we only know that it will run prior to the reusing the memory cells of the object.

The class level version of *finalize()* method is *classFinalize()* which is a class method. It is called before the class is deleted by the Garbage Collector.

Eiffel

In Eiffel, we can declare a constructor with the *creation* clause.

```
class COMPLEX
  creation
    assignment
  feature
    real, imaginary : REAL
  feature
    assignment(r, i : REAL) is
      do
        real := r; imaginary := i
      end;
end -- class COMPLEX
```

Due to the following statement a new object will be constructed and assigned to z with an initial value:

```
z : COMPLEX;
!!z.assignment(1,0);
```

In Eiffel, there is no explicit destructor (just like in Java). When an object is not referenced by the program, then an automatic garbage collector will destroy it.

Ada

In Ada, there are two predefined **tagged** types – the *Controlled* and the *Limited_Controlled* – which have a method called *Initialize*. In the subclasses of these types we can override the *Initialize* method to tell what to execute when we create the object. In these classes, the *Finalize* method will be called before freeing the memory cells of the object.

Python

Python generates an empty constructor for every class that allows us to create an empty instance of that class. This constructor can be changed if we define the *__init__(self)* method. This method can also be changed to require parameters.

```
class Complex(object):
    def setup(self, real, imag):
        self.r = real
        self.i = imag

    x = Complex()
```

This class can be created and it will hold no value on creation time.

```
class Complex(object):
    def __init__(self):
        self.r = 1
        self.i = 2

    x = Complex()
```

In this example X holds 1 and 2 in its variables.

```
class Complex(object):
    def __init__(self, real, imag):
        self.r = real
        self.i = imag

    x = Complex(3.0, 2.0)
```

If we define an initialization method that requires parameters we can set up an object in construction time.

Scala

We should not care about object destruction in Scala, it is done automatically, but we have to care about instance creation. This is done by the `new` operator, and the name of the class. (Just like in Java.) In Scala we can have multiple

constructors, but these constructors are just syntactic sugars, they have to call the default constructor first.

Constructor declaration is special in Scala. The default constructor is defined by the name of the class, and other constructors are defined by `this` keyword.

```
class Complex(val real : Double, val imag : Double) {
   var re : Double = real
   var im : Double = imag
   def this() = this(0.0, 0.0)
   def length : Double => Math.sqrt(re * re + imag * imag)
}
object Test {
   def main() {
      var c1 = new Complex(3.0, 4.0)
      var c2 = new Complex()
      println(c1.length())
      println(c2.length())
      c2.re = 3.0
      c2.im = 4.0
      println(c2.length())
   }
}
```

In the above example, `Complex` is defined with a constructor that requires two `Double` arguments. This is the default constructor of the class. We do not have to write a constructor body, because the effects of the constructor impact the whole class. Parameters of a constructor are available in all methods (we could call `re` and `imag` in the `length` method although `re` is a variable of the object, and `imag` is a parameter of the constructor) if they are immutable values. This causes that the above example code would print: `5.0` (as the length of `c1`), `0.0` (as the length of `c2`), and `3.0` because the setting of `c2.im` does not take into account in the calculation of `length`.

The second constructor of `Complex` is a parameter-less constructor that calls the default constructor of the class with `(0.0, 0.0)` parameters. We could have executed other commands in this constructor, and then we should have written the body of the constructor between curly braces. (We can introduce blocks with curly braces.)

```
class Complex(val real : Double, val imag : Double) {
   var re : Double = real
   var im : Double = imag
   def this() = {
      this(0.0, 0.0)
      this.re = 1.0
      this.im = 2.0 * this.re
   }
   def length : Double => Math.sqrt(re * re + imag * imag)
}
```

10.3.1 Instantiation and the concept of Self (this)

If we look at the definition of class and object we can see that the representation of an object in the memory is nothing else than the representation of a data structure. The class is a type definition pattern. We represent data and methods. During instantiation the object is created – with all of its data members – and it is waiting for instructions to execute statements. We connect the methods to the class and do not copy them to every object. This means in the description of the object that there are no methods, only in the description of the class. Every object knows what class it belongs to and it chooses the right methods by this information.

It is wise to ask: If we have multiple objects of the same class how do we know to which object should we apply the method? The method will use the data of which object? The programming languages have a solution for it. They have a specific reference to the actual calling object. This is the `Self` (in some languages `this` or `Current`) "argument". The `Self` variable clearly references the object that the function must use. That also means that if the method wants to send a message to itself it has to use the `Self.Message(Arg)` form. (We do not have to write it all out, in plenty of languages it is the default behavior.)

Python's philosophy is a little bit turned. It requires an explicit variable that will be the reference to `Self`. This is traditionally named as `self` but we can name it any way it will point to the actual object. This parameter is invisible when we call a method.

```
class Complex:
    def setup(self, real, imag):
       self.r = real
       self.i = imag

   x = Complex()
   x.setup(3.0, -4.5)
```

As we can see in this example the method was defined with three parameters but it was called with only two values. The third one will be explicitly passed

and it will be the object instance which was the base object of this method invocation.

10.4 Encapsulation

In a class the data used to describe the functionality and the methods that work on those data are strictly held together to have a better model of reality.

The *encapsulation* means that we consider the data structure and the methods over the data structure as one unit, and we hide them from the outer world. The state and the behavior of the object (the *how*) is a private matter of the object. The inner world of the object is untouchable. The object protects its data and does not allow anybody to do work with them only through the own methods of the object. The implementation of the methods is also hidden from the other objects of the System.

This approach – as we have earlier mentioned – is a return to the abstract data structures. The approach of data abstraction means that we do not describe the task but we use the available simple features and concepts to build more complex features and concepts until the model of the whole domain of the original problem becomes available for us. Then we solve the problem as we have observed it in reality. This approach fits the object-oriented concept. As we have seen in Chapter 9 the *specification* gives the (outer)description of the task, and the important attributes (*the value-set of the type*) to model the problem and the essential behavior (*methods*). The second layer is the computer *representation* and the inner behavior (*implementation*). This kind of description is not restricted to cover some objects of the reality, but should be done with everything with the same attributes. The sum of these similar objects with heir behavior description is called *type* or *class*. We can set pre– and post–conditions to the behavior schemes (methods) and we can say that object-oriented programming with the feature of encapsulation fulfills every aspects of abstract data types. (See Chapter 12.) A lot of object-oriented programming languages implement encapsulation with the extension of the concept of structure or record (**struct**, **record**) thus introduce the *class type* (**class**). There can be data members in a class type just like in the previous records and there also can be methods. This path was followed by the designers of C++ or Object Pascal.

In the programming languages, which were designed to be ”fully” object-oriented – just like Smalltalk, Java and Eiffel –, the concept of union type and variant record was omitted (See Chapter 6.) because *class type* and inheritance were sufficient.

In Ada the support of encapsulation was only possible to implement indirectly. They have expanded the record concept of Ada 83, but these records only held the attributes of the class (without the methods). Thus encapsulation was only implementable through **tagged** record and with collecting the methods to a module (**package**).

10.5 Data hiding, interfaces

The concept of encapsulation also means that the object hides its inner representation to avoid of ”being corrupted” by other objects (and inner mistakes cannot spread through the System). To make it possible to communicate with the world and keep the object as a closed unit we have to set some sort of filter to make only those parts visible which cannot cause any harm but they make possible to use the object. This filter is called *interface*.

This means encapsulation can be done with the following rules:

- An object only responses to a previously defined set of messages
- An object can only be touched through its interface.
- An object should have as small interface as possible.

The requirement of this methodology is to make data only accessible through methods. Some of the languages make it possible to reach data directly – we should not use this feature.

The *definition* of the interface is the responsibility of the programmer. This can be done by explicitly setting some of the methods to be accessible (in special cases even data members can be set accessible). These methods and data are *public*. The hidden members of the class are *private*. The private members of the object can only be accessed by the object, these are invisible, inaccessible for other parts of the System. In an ideal case, when some data are public this only means it can be read but cannot be written. This is the priority of the methods of the object. (This was implemented in Eiffel.)

In object-oriented programming languages, it is possible to separate public and private methods (mostly with the *public* and **private** keywords.) The rules of visibility is checked in compilation time and the breaking of such rule leads to a compile-time error.

In some programming languages beside private and public visibility they have defined different visibility levels. We will talk more about them in the exposition of protected data hiding mode.

Data hiding solutions of Smalltalk

In Smalltalk, the inner variables of the object can only be seen directly by the *object*'s methods. By this, Smalltalk hides the representation from the client. A client can only manipulate an object by sending a message which means that the changing of representation does not have to lead to the changing of the client. Objects with the same class cannot reach directly the members of each other – this is called *object-level* visibility.

Every method is *always* visible, so we can only propose not to call those methods directly which were marked to be **private**.

Every data is completely hidden and we cannot change them. If we want to reach a data we have to implement a method to read and write it. (The name of these functions can be the same as the name of the data member.) For example:

```
Object subclass NewClass
instanceVariableNames: 'anInt'
classVariableNames:''
poolDictionaries:''
...
anInt
  ^anInt.
...
anInt : a
  anInt:=a.
```

Access control in C++

In the class definition of C++ the members after the keyword `public:` are publicly accessible, and the members after `private:` are hidden towards the clients of the class. C++ has a *class-level* visibility rule, which means that objects of the same class can reach directly the private parts of each other. In case of `class` types the `private` visibility is the default.[5] The definition of public and private parts can be done in any order. This means that the following definitions are equivalent:

[5] In case of record types (`struct the default visibility is public for every method and data.`)

```
class A {
    int i;
  public:
    void set_i(int n) { i = n; };
    int get_i() { return i; };
};
class A1 {
  private:
    int i;
  public:
    void set_i(int n) { i = n; };
    int get_i() { return i; };
};
class A2 {
  public:
    void set_i(int n) { i = n; };
    int get_i() { return i; };
  private:
    int i;
};
```

It is recommended [Str00] to make all the data private or protected. (See Section 10.7.1).

Data hiding of Object Pascal

In Object Pascal, we mark the inner parts of the object with the keyword **private**, but they are only invisible for parts of the System that were defined outside of the compilation unit (the source file). We can only fulfill the data hiding concept if we implement every class to a separate file, or we use the keywords *strict* **private**.

As usual, the **public** keyword means that it can be accessed by anybody.

The default visibility is **published**. The published fields and methods are available in run time, but also available at design time. Every component of Delphi has a **published** interface, which are used by different Delphi tools, such as Object Inspector. Every field of a class which is not marked with other keyword the **published** will be prevalent.

The language makes it possible to define data fields **public** but it is better (as it is advised by Marco Cantù [Can00]) to use properties instead (see Section 10.1.1).

Accessibility categories of Java

In Java, we can use **public** and **private** keywords to mark public and private parts of the class.

The default visibility – when we do not set any visibility – is the so called ”package-private” visibility which means that the member of the class can be accessed from the classes with the same package [Nyek08].

In Java, there is class-level visibility, which means that objects of the same class can access the private members of each other.

Selective visibility of Eiffel

The designers of Eiffel have implemented a rarely used technique: the *selective* visibility. This means that *every attribute of a class* (**feature**) can be set a visibility property that tells which classes can access it. In Eiffel, there is object-level visibility, which means that if we want other object of the class to reach a member of an object of their own class we have to explicitly set this property.

We can set our visibility properties by listing the name of all classes which should reach the feature after the **feature** keyword. We can use two special keyword here: one is the default – which means we can omit it – *ANY* (available for everybody), and the other is *NONE* (not available outside the object.)

```
class C
  feature {ANY}
    x : T; -- every class can see it.
  feature {C}
    y: U; -- instances of class C can see it
  feature {NONE}
    z : V; -- only this object can access it
end -- class C
```

10.5.1 Friend methods and classes

Encapsulation determines a very important attribute (private members of the class can only be reached through the public methods of the class). We can only communicate with the object through the public interface. That is the way object-oriented approach lays down the visibility settings and data protection disciplines.

In C++, they have made a compromise that – because it was unavoidable even if it does not fit to object-oriented concept – it is possible to let instances of another class to access the private members of an object. This made the implementation of the acquaintance relations possible.

These procedures or methods are called **friend** procedures or methods. These methods are not messages to the object, which means they will not get the object through *self* we have to pass the actual object as an argument to them. A class can declare a whole class as a friend. In this case the methods of the friend class can reach the private members of the other class.

We can use the following declaration solution to declare friend relation:

- Procedure: **friend** *procedureName(argumentList);*
  (Every procedure with the given name can access private members.)
- To another method: **friend** *ClassName.methodName(argumentList);*
- To an other class: **friend** *ClassName;*

C++ example

In C++ [Str00] it is possible to redefine the operators. This means we can declare a *Complex* class and redefine the basic arithmetic operators.

We require from the class to add two complex numbers, or to add a real number to a complex number *regardless the order of the numbers.*

If *a* and *b* are instances of the class *Complex*, and we say *a + b* then the *+* method of *a* will be called with a *b* parameter. A similar result will be received in the case of *a + 2.5*, but the argument will be *2.5*. It is hard to interpret the *2.5 + a* expression, because *2.5* is of type **double** and its *+* operator cannot be called with a *Complex* as argument. A possible solution to use the **friend** keyword for the *+* operator:

```
class Complex {
      double re, im;
    public:
      Complex(double r = 0, double i = 0) { re = r; im = i; };
      Complex operator+(Complex p);
      Complex operator+(double p);
      friend Complex operator+(double, Complex);
};
Complex Complex::operator+(Complex p){ . . . };
Complex Complex::operator+(double p) { // The class method
    Complex temp;
    temp.re = re + p; temp.im = im;
    return (temp);
};
Complex operator+(double p1, Complex p2) { // The friend function
    Complex temp;
    temp.re = p1+p2.re; temp.im = p2.im;
    return(temp);
}
int main() {
    Complex c1,c2;
    c1 = c2 + 2.5;   // The class method.
    c2 = 2.5 + c1;   // The friend function.
    return 0;
}
```

10.5.2 The private notation of Python

In Python, the language designers have decided to make every part of the class public. This means that we cannot make any method or variable private, however it is a common notation in the Python community to start every private part with one underscore (_). This is a very weak implementation of the private concept but this is quite popular in the community. This implementation makes easy to implement debuggers, and development environments still can be set to warn programmers when they access a private part outside from the object.

```
class Foo(object):
   def __init__(self, bar):
      self._bar=bar;
      self._check_bar();

   def validate(self) #This method is considered public
      return true;

   def _check_bar() #This method is considered private
      return hasattr(self, bar);

f = Foo(new Bar());

f.validate();
f._check_bar();     #Both method calls are considered valid.
```

10.5.3 Visibility Rules of Scala

Scala has a very fine tune visibility rule set. The default visibility in Scala is ”public”, however there is no `public` keyword in the language. It is only an implementation detail, which have come with the JVM platform, but we have to mention: to make mutable and immutable fields on JVM platform Scala had to apply some tricks: immutable value fields (defined with the `val` keyword) will be hidden under a ”private” (more on this later) field, and a public reader function will be provided for the member methods and the outer words. Mutable variables (those which are defined with the `var` keyword) are handled similarly but a writer method is generated as well.

Beside these Scala provides us with plenty of opportunities. We have the ”standard Java visibilities” but in an other way. As we have mentioned if we do not mark the class, field or method with any keyword then it is considered as public. If we mark a member with `private` it will be visible for the instance and other instances of the same type (just like in Java.) If we mark the member as `protected` then the instance and the descendants of the instance and other objects with the same type (or derived type) can access it.

A class can also be ”package-protected” which means it can be extended by classes inside the same package (equally for public, protected and private classes of the package) but cannot be extended by classes from other package.

But Scala also has `private['X']` and `protected['X']` visibilities where we can give the scope of the visibility (These are called *scoped private* and *scoped protected*). The scope of the visibility can be a package, a class and an instance. What does it mean? Consider the following hierarchy:

```
package Pk1 {

  class Cl1 {
    val field : Int
    def method = { println(field) }
  }

  class Cl2 extends Cl1
}

package pk2 {
  class Cl3

  class Cl4 extends Pk1.Cl1
}
```

In this hierarchy we have two packages (`Pk1` and `Pk2`), and four classes (`Pk1.Cl1`, `Pk1.Cl2`, `Pk2.Cl3`, `Pk2.Cl4`). `Pk1.Cl1` has two members: a field (`field`) and a method (`method`). Both `Pk1.Cl2` and `Pk2.Cl4` are derived from `Pk1.Cl1`.

What scoped visibility rules could have been applied to them, and what would that mean? The rule `private[Pk1]` before `class Cl1` would mean that this class is private in this package, and it cannot be referenced from outside of the package this rule is so strict that we cannot even derive a class from it (like `Cl2`) if it is not set to be private in the package (We could not even define `Cl4` this way because it is in an other package.)

If we write `private[Pk1]` in front of the members of `Cl1` it means that the instances of `Cl1` and `Cl2` can access them but `Cl2` would not inherit them.

The rule `private[Cl1]` in front of `field` or `method` would result a private member which is accessible for other instances of `Cl1`. (This is just like common private.)

The rule `private[this]` in front of members of `Cl1` would mean that they are visible only for the instance, and other `Cl1` instances could not access them. (This is how the constructor values are stored.)

`private[this]` could be also be written in front of a class (like `Cl1`) which would bound it to the enclosing package. (The same package-private rule that we have seen in case of `private[Pk1]`.)

We can write `protected[Pk1]` in front of class members of `Cl1` which means they behave as protected variables in the package and as private variables outside the package. (Instances of a class in the same package as `Cl1` which is not in an inheritance hierarchy with `Cl1` – there is not any in this example – could not access the protected members.)

`protected[Cl1]` and `protected[this]` in front of any member variables will mean the same as a simple `protected` rule.

10.6 Class data, class method

We have considered that the class is a declarative matter, and objects are concrete instances of those. We work with the concrete objects and not with the classes. If we need a concrete object, then we instantiate one, and we communicate through the interface by sending messages to the object to reach a specific goal.

It is clear that the object holds the data, and the temporary values of the variables declare the concrete state of the object. Thus we call these data as *member variables* or *instance variables*. The methods affect on a specific instance and we can call these methods by a `Object.Method(Argument_list)` message call and that is why we call these methods *instance methods*.

The function of the class is to define the type and the procedure to instantiate them, to allocate the memory for the instance variables and to pass the reference through the `Self` parameter to tell the methods what object to use.

In some programming language – like in Smalltalk – the designers of the language stated that everything is an object and that means classes are objects as well and can have an inner state (attributes) and methods.

It is also true that every object is an instance of a class. It raises the question: what is the class of these class objects? In Smalltalk every class has a generated "metaclass" which has only one possible instance which is the class. We can put the question what is the class of these special metaclasses? These metaclasses are the instances of a special class which was predefined for only this purpose and is called `Metaclass`.

A class as an object might have an inner state which is called *class data*. These variables hold data of the class and they are not necessarily changed by the state changing of instances. They are allocated during the allocation of the class and they are freed with the freeing of the class. In some languages – like C++, Java or C# – these class variables are called *static* and are marked with the keyword `static`. These data are accessible by the methods of instances. There is always only one instance of a class variable which is shared among the instances of the class. (For example a class can count its instances in a class variable.)

There can be special methods which do not affect the data of the object but on the data of the class. These methods are the class methods. Of course

a class method cannot use the data of an instance because a class method does not know anything about the instances. There is no `Self` (`this`) argument in the case of a class method because it has no use to bind a class method to a concrete instance. A class method can be called without any existing instance of the class. For example the retirement age can be a class variable of the `Employee` class which means it can be changed without any existing instance.

Class methods regularly can be called through the name of the class, i.e.: `ClassName.MethodName(Arguments)` or `ClassName::MethodName(Arguments)`, but usually we can access them through the instances. If we do not mark the class when we call the class method then the method of the actual class will be called.

The class variables get their values only once at the initialization time of the class, in order of their appearance.

Smalltalk

As we can see in example 10.1.1 when we introduce a new class:

```
instanceVariableNames:...
```

and the

```
classVariableNames:...
```

closes are strictly following each other, there we can set the name of the class and instance variables of the class.

We can define class and instance methods.[6] The system differentiates them by the `class` keyword written after the class name.

⟨*classname*⟩ `class methods`
  ⟨*methodname_1*⟩ ⟨*methodbody_1*⟩
  ⟨*methodname_2*⟩ ⟨*methodbody_2*⟩
  ...
⟨*classname*⟩ `methods`
  ⟨*methodname_1*⟩ ⟨*methodbody_1*⟩
  ⟨*methodname_2*⟩ ⟨*methodbody_2*⟩
  ...

Every object is instantiated by a message sent to the *class*.

C++, Java, C#

In C++, Java and C# class methods and variables are declared with the **static** keyword. They can be used as usual.

The following example shows the declaration and use of class methods and variables in C++ [Str00]. The *Date* class – and not the instances – has and sets

[6] In the Smalltalk development environment there is a separated menu to introduce them.

default *Date* attribute. We can use this during the instantiation of the objects as a default value:

```
class Date {
    int year, month, day;
    static Date default_date;
public:
    Date(int yy=0, int mm=0, int dd=0);
    //...
    static void set_default(int,int,int);
};
Date::Date(int yy, int mm, int dd) {
    // If we call the default constructor then we set the
    // default_date value to the new object.
    year  = yy ? yy : default_date.year;
    month = mm ? mm : default_date.month;
    day   = dd ? dd : default_date.day;
};
void Date::set_default(int y, int m, int d) {
  //Changing the values of the static variable.
  Date::default_date = Date(y, m, d);
}
// Defining the static variables.
Date Date::default_date(2003, 3, 16);
```

Object Pascal

In Object Pascal if we want to use a method as a class method we can mark it in the definition of the class with the **class** keyword [Can00]:

```
type
  A = class
    procedure B;
    class function ObjNum :Integer;
  end;
```

Here the *B* is an instance method, and *ObjNum* is a method of class *A*. It is good to know that there is no keyword (currently) to create class variables. We can simulate this behavior if we declare a hidden variable in the implementation part of definition **unit** which will be only present once and can be accessed by all the methods of the object and the class methods. This variable can have a default value:

```
implementation
 var
   ObjNo : Integer = 0;
 class function A.ObjNum :Integer;
  begin
    Result := ObjNo;
 end;
```

In the body of *ObjNum* class method we use the *ObjNo* hidden variable.

Python

Class is an object in Python. It can have member methods and variables. If we start a member at least with two underscore (__) and finish it with at most one underscore (so like *__foo* or *__bar_*) then it will be considered as a class variable that will be textually replaced to *_classname__foo* (or *__classname__bar_*) where *classname* is the name of the class in which we have defined the variable.

There is not any special place in the class body where we should define our class members, every member which was defined this way inside the class definition will be considered as a class member.

```
class Foo(object):
    __bar_ = 42;

print _Foo__bar_; # Prints 42 to the screen.
```

Scala

There is no such concept in Scala like class method or class data. Instead, singletons can be created with the `object` keyword. These objects are instantiated by definition and they can not be instantiated any more. These singletons can replace the static functionalities of other object-oriented languages.

```
object ThisIsASingleton {
    val quasiClassValue = 1
    var quasiClassVariable:Int
    def quasiClassMethod(x:Int) = 2 * x
}
object Test {
    def main() {
        ThisIsASingleton.quasiClassVariable=2
        println(ThisIsASingleton.quasiClassValue) //1
        println(ThisIsASingleton.quasiClassVariable) //2
        println(ThisIsASingleton.quasiClassMethod(3)) //6
    }
}
```

10.6.1 Class diagrams

In case of class diagrams we note if a variable is a class variable or a method is a class method, which can be done with a `C` character (which is an abbreviation of *class*) printed in front of these attributes. The `+` sign means the public the `-` sign means the hidden visibility. (See on Figure 10.5.)

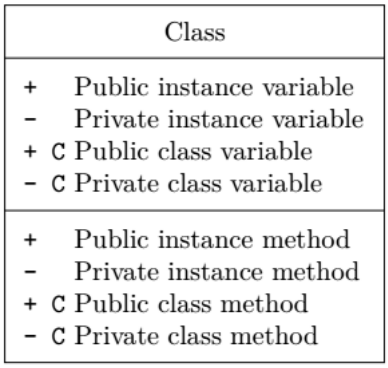


Figure 10.5: Class variables and class methods

We call instance variables as *data members* or *variables* and instance methods as *methods* and we will only note class variables and class methods.

10.7 Inheritance

In the object-oriented approach – as we have seen – we describe the world with *classes*. For example we differentiate living things from lifeless objects. The next logical step is to introduce sub-classes into classes – like we can group living creatures as the animal taxonomy: if we come across a new animal we can describe it by deciding if it is a mollusc, an arthropod, a vertebrate, etc. If we find it is an arthropod we decide whether it is a crustacean, an insect or an arachnid. If it is an insect we examine if it is winged or apterous. Winged insects have a lot of subclasses like: butterflies, flies, bugs, etc.

Every subcategory can have a great number of subcategories which can have more subcategories. Every level is much more specific, than the level above it, but it is also true that we can introduce an attribute in one level, and this will be held by the subclasses below it. This means if we have discovered that a creature is a butterfly, then we do not have to tell that it is an arthropod, it is obvious because butterflies are anthropoids.

In the case above we can describe what the attributes of a class are, and what are its sub-classes ([Mey00], [Ada95] and [Bor89]).

When we introduce derived classes we always reckon the attributes of the super class. We also use the term: *the derived class inherits the attributes of the*

super class. It is possible that the super class is a derived class, thus some of its attributes were inherited. This means we can describe the derived classes simpler because we do not have to note all inherited attributes, we can just reuse the already defined classes.

Then the new derived class (or subclass) is the *heir* of the super class, and the base class is the *parent* class. So we imagine inheritance as a subclass forming operation. The classes can be organized into an inheritance hierarchy.

Thus the *Student* class is a subclass of the *Person* class then the instances of *Student* are instances of *Person*. That means if we declare a variable with the type *Student* we can handle it as a *Person* typed object. This is not true backwards: a *Person* does not have to be a *Student*. This feature that a variable can be handled as an instance of more than one class is called *polymorphism*. (See Section 10.7.2.)

This means we need modules which are open and closed at the same time. They are closed, because they alone describe a class with their well-defined methods, but they are also open which makes it possible to derive specialized classes from them, this way we do not have to describe everything again just use them. The closed modules which were described with interfaces allow new modules to use them easily and build their services upon them. If we can use them as open modules we can inherit from them.

Specialization is usually reached by introducing new features to the class. For example we can inherit *Human* from *Creature*, then we can extend the attributes of *Creature* with *intelligence* or *nativeLangue*, but at the same time we shrink the group of possible instances to the type set of *Human*. The type set of *Human* will be a subset of the type *Creature*.

We should make it possible to introduce new methods in subclasses, that would not fit the super class. A typical example for this is the *ring* of algebra which was introduced by adding one more possible operation to *group*. A group has one operation (let's say the +) with some attributes and ring built over this group will have this operation also with the same attributes, but would also have a new operation (like *) with new attributes [Mey00].

When we introduce new subtypes we should make change possible. It means that some methods should be possible to be redefined: their implementation or their specification can be changed. For example the class of *Shape* can have a *paint* method, which is used to paint a general shape to the screen. It is a logical to expect that a subclass of *Shape* like *Circle* or *Polygon* the *paint* method should paint a circle or a polygon to the screen. *Dynamic binding* (or *dynamic dispatch*) is responsible for linking – in run time – the actual type of the variable to the corresponding implementation. The substitution of methods cannot be optional as we will see later. This approach is called *specialization inheritance* or *specification inheritance* [SV01].

An existing class can be used to define new classes to reuse existing code. This is possible if in the implementation of the derived class the use of the

parent class is hidden. This is called *reuse inheritance* [SV01], or *implementation inheritance* [Mey00]. In design it is better to use a hidden inner instance.

Inheritance can be *single* or *multiple*. At *single inheritance* the derived class can only have *exactly one* direct super class, in the case of *multiple inheritance*, *at least two* direct super classes. The derived class *always* inherits all the methods and attributes of the super class. Multiple inheritance can be problematic when two direct parent classes have one attribute or method with the same name, because it will lead to ambiguity in the derived class – see Section 10.7.5.

Inheritance in SIMULA 67

SIMULA 67 was the first language that made it possible to use the features of inheritance. If we prefix a class definition (or a block) with an existing class name then we can get the attributes and methods of the prefixing class (the parent class). These could be extended or redefined. This means single inheritance.

For example we have a *Point* class, where we define the points with their coordinates, and we can *print* or *shift* them. A derived class of this can be defined as *Colored_Point* to also hold and print color information.

```
Class Point(X, Y); Real X,Y; ! coordinates;
 Begin
   Procedure Print;
   Begin
     OutText("Point: ");
     OutFix(x, 1, 4); OutText(","); OutFix(y, 1, 4);
   End of Print;
   Procedure Shift(Dx, Dy);  ! Dx, Dy values of shifting;
   Real Dx,Dy;
   Begin
     X := X + Dx; Y := Y + Dy;
   End of Shift;
   Print;                          ! The life cycle of a point;
   OutText(" instantiated"); OutImage
 End of Point;


Point Class Colored_Point(C); Character C;
        ! We use Point to declare Colored_Point. ;
        ! We declare one more attribute. The life cycle changes. ;
 Begin
   ! The life cycle of Colored_Point follows the life cycle of Point: ;
   OutText("Color "); OutChar(C); OutImage
 End of Colored_Point;
```

Ref(Colored_Point) A;
Ref(Point) C;
C :– New Point(5, 6);
A :– New Colored_Point(3, 4, 'G');

After the previous statements the followings would appear on the screen:

```
Point:  5.0, 6.0 instantiated
Point:  3.0, 4.0 instantiated
Color G
```

In the implementation of inheritance there is a special opportunity for the use of the *inner* keyword. The position of this in the program code defines when to call the code of the derived class. In every derived class first the code of the super class will be executed and in the end of that block or, when an *inner* will be reached only then would the code of the derived class executed.

With the use of the prefixes we can build a *prefix chain*. If $C_1, C_2, \ldots, C_n$ are classes and C_1 does not have a prefix, and the prefix of C_k is $C_k - 1$ (where $k = 2, \ldots, n$) then $C_1, C_2, \ldots, C_{k-1}$ classes are the prefix chain of C_k. No class can appear in its own prefix chain.

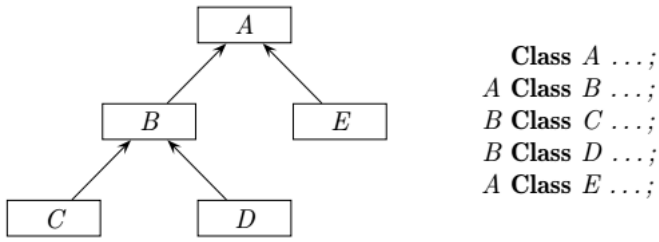


Figure 10.6: Prefix chain in SIMULA 67

Inheritance in Smalltalk

In Smalltalk inheritance was implemented by sending the `subclass:` message to the parent class (as we have seen in Section 10.1.1). In the following example the `Bird` class is a derived class of `Animal` [Sma91].

```
Object subclass: #Animal
  instanceVariableNames:
    'name ability behavior maxSpeed color '
  classVariableNames: ''
  poolDictionaries: '' !
...
 Animal subclass: #Bird
  instanceVariableNames: 'flying '
  classVariableNames: ''
  poolDictionaries: '' !
```

Inheritance example of C++

We introduce the *width* and *height* attributes to the *Shape* class and methods to set them, and derive the *Triangle* and *Rectangle* classes.

```
class Shape {
  protected:
    int width, height;
  public:
    void set_values(int a, int b) { width = a; height = b; }
};
class Rectangle: public Shape {
  public:
    int area() { return (width * height); }
};
class Triangle: public Shape {
  public:
    int area() { return (width * height / 2); }
};
```

The *Triangle* and *Rectangle* classes have the attributes and methods of *Shape*. The *area* method is new in both of them, and it is defined differently. This also raises a design question we will cover later in Sections 10.7.2 and 10.7.4. C++ also allows multiple inheritance. For example if we add the *EqualSided* class, then we can derive *Square* from this and the *Rectangle* classes:

```
class EqualSided {
public:
     bool e;
};
class Square : public EqualSided, public Rectangle {
     Square() { e = true; };
};
```

Inheritance in Object Pascal

Object Pascal supports single inheritance, and interfaces are used to implement multiple inheritance. The syntax is represented in the following example.

```
type
  TAnimal = class
  public
    function eat : string;
  private
    kind : string;
  end;
```

```
  TDog = class (TAnimal)   { the heir of TAnimal . }
  public
    function bark : string;
  end;
var
  Dog1 : TDog;
begin
  { ... }
  writeln(Dog1.eat);
  writeln(Dog1.bark);
  { ... }
end.
```

TDog is derived from *TAnimal*, and introduces the *bark* method.

Inheritance in Eiffel

In Eiffel, the inheritance relations can be defined after the **inherit** keyword. A derived class can access all the data and methods of the super class, and it can also change them during the inheritance process. After the **inherit** key word we have to list the classes that we want to be derived from, and we have to set in the **redefine** clause the attributes that we want to change. The features that were not mentioned in any **redefine** class will be inherited without a simple change. The language is capable of multiple inheritance which will be discussed in Section 10.7.5.

```
class POLYGON inherit
   SHAPE
 feature
   print is
     do
      io.putstring ("I am a polygon");
    end;
end
class SQUARE inherit
   POLYGON
     redefine print
     end
 feature
   print is
     do
      io.putstring ("I am a square");
      io.new_line;
    end;
end
```

Inheritance in Java

In Java, there is a single inheritance. The *extends* keyword shows us that it was designed as the main feature for sub-type relationships. The following *Boss* class is derived from the *Employee* class from Section 10.1.1:

```
public class Boss extends Employee {
  final int MAXEMPLOYEE = 20;
  Employee[] employees = new Employee[MAXBEOSZT];
  int numOfEmployees = 0;
  public void newEmployee(Employee e) {
    employees[numOfEmployees++] = e;
  }
}
```

In Java, we can declare a class which cannot have any subclasses. For this we have to use the **final** keyword.

In Java, the feature of multiple inheritance is implemented by interfaces. (See Section 10.7.6.)

Inheritance in C#

In C#, inheritance – "officially", and in syntax is similar to C++ – works like Java: There can be a single inheritance between classes, and we need interfaces for multiple inheritance. We can forbid the forthcoming derivation by the **sealed** keyword.

```
class SuperClass { . . . }
class Derived: SuperClass { . . . }
```

Inheritance in Ada

In Ada the *tagged* records can have derivatives. We can derive a new class from the class in Section 10.1.1. in the following way:

```
with Person_Type; use Person_Type;
package Student_Type is
   type Student is new Person with
      record
         Average_Credit : Float;
         Level          : Integer;
      end record;
   procedure Show (P : in Student );
end Student_Type;
```

In the example the *Student* class was extended with two new attributes to the *Person* class. It also inherits the *Person* type methods to set and get name, address and age, and redefines the *Show* method.

We can instantiate objects the following way:

```
P1      : Person;
S1      : Student;
Gabriel : Person;
Gabe    : Student;
begin
   Write_Name(Gabriel,"Gabriel");
   Write_Address(Gabriel,"Budapest" );
   Write_Age(Gabriel,35);

   Write_Name(Gabe,"Gabe");
   Write_Address(Gabe,"Pomaz" );
   Write_Age(Gabe,20);
   Gabe.Average_Credit:=4.5;
   Gabe.Level:=3;

   Show(Gabriel);
   Show(Gabe);

   P1 := Person(Gabe);
   S1 := (Gabriel with 3.05, 2);
end;
```

The hidden parts can only be reached through the setter methods, and the public parts can be set directly. The *Show(Gabriel);* will call the method of *Person*, and the *Show(Gabe)* will call the redefined method of *Student*. In Ada we can only set a derived value to a super class by indicating of conversion, and in case of *P1 := Person(Gabe)* the average and level attributes of *Gabe* will be lost.

It is an interesting solution that in the conversion of the super type to a derived type (*S1 := (Gabriel* **with** *3.05, 2)*) we can set the two new values directly, and this way we can ”extend” the object.

Inheritance in Python

Python also provides inheritance in its Object implementation. We can simply derive a class from a base class if we give the name of the base class in parentheses after the class's name. If we derive a class from an other class, then we will get a new dictionary that allows us to register new names (names of methods and member variables), and we will have a reference to the dictionary of the base class, which holds the definitions of the parent class.

The reference to the super class can be checked from the outside. This is usually done by the *isinstance()* method. This method can check whether the object is an instance of the given type or an instance of a derived class of the given class.

```
class Person(object):
   def __init__(self, name):
      self.name = name;
class Teacher(object):
   def __init__(self, name, course):
      Person.__init__(self, name);
      self.course = course;
   def add_mark_to_student(self, student, mark);
      student.add_mark(self.course, mark);
class Student(object):
   def __init__(self, name):
      self.marks = {};
      super(Student, self).__init__(names);
   def add_mark(self, course, mark):
      self.marks[course]=mark
s = Student('Peter');
t = Teacher('Mr. Johnson', 'Mathemathics');
print s.name;                              # Prints Peter
print t.name;                              # Prints Mr. Johnson
t.add_mark_to_student(s, 5);
print isinstance(s, Person);              # Prints true
print isinstance(s, Student);              # Prints true
print isinstance(s, Teacher);              # Prints false
```

Inheritance in Scala

Scala supports single inheritance, and is really similar to Java. It uses the keyword `extends` to derive a class from an other. It also uses `extends` when it only inherits from a trait, and it uses `with` keyword when it lists other traits in the inheritance list. It is possible to forbid further inheritance from a class, and we can also make it compulsory to extend one.

```
package pack {
  class BaseClass {
    val field : Int
    def method = { println(field) }
  }
  class DerivedClass extends BaseClass
}
```

Of course we have to call the constructor of the parent class if it had not got a parameter less constructor.

```
package pack {
  class BaseClass (val field:Int) {
    val field : Int
    def method = { println(field) }
  }
  class ParameterLessDerivedClass
        extends BaseClass(0) // Some integer value
  class ParameteredDerivedClass(val theValue:Int)
        extends BaseClass(theValue)
}
```

10.7.1 Data hiding and inheritance

Beside the (**public**) and (`private`) data hiding solution there is a third one in many object-oriented programming languages, which provides accessibility through inheritance. This data hiding solution is the `protected` mode. In protected mode the attributes are so unreachable for the world as they were `private` and they are accessible for the derived classes as they were `public`.

	outer world	**the object**	**the heir**
public	accessible	accessible	accessible
protected	not accessible	accessible	accessible
private	not accessible	accessible	not directly accessible

Table 10.1: The three different visibility modes

We have put together the effect of the three data hiding solution in Table 10.1. On Class diagrams these are marked as on Figure 10.7. The # sign shows the protected (only visible for derived classes) attributes and methods.

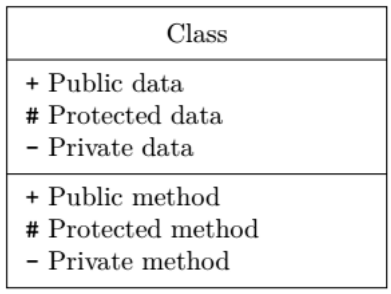


Figure 10.7: Public, protected and private data and method

In an inheritance hierarchy the data hiding principles work in the same way, a descendant class cannot see the private parts of a parent class. More precisely: A derived class technically inherits all the methods and variables of the parent class but cannot access them directly: they are inherited *invisibly* or *transparently*.[7] All other members and methods will be accessible directly.

Suppose that class `A` has a public `m` method, and a private `i` member variable, and `m` uses variable `i`. If the class `B` is derived from `A` then `B` will have `m`, which can use `i`, but `B` (or the new methods of `B`) cannot access or see `i` directly. The `private` members will be inherited as well, but they won't be seen. This is shown on Figure 10.8:

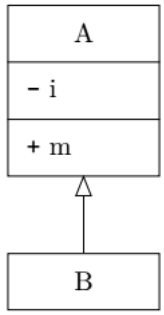


Figure 10.8: Inheritance with data hiding

A lot of programming languages allow to change the data hiding level of the inherited members: if a class inherit a *protected* variable, it can change it to be *public*, and from then it will be public for everyone.

Different kinds of inheritance

When we declare inheritance in some programming languages – like in C++ – we can modify the inheritance modes. This way we can override the inheritance modes of the base class, or we can brake some inconveniences of multiple inheritance.

If we use hidden (private) inheritance mode, we can inherit implementation, and we can hide (even from derived classes) that we have inherited from an other class. The public inheritance mode – which is the default in most of the programming languages – helps to implement specification inheritance.

Inheritance modes of C++

In C++ we have three inheritance (visibility) mode: **private**, **public**, **protected**, and **private** is the default. These inheritance modes will modify the visibilities.

[7] In the case of some languages the rules of private data can differ. For example in Object Pascal these are only true in case of different Units

Data hiding in base class	Inheritance mode or visibility mode	Visibility mode in derived class
`private`	`public`	cannot be accessed directly
`protected`	`public`	`protected`
`public`	`public`	`public`
`private`	`private`	cannot be accessed directly
`protected`	`private`	`private`
`public`	`private`	`private`
`private`	`protected`	cannot be accessed directly
`protected`	`protected`	`protected`
`public`	`protected`	`protected`

Table 10.2: Inheritance modes of C++, and visibilities.

We can see that **private** members will stay private and won't be accessible directly from the derived class. In the case of **public** inheritance the heir class will keep the specified visibilities. In **private** inheritance all the methods and variables of base class will be **private** in the derived class, which means it will be hidden during the next inheritance. In case of **protected** inheritance we only hide inheritance information from the outer world, and the derived classes can use the inherited members.

Inheritance modes in Eiffel

In Eiffel, the derived class can change the visibilities of the base class freely. It can publish a method which was only visible inside of the class and vice-versa. This can be done with the *New_exports* inheritance clause. The default is the following: if a derived class does not change something, then it is to be handled that way.

```
class FIXED_STACK[T]
  inherit
    STACK[T]
      -- ...
    ARRAY[T]
      rename
        put as put_array,
        element as array_element
      export
        {NONE} all;
      end
end
```

The *FIXED_STACK* will inherit its behavior from class *STACK*, and we do not have to change the visibility. The implementation comes from *ARRAY*

class, and later we do not want to make possible to handle the objects of the *FIXED_STACK* through array methods, so we hide all the inherited attributes of *ARRAY*.

10.7.2 Polymorphism and dynamic dispatching

In case of specification inheritance we can speak about *substitutability*. Substitutability means that the instances of a super class can always be replaced (substituted) with the instances of a its derived class. This also means that the derived classes can fulfill the roles of the super class, and can mimic its behavior, and sometimes we cannot differentiate them from each other.

This is trivial, because the derived class holds all the methods and variables of the super class, and can always reach them. This also means that in every case when the super class stands as a formal parameter the instance of the derived class can be an actual parameter. Substitution lies on the **is-a** relationship between the base and the derived class. This also shows that there is a type *narrowing*, a *specialization* relationship that points from the base class to the derived class, and there is a *generalization* relationship that points backwards.

As we have seen the specification inheritance is a way of sub-typing. This means that if *Circle* was derived from *Shape*, then every instance of *Circle* is an instance of *Shape*. But if we take a variable called *circleInst* which is an instance of *Circle*, then we can get a *shapeInst* variable with *Shape* type, and assign the *circleInst* variable to *shapeInst* variable. (This means that the *shapeInst := circleInst* assignment is valid.)
But an instance of *Shape* does not have to be an instance of *Circle*. This means that the *circleInst := shapeInst* assignment does not have to be valid, and that is why it is forbidden. This relationship shows some problem in the traditional type systems, because the *circleInst* is a *Circle*, and the *shapeInst* was declared *Shape*, which means that the *shapeInst := circleInst* should lead to type mix-up.

A solution for the problem is to take every variable as a typeless object, and we allow every assignment.[8] A better and more common solution for this is to introduce the concept of *static type* and *dynamic type*. The static type of a variable is the class that was used to declare the variable. The dynamic type of the variable – which shows the actual type of the object associated to it in run time – can be the static type, or any derived type of the static type. In the example above the static type of *shapeInst* is *Shape*, and the dynamic type is *Circle*. That is why we represent objects – in pure object-oriented languages – as implicit references.
The possibility that a variable can reference instances of more than one class is called *polymorphism of variables*.

There are different kinds of polymorphism, see Chapter 11. In object-oriented programming the polymorphism is only subtype polymorphism.

[8] Smalltalk have chosen this solution.

It is possible to change implementation or specification of some methods during polymorphism. As we have mentioned earlier if we have a *paint* method over the class *Shape* which is used to paint a general shape to the screen, then we can change this method in the derived *Circle* or *Rectangle* class to paint a circle or a rectangle.

Consider the following case: we override the *paint* method in class *Circle*, which was derived from *Shape*, and executed a *shapeInst := circleInst* assignment. After this we call the *paint* method on *shapeInst* object. Which method should be executed? Should we execute the *paint* method of *Shape* (the static type), or should we execute the *paint* method of *Circle*, which is the current dynamic type?
Of course we expect to execute the more specific method, which (in this case) means to execute the *paint* method of *Circle*. *Dynamic dispatching* (or *dynamic binding*) makes it possible to execute the methods of the dynamic type. Dynamic dispatching is a run time event. Several programming languages call the methods that can be dynamically dispatched as *virtual methods*.

We find the same problem in case of the *shift* method of *Shape*. If we implement the *shift* method in *Shape*, as a sequence of *hide*, *move*, *paint*, and we override the *paint* and *hide* methods in *Circle*[9] to fit the concept of circle painting. What should happen when we call the *move* method on the *circleInst* object?

The method will call the *hide* method, but which one? Should it call the method from *Shape*, or from *Circle*? Dynamic binding makes it possible to call the method of the dynamic type (the *hide* from *Circle*). Then it will call *move* from *Shape*, and after that the *paint* from *Circle* (because of dynamic binding again).

Redefining methods in derived classes

Let's take a look at how we can redefine methods in the derived classes. Should we force any limitations, if we want to keep the power of substitution?

It is important to differentiate between *overloading* a method name and *overriding* a virtual method. The overloading means (see Chapter 7, Section 7.6) that at one point of the program we have more definition for the same method name. Usually we have to make the overloaded methods distinguishable on call – it is simple if we have different number of parameters. We will see that distinguishing by the type of the parameters is not so trivial in some cases.

Static and dynamic redefine

In several programming languages – like Pascal or C++ – the methods of objects are statically bounded. This means if we find an *object.message()* statement in

[9] The *move* only changes the coordinates of the shape.

the source code then it can be replaced by a pointer to method called *message* and is defined in the class which is specified by the static type of the object. For example in the case of *shapeInst.paint* the *paint* from *Shape* will be called and in the case of *circleInst.paint* the method from *Circle* will be called.
Static redefining does not support the dynamic dispatching of subtype polymorphism which is a very important part of object-oriented programming. In case of dynamic dispatching the method that should be called de facto will be chosen in run time, when the dynamic type of the object will be indubitable. If a variable can hold a reference to any derived class then it is not determinable in compile time which method to call. That is why every programming language supports some form of dynamic dispatching.

It is usual that for languages that were designed to be object-oriented – like Java, Eiffel, Smalltalk, etc – this is the default behavior. Other object-oriented languages support dynamic dispatching by marking some methods as "virtual", and the calls of virtual methods will be dispatched dynamically. When we define a new class the reference of the virtual methods will be registered in the *Virtual Method Table* (or VMT). When we call a virtual method the program will chose the appropriate method by the dynamic type of the object from the suitable Virtual Method Table in run time.

Methods of subtypes

Inheritance makes it possible to derive classes from existing classes, – for example, *Circle* $\subseteq$ *Shape*. But this is not only about the type values, but the behavior of the subtype instances. This means that we can only change methods that way safely if the substitution of instances of derived classes is possible for instances of base class [Bru02]. We should be able to handle a list of *Shape*s by the methods of *Shape*, when the concrete objects are instances of *Circle*, *Rectangle*, etc. too. A method is described by its signature (the type of its parameter and the type of its result), and the pre- and postconditions.
Let

$$\begin{aligned} fss &= \textbf{proc}(Shape)\ \textbf{returns}\ (Shape) \\ fks &= \textbf{proc}(Circle)\ \textbf{returns}\ (Shape) \\ fsk &= \textbf{proc}(Shape)\ \textbf{returns}\ (Circle) \\ fkk &= \textbf{proc}(Circle)\ \textbf{returns}\ (Circle) \end{aligned}$$

be any methods and we should examine which method can be replaced by which one. (What redefinitions are possible in the derived classes?)

$$\begin{aligned} fcs \subseteq fss\,? \qquad & fsc \subseteq fss\,? \\ fcc \subseteq fss\,? \qquad & fss \subseteq fcs\,? \end{aligned}$$

A method is marked as $A \to B$ if it gets type A as parameter, and returns type B. If $A' \to B'$ methods are subtypes of $A \to B$ methods (formally: $A' \to B' \subseteq A \to B$) Then we should be able to use any elements of the first method type in every context where we used any elements of the second method type.

Suppose we have a f method which is typed $A \to B$. If we want to substitute $f' : A' \to B'$ for f, then f' should take A as parameter and should return B as result. The domain of f' is A' this means we can only apply f' if A is a subtype of A' ($A \subseteq A'$) so $a : A$ is a A' and $f'(a)$ is correctly typed.
On the other hand: the result of f' has the type of B', but we should be able to handle it as an instance of B, which is only possible if $B' \subseteq B$ is true.
Summary: $A' \to B' \subseteq A \to B$ is only possible if $A \subseteq A'$ and $B' \subseteq B$.

This means that in case of redefining methods the result type can be the same (novariant), or more specialized which is called covariant or monotone (see Section 11.3), but the type of the parameter can be the same (novariant), or more general, which is called contravariant or anti-monotone. So in the previous example:

- the $fsc \subseteq fss$ – result can be more specific
- the $fss \subseteq fcs$ – parameter can be more general

The same rules stand for the preconditions and postconditions of the methods: the postcondition can be the same (novariant), or more specific (covariant), and the precondition can stay the same (novariant) or can be weakened (contravariant), see Chapter 12. Most of the programming languages require signature identity in case of overriding, thus the method call can be checked in compile time.

It can happen that the derived class has to override a method with a covariant parameter. For example we should derive the *Skier* class from *Sportsman*, and derive two more classes from *Skier*: *Skier_boy* and *Skier_girl*. State that we have introduced a *roommate* method to *Skier* which takes another *Skier* as parameter.

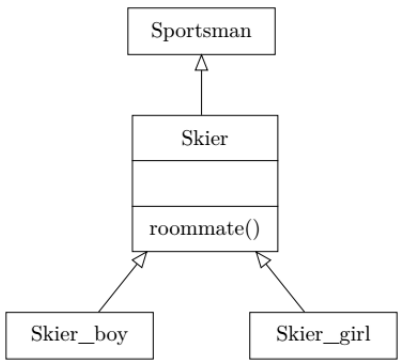


Figure 10.9: Class hierarchy of Skiers

How can we override the *roommate* method in *Skier_girl* class? Is there a solution for covariant overriding? (To make it accept only *Skier_girl* as parameter.) Is there only a novariant override? (It will take any *Skier* as parameter.) Or is there a contravariant overriding? (We can override it to take *Sportsman*.)

Polymorphism and dynamic dispatching in C++

In C++, the compiler will only allow dynamic dispatching in case of methods marked with the **virtual** keyword. The instances of derived classes can be assigned to an object of the base class, but we should mind not to work with objects, but with references, otherwise the assignment result will be truncated, and the dynamically dispatched methods will be statically bounded. Dynamic dispatching works correctly only in case of references.

For example we should introduce the *foo* method to *Rectangle*, which should return *0*. Then we should derive the *ColoredRectangle* class, which should hold *color* as an integer, and we should override the *foo* method to return *color*. Then we instantiate a *Rectangle* and *ColoredRectangle* object. Polymorphism seems to be allowed, and it looks possible to assign the derived object to the base object, but it will be truncated, and the *foo* method would be statically bound to the object. The real polymorphism and dynamic dispatching will only be possible if we reference the addresses of the objects, as in the example below:

```
class Rectangle {
  protected:
    int width, height;
  public:
    Rectangle(int w, int h) {
        width = w; height = h;
    };
    virtual int foo() { return (0); }
};

class ColoredRectangle: public Rectangle {
     int color;
  public:
     ColoredRectangle(int w,int h,int c):Rectangle(w,h),color(c){};
     virtual int foo() { return (color); }
};

void main() {
    Rectangle r(2,3);
    ColoredRectangle cr(3,4,5);
    r = cr;                  // Truncated
    cout << r.foo();         // Rectangle::foo()
    Rectangle* r1 = &r;
    Rectangle* r2 = &cr;
    cout << r1->foo();       // Rectangle::foo()
    cout << r2->foo();       // ColoredRectangle::foo()
}
```

The heir can only redefine the virtual methods ”safely”: the type of the parameters cannot be changed, the type of the result can be covariant. To change the type of the parameters we can use overloading. In the previous *Skier* example the derived *Skier_girl* wants to change the type of the parameter of *roommate* to *Skier_girl* (in a covariant way). It can be done, but the new *roommate* method will be an overloading not an overriding of the inherited method.

```
class Skier {
   public:
      ...
      virtual void roommate(Skier * s){
          cout « "Skier with a Skier roommate\n";
   };
};
class Skier_girl : public Skier {
   public:
      ...
      virtual void roommate (Skier_girl *g) { // Overloaded
           cout « "Skier_girl with a girl roommate\n";
      };
};
class Skier_boy : public Skier {
    ...
};
void main(){
    Skier *s;
    Skier_girl *g;
    Skier_boy *b;
    g = new Skier_girl;
    s = g;
    s->roommate (b); // The Skier::roommate is called.
    g->roommate (g); // The overloaded Skier_girl::roommate is called.
    s->roommate (g); // The Skier::roommate is called.
}
```

We should note that if we do not override the inherited *roommate* method then at the call `g->roommate(b)` some compiler will not recognize that it should call the *Skier::roommate* method but it would fail with a compile time error. In case we introduce the redefined version of the inherited method then the compilers can understand the method call.

There is one exception when C++ does not call the method of the dynamic type. If we call a method from the constructor, and that method is marked as **virtual** then that method will be called every time, when we instantiate any objects through that specific constructor. It is irrelevant whether the descendant class has overridden the method or not.

The logic behind this is class invariant. The function of constructor is to set up class invariant correctly, if a class is defined with a function call in its constructor, then it is considered as the execution of that method is necessary for setting the correct class invariant. We cannot replace it.

```
class Foo {
   public:
      Foo() {
         method();
      }
      virtual void method(){
         cout « "Method of original class\n";
      }
};
class Bar : public Foo {
   public:
      Bar() {
         method()
      }
      virtual void method () {
         cout « "Method of derived class\n";
      };
};
void main() {
    Foo f = new Foo();          // Prints "Method of original class"
    f.method();                    // Prints "Method of original class"
    Bar b = new Bar();          // Prints "Method of original class
                                   // Method of derived class"
    b.method();                    // Prints "Method of derived class"
}
```

In the example we can see that when we have constructed *b* then we have printed two messages, the message of the base class, and the message of the descendant. This is because both constructors have called *method* and the constructor of the base class has run first. In this case *method* was called as it had not been overridden at all. In the second case when the constructor of *Bar* was executed, then it has printed the message of class *Bar*.

The features of Object Pascal

In Object Pascal the variables are references of objects. Polymorphism means that a base class typed variable can reference a derived instance. The methods are statically bounded by default – as in the case of C++. For dynamic dispatching we have to introduce the method with the **virtual** keyword and then the method will be virtual. (For example: **procedure** *foo;* **virtual**.) If we want to override it

in the derived class then we have to use the **override** keyword. (For example: **procedure** *foo;* **override**.)

It is possible to overload method names, and we have to use the **overload** keyword for this. (For example: **procedure** *foo(a:string);* **overload**.) In case of overloading we can use methods with different parameter types. If we introduce an overload with the same signature, the new method will hide the inherited one. It is possible to hide virtual methods with overloading. In this case we need the keyword **reintroduce** (e.g. **procedure** *foo;* **reintroduce**; **overload** [Can00].)

Polymorphism and dynamic dispatching of Java

In Java, the variables hold the references of objects, so their polymorphism is default, and we do not need a keyword to support dynamic dispatching. If we introduce a method with the same signature in the derived class then overriding will take place and the method will be dynamically dispatched.[10]

In the following example [Nyek08] we will introduce *bonus* method to the *Employee* class. The *Boss* class is derived from *Employee* and the *bonus* method is overridden.

```
public class Employee {
  ...
  int numberOfLanguages;
  public int bonus() { return numberOfLanguages * 50; }
  public int salaryWithBonus()
    { return salary() + bonus(); }
}

public class Boss extends Employee {
  ...
  //@Override could be written in front of the method
  //to avoid overload by mistake
  public int bonus()
    { return super.bonus() + numberOfEmployees * 100; }
}
```

It will call the method by the dynamic type of the variable by default:

```
Employee e = new Boss("John Doe");
int b = e.bonus();              // bonus method of class Boss.
int sb = e.salaryWithBonus(); // method of class Employee.
```

[10] In Java 1.5 *@Override* annotation was introduced to make it possible to avoid overloading with a mistake in method signature.

Polymorphism and dynamic dispatching in C#

In C#, – just as in Java – objects can be handled through their references, but methods – just as in C++ – can be statically and dynamically bound. For dynamically bound methods we have to use the keyword **virtual**.

public virtual void *foo() {. . .}*

For overriding – just as in Object Pascal – we have to use the **override** keyword:

public override void *foo() {. . .}*

We do not need any keyword for overloading. The compiler chooses the "closest" method by the parameters. (For more information, see [Csref03].) The derived class can hide the inherited method with the **new** keyword – this is similar to *reintroduce* in Object Pascal. So the **new public virtual void** *foo()* method hides the inherited *foo()* method.

Polymorphism and dynamic dispatch in Eiffel

Eiffel was designed in a fully object-oriented approach, thus the variables are references by default and the call of the actual methods can only happen dynamically. The derived class has to give in the inheritance clause (with the keyword *redefine*) the methods which wants to override.

```
class POLYGON inherit
    SHAPE
        redefine F end
```

Pre- and postconditions can only be redefined in a "safe" way, thus preconditions can only be weakened and post conditions can only be strengthened (see Chapter 12.). The type invariant can only be narrowed to show that the type values of the derived class are between the type values of the base class [Mey91]. The language also supports covariant parameter change, which can cause interesting run time errors. For example, suppose that in the *POLYGON* class the redefinition of the method *F* calls the method *G* which is introduced in *POLYGON* class and is not available in an other derived class (like *CIRCLE*) of class *SHAPE*.

```
class SHAPE
 feature
  F(S:SHAPE) is do
    io.putstring ("F in SHAPE");
  end
end
class POLYGON inherit
    SHAPE
        redefine F end
```

```
 feature
   F(S:SHAPE) is do
     io.putstring ("F in POLYGON");
     S.G;        -- Call the new G method of the parameter.
    end
  G is do
     io.putstring("G new in POLYGON")
   end
 end
class CIRCLE inherit
    SHAPE   -- Does not redefine F. ...
 end
```

In this case if we assign an instance of *POLYGON* to a *SHAPE* reference, and we do not call the *F* method with a parameter that fits the dynamic type, then the compiler will not remark this and it will only turn out in run time that the parameter is not suitable.

```
o1, o2: SHAPE;
make is
 do
    !POLYGON! o1 -- Assigns a POLYGON object to o1.
    !CIRCLE! o2  -- Assigns a CIRCLE object too2.
     o1.F(o2);   -- Run time error.
```

Polymorphism and dynamic dispatch in Python

The case of Python is very simple since we have a dictionary object that holds names and definitions. If a method calls another method, then the name of the called method will be looked up in the dictionary of the instance. If no definition can be found then it will search in the dictionary of the class. If the class does not hold the definition then the search will continue in the dictionary of the first base class, etc.

In Python, we can have more than one base classes. That also means, we can have more than one base class that has introduced a method that we have not overridden. Which method will be called then? The rule in Python is mainly depth-first, left-to-right. It will look for the method in the class, if the execution environment cannot find any methods with the given name in the dictionary of the class, then it will take the first class from the list of its base classes and will look for the method in it. (If the method is not registered in the class, it will search in its base classes.) If the (recursive) search of a base class does not give any result, then the execution environment will look for the method in the next base class.

This easy algorithm had one problem: in a diamond inheritance situation the common base class could be processed multiple times, and the general method

of the base class could be resolved before the specialized method of descendant class. Consider this example:

```
class A(object):
   def f(self):
      print 'A';
class B(A):
   def f(self):
      print 'B';
class C(A):
   pass
class D(C,B):
      pass
```

In the above example `D` has inherited two `f()` methods. In the precedence order of `D` `C` has a superior precedence than `B`. This means that method `f()` will be searched in `C` first than in `B`. This was a problem, that is why Python uses here the *C3 linearization* which respects the local precedences of the objects and the monotonicity of resolving. This means that a base class cannot be resolved prior to any of its specialized classes. In the above example `D` would resolve in `C` then in `B` and only after that would resolve a name in `A`.

You cannot always respect these rules in a situation where two classes define conflicting preference orders, this algorithm fails. Consider this example:

```
class X(object):
   pass
class Y(object):
   pass
class A(X,Y):
   pass
class B(Y,X):
      pass
class C(A,B):
      pass
```

In this example `C` should resolve in `X` then in `Y` according to `A`, but it should resolve in `Y` and only after it in `X` according to `B`. In situations like this Python raises an exception to warn the programmer it is impossible to derive from those two classes. (In earlier versions of Python it only has chosen an *ad hoc* ordering.)

Polymorphism and dynamic dispatch in Scala

Scala has the keyword `override` which is used to override a method. We have to mark every methods with override if we want to redefine them. Overriding a method without the keyword, or marking a newly defined method with override

(which does not override any methods of any base classes) will both result in a compile error.

Dynamic dispatching takes place just like in case of Java. (If our Scala code is executed on JVM it is done exactly like in Java.)

Calling the methods of the base class

During implementation we might have to call the methods of the base class, or to extend a method in a child class by using the original method.

In single-inheritance languages there is usually a keyword to reach base class.

For example in Object Pascal we can use the **inherited** keyword in a method or in a constructor to reach the method with the same name in the base class, and we can execute any statement before or after it [Can00].

```
type TParent = class ...
  procedure f(...); virtual; ...
end;
type THeir = class(TParent) ....
  procedure f(...); override; ...
end;
procedure THeir.f(...);
begin
  inherited f(...);  { Here we call f method of TParent f. }
  ...
end;
```

In Java (and in Smalltalk), we have the keyword **super** to call the base class, as we have seen in the example in Section 10.7.2 when *Boss* class has overridden the inherited *bonus* method of *Employee*:

```
public int bonus() {
    return super.bonus() + numberOfEmployees * 100;
}
```

In C#, we can use the **base** keyword to reach a method of base class [Csref03]:

```
public class A {
      public virtual void Print() { ... }
}
class B: A {
      public override void Print() {
         // Calls the Print method of base class:
         base.Print();        ....
      }
}
```

In some languages we can use the name of the base class to reach its methods.

In C++ we can use the *::* scope operator to reach one of the base classes: *ParentClass::Method(parameter_list);* This is also used to solve the conflicts of multiple inheritance. (See Section 10.7.5.)

In Python, we can call the method of the base class by calling the method on the name of the base class, and passing the parameter *self* to the method.

```
class Foo:
    def print(self)
       print "Foo's print";

class Bar(Foo):
    def print(self)
        print "Bar's print ";
       Foo.print(self);

b = Bar();
b.print();                  # Prints "Bar's print Foo's print"
```

In Scala the methods of the parent class are available through `super` keyword. This reference will point to the base class just as in java. We cannot call the constructor of our base class (like `super()` in Java) since this call is considered to be strongly bound to the hierarchical object creation, and the constructor of the base class was called by the definition of the class. However we can inherit multiple implementation from more than one `trait` and in this case we can choose between implementations with square brackets. (If we want to specify that we want to reach parent `A` than we can say `super[A]`.)

It was a design concept of Eiffel that we should not have any references to the super class outside of the inheritance clause. That is why the language support a special type of multiple inheritance the *repeated inheritance*.

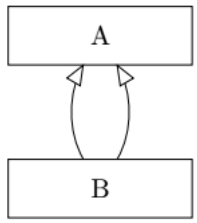


Figure 10.10: Repeated inheritance

If we want to use the feature f that was inherited from the base class and also want to redefine it then we have to reinherit it: In one branch we rename it (let's say to *old_f*) to be able to use it, and on the other branch we override it (where we can use the renamed old version):[11]

[11] The mean of the **select** statement see at 10.7.5.

```
class Ds
  inherit D
    rename f as old_f
  end;
  inherit D
    redefine f
    select f
  end
feature
  f is
    do
      ...
      old_f;
      ...
    end
end
```

Calling the methods

The derived classes can introduce new methods that were pointless in the base class, and can use them to extend the set of possible operations. For example we can introduce *bark* method in class *Dog* which was derived from *Animal*. The polymorphism and dynamic dispatching puts the question: how should we call these new methods?

Imagine, that we have a *pet* variable which is an *Animal*, and we have assigned an instance of a derived class (like *Dog*) to this variable. If we have introduced an *eat* method to *Animal* that we might have overridden in the actual dynamic type of *pet* it is absolutely natural to allow the call of *pet.eat* which will be dynamical dispatched – as the language makes it possible – to the corresponding method. But if we have introduced *bark* method in *Dog*, should we allow to call *pet.bark*?

Most of the programming languages – like Eiffel, C++, Java, Object Pascal, etc. – chooses the method to call by the *static type* of the variable, which leads to a compile time error in the case above. This decision makes the programming languages safer, because if they had used entirely dynamic dispatching (like in Smalltalk or Dylan, etc.) it could have led to some interesting run time errors. For example in case of some mistake the dynamic type of *pet* would not be *Dog*, but for example *Cat* which of course does not have a *bark* method, it could have led to a run time error.

10.7.3 Abstract class

When we design a program, and we design the class hierarchies, the most important task is to make the appropriate type specifications. This layer is independent of implementation. The representation of the type values, and the

implementation of behavior will come in the next step. Polymorphism and dynamic dispatching make it possible to design abstract classes and handle them uniformly to make it possible to segregate a general code from actual implementation.

An abstract class can contain methods that have no implementation and they are only specified and abstract classes can also have implemented methods as well. The inverse of this statement is true as well: if we have a class that contains at least one abstract method, then that class is abstract.

Usually an abstract class cannot have an instance. The variables with the type of an abstract class can reference objects of a derived class of that class. For example in a general *Shape* class we cannot provide a corresponding implementation of the *area* method, but we can provide easily an implementation for *Circle*, *Rectangle*, *Polygon*, *Triangle*, etc. It is useful to introduce the *area* method in the base class because we want to use this in every derived classes, and this way we can make it possible to handle the list of *Shape*s uniformly.

Most of the object-oriented programming languages make it possible to introduce abstract classes. There are languages – like Java, C# or Eiffel – that make it possible to declare a class abstract without a single abstract method, and there are languages – like C++ – that require at least one real abstract method to make the class abstract. In languages that provide mechanisms to forbid further inheritance[12] they forbid to use these two features together. This behavior is a trivial consequence of the fact that an abstract class must have a derived class, and a class which is closed for further inheritance cannot have any derived classes.

We can build the hierarchy of abstract classes, the derived classes can have abstract methods, they might introduce new abstract methods, or they can add the implementation of an inherited abstract method, but they will remain abstract classes as long as they have at least one method without an implementation.

Abstract classes in programming languages

SIMULA 67 - Smalltalk

It is interesting to note that in SIMULA 67 – which can be considered as the first language that supports object-oriented programming – it was possible to make a method without an implementation.
In Smalltalk, – which was the first language that was designed to be pure object-oriented – it was not possible.

[12] For example: In Java a class can be *final*, in C# a class can be *sealed*, in Eiffel a class can be *frozen*

C++

In C++ abstract classes are not marked with any keyword if they have any abstract methods – which is marked with a = *0* instead of the body of the virtual method[13] – then the class is considered as an abstract class, and it cannot have any instances. For example in the *Shape* class method *area* should be abstract, what we can achieve with the

```
virtual int area() = 0;
```

specification, which is equal to the following empty implementation:

```
virtual int area() { return (0); };
```

Object Pascal

In Object Pascal, we have to mark the abstract method with **virtual** and **abstract**. For example:

```
function foo: string; virtual; abstract;
```

The implementation should be marked with **override** as in case of other **virtual** methods. The class can have abstract methods, but the class will not be considered as an abstract class, we can instantiate it and it will raise a run time error if we call an unimplemented method [Can00].

Java

In Java, we have to use **abstract** if we want to introduce an abstract class or method. The general rules are applied here [Nyek08]. For example we can declare the abstract *Shape* class with the specification of the abstract *area* method, and we can easily give the implementation in the actual derived class:

```
public abstract class Shape { . . .
   public abstract double area();
}
public class Rectangle extends Shape { . . .
   //@Override annotation can be used since Java 1.5
   public double area() { return width * height; }
}
```

[13] Abstract methods in C++ are called as *pure virtual* methods.

C#

In C#, we have to use the keyword **abstract** to introduce an abstract class or method. The implementation – over against Java, like in Object Pascal – have to be marked with the keyword **override** [Csref03].

```
abstract class Shape {. . .
   public abstract int area(); . . .
}
abstract class Convex : Shape {. . .
   public void foo() {. . .}
}
class Rectangle : Convex {. . .
   public override int area() {
      return Width * Height;
   }. . .
}
```

In the *Shape* abstract class we introduce the *area* abstract method. In the *Convex* class we introduce the new *foo* method but we do not give the implementation of the abstract *area* method, so *Convex* will stay abstract. The *Rectangle* implements the *area* method, there are no other abstract methods left, so *Rectangle* does not have to be marked with the keyword **abstract**, however, it could be.

Eiffel

In Eiffel, we can mark abstract methods with the keyword **deferred** in the first line of the class and in the place of the method body.

```
deferred class SHAPE ...
feature ...
  print is
    deferred
  end ....
end
```

The implementation of the abstract method does not mean overriding, so it does not have to be marked in the inheritance clause.

```
class CIRCLE inherit
    SHAPE
feature ...
    print is
        do
            io.putstring ("I am a CIRCLE!");
        end; ...
end
```

It is possible to make an already implemented method to be abstract. It can be done with the keyword **undefine** in the inheritance clause. This means that we change an implemented method to be **deferred**. This method retains the whole signature of the method with its preconditions and postconditions, it will only remove the body of the method. The **undefine** clause is useful in case of multiple inheritance (See Section 10.7.5.) which explains why it is so common in the standard library of the language. For example:

```
deferred class CHAIN [G]
  inherit
    CURSOR_STRUCTURE [G]
        undefine
            prune_all ....
```

It is a very important feature of Eiffel that it makes possible to add a type invariant to a deferred class, and to set pre- and postconditions to deferred methods thus makes it possible to build very type safe class hierarchies. For example, most of the methods of the *LIST* class are abstract, but the type invariant, the post and preconditions make it possible to communicate its semantics:

```
deferred class LIST[G]...
feature ....
  forth is  -- Moves cursor forward with one positions
    require
     not after
  deferred
    ensure
      index = old index + 1
  end
invariant
non_negative_count:count >= 0
offleft_by_at_most_one:index >= 0
offright_by_at_most_one: index <= count + 1
end
```

Python

There is no language feature in Python to implement abstract classes. Python has a module, which is called *abc* (*abc* stands for Abstract Base Class), and it has the *ABCMeta* class which can be the ”base class” of our abstract classes. If we set *ABCMeta* as the meta class of our class, then we can create ”abstract methods”, by marking them with *@abstractmethod*. If we derive a class from an abstract class, all the abstract parts of that class must be overridden.

Abstract methods of our Abstract Base Class can have an implementation. The overriding of these methods can call the implementation of the abstract class, but this does not happen automatically.

We can *register* classes to an abstract class, to make it a "virtual-subclass" of our abstract class. This means that if we call *isinstance()* on the class and on the abstract class, then the Execution Environment will return true. However methods in the abstract class will not be available in the virtual abstract class.

If we want to change this behavior then we have to redefine *__subclasshook__* method of our Abstract Base Class. *__subclasshook__* method can return three values: *True* which means that this class is a subclass of the Abstract Base Class, *False* which means the opposite, and *NotImplemented* which means that the given class is not a sufficient class, the class check can go on as usual. We can make interface checks in this method.

We can also define abstract properties with the *@abstractproperty* notation.

```
class Foo(object):
    def f(self):
        return 0;
class MyAbstract(type):
    __metaclass__ = ABCMeta
    @abstractmethod
    def f(self):
        return 1;


    @classmethod
    def __subclasshook__(cls, C):
        if cls is MyAbstract:
            if any("f" in B.__dict__ for B in C.__mro__):
                return True
        return NotImplemented

MyAbstract.register(Foo)

class MyDirect(MyAbstract):
    def f(self):
        return 2;
```

In this example, we have `Foo` which is a quite common Python class derived from `object`. We also have `MyAbstract` which is an Abstract Base Class, it has `ABCMeta` as his meta class. It has made `f()` to an abstract method which means we have to override it in every subclasses. In `MyDirect` which is a direct subclass of `MyAbstract` we have implemented `f()` as it is required. We have defined `__subclasshook__` to check whether any subclasses of MyAbstract (which can

be done by virtual or direct sub-classing) has `f()` defined in the class or in any base classes of the class. (`__mro__` is the Method Resolution Order list.)

Scala

Scala has the keyword `abstract` to make a class abstract. If a class is abstract it can have methods without body. We do not have to mark these methods with the keyword `abstract`, they are automatically considered to be abstract. If our concrete class inherits from an abstract class then it has to provide an implementation for the unimplemented methods, or the class should be marked with `abstract`. We have to use the keyword `override` when we give the implementation of abstract methods.

10.7.4 Common ancestor

Some programming languages – like Smalltalk, Java, C#, Eiffel, etc. – apply the idea that every class has a common abstract ancestor. This class has all the common attributes of the classes like bitwise comparison, copying, etc. Other languages – like SIMULA 67, C++ or Ada – do not introduce common ancestor thus make it possible to build new independent class hierarchy trees.

The other benefit of common ancestor is substitutability. We say that every derived class can replace the base class, which can be a used at parametrization. We can declare methods with the common ancestor as a formal parameter and when we call the method the formal parameter can be assigned to instances of any derived class.

In the languages that have a common ancestor[14] we cannot introduce any class without deriving it from the common ancestor. If we do not derive it from any classes the compiler will set the common ancestor to its base class.

It is interesting that in Object Pascal, we have both. Both the **object** and **class** keyword can mark a class, and the language solves this problem differently. The **object** class was introduced in Turbo Pascal 5.5 and Object Pascal took this idea. The keyword **class** was introduced in Object Pascal. The **object**s of Turbo Pascal do not have a common ancestor thus the **object** type variables of Object Pascal do not have a common ancestor. However every class that was introduced with the **class** keyword has a common ancestor called *TObject*.

Python did not have any common ancestor in its early history, but later the designers of the language have introduced **object** which can be the ancestor of all classes. For this, there are two types of class creations in Python. The first one is called old-style class creation when we just simply define a class and the other one when we derive our new class from **object**. (This is called new-style class creation.) We even get a new-style class, if we derive our class from a class which has **object** as one of its base class. The aim of the language implementers

[14] The common ancestor is usually called `Object` or `TObject`.

was to create a unified object model with full meta model. (Old-style classes are removed from Python 3.0.)

In Scala the common ancestor is called *Any*, which is closer to the philosophy of functional programming languages. *Any* has two direct subclasses *AnyRef* which is the ancestor of every **var** (variable) in the language, and *AnyVal* which is the ancestor of immutable values. Even **Unit** is a descendant of *Any* (through *AnyVal*) which is a special value object that is returned by methods with side-effects (usually the *void* methods in other strongly typed languages).

10.7.5 Multiple inheritance

We talk about multiple inheritance if a class has more than one direct base classes. In the simplest way it means that a class can be considered as a specialization of two (or more) independent classes. In his book [Mey00] Bertrand Meyer lists several natural causes why a class could have more than one base classes. For example, the *COMPANY_PLANE* can inherit the attributes of a *PLANE* so a pilot can handle it, and it can also inherit the attributes of *COMPANY_PROPERTY*, so the accountant of the company can handle it. Based on similar considerations they have introduced several possibilities to the standard library, like: the objects of the derived classes of *NUMERIC* have the $+$, $-$, $*$, $/$, etc. operations. The objects of the derived classes of *COMPARABLE* have operations like $<$, $>$, etc. Both of these classes are used to derive the *INTEGER* and *REAL* classes.

An other important feature of multiple inheritance can be observed when we use it to combine specification and implementation inheritance. One of the base classes gives the behavior, and the other – which is preferred to be hidden – gives the representation. For example the class *FIXED_STACK* inherits its behavior from the abstract *STACK* and its representation from *ARRAY*. Bertrand Meyer calls this form of multiple inheritance as "marriage of convenience" [Mey00]. In his witty example the child of the impoverished noble family (the abstract class) marries the child of the rich commoners (the representation), and they both reach what they want: title (functionality) and wealth (implementation).

There are several solutions for multiple inheritance in C++:

- For multiple specification inheritance a class can have more public base classes, like:

```
class A { //...
  public:
    virtual void foo() { cout << "foo in A\n"; };
};
class B { //...
  public:
    virtual void bar() = 0;
};
class C : public A, public B { //...
  public:
    // Override A::foo() method.
    void foo() { cout << "foo in C\n"; };
    // Implements a B::bar() method.
    void bar() { cout << "bar in C\n"; };
};
```

The derived class inherits the member variables and methods of the base classes, it can override, implement or can leave them unchanged. Polymorphism and dynamic dispatching makes it possible to reach the methods of the base classes through the references of the base classes:

```
int main() {
   A* a1;
   B* b1;
   C c;
   a1=&c;
   b1=&c;
   // Calls the method of the dynamic type:
   a1->foo();
   b1->bar();
   return 0;
}
```

Through ancestors we can reach only the methods they have inherited to us. For example we cannot write *a1->bar()* because class *A* does not have a *bar* method.

- For the specification and implementation inheritance we have to set for the specification inheritance the base class **public**, and for the implementation inheritance the base class **private** or **protected**. It is interesting that the default inheritance mode is the **private**. For example class *C* will inherit specification form class *A* and implementation from class *B*:

```
class C : public A, B { //...
  public:
    void foo() { cout << "foo in C\n"; };
    void bar() { cout << "bar in C\n"; };
};
```

Here polymorphism and dynamic dispatching only works through A references. The class C has published the *bar* method but it is not a subtype of B, it cannot be referenced through B.

- It is possible to have multiple implementation inheritance. In this case the derived class can make public any methods it has inherited from the base classes (the **public** and **protected** parts of the base classes) but the other inherited methods will not be available for the users of the class, and we cannot reference the class through a variable of a base class.

```
class C : A, B { //...
  public:
    // Override A::foo() method.
    void foo(){cout<<"\n foo in C \n";};
    // Publish the bar method which was inherited from B
    B::bar;
};
```

If we have a c instance of class C and call *c.foo()* then we call the overridden method. If we call *c.bar()* we call the *bar* method of B class.

In Eiffel, the default inheritance is specification inheritance but – as we have seen in Section 10.7.1 – the selective visibility makes implementation inheritance possible, when we hide every inherited attribute from the users of the class.

Handling name conflicts

The possibility of multiple inheritance raises some interesting questions: even totally different classes can have methods with the exact same signature – which one should be inherited in the derived class? Renaming can mean a good solution for this if we want to differentiate the methods but we can also ”join” them and give a common body that fulfills the requirements of all base classes.

Renaming

In Eiffel, the derived class simply renames the inherited method in the inheritance clause. This will not change the inherited method just introduces a new name to the class. This can be used to resolve name conflicts:

```
class A
 feature
  foo  ....
end;

class B
 feature
  foo ....
end;

class C
 inherit
   A
    rename foo as aFoo
    end;
   B;
 feature  ....
end;
```

The *C* class inherits a *foo* method from *A* and from *B*. It can rename any of them – for example the *foo* of *A* to *aFoo* – which makes differentiation possible between them. In the example above, the *foo* feature in *C* will mean the feature of *B*.

In Eiffel, it is not possible to override names so we have to rename methods with the same name and different signature.

According to Bertrand Meyer it is a good way to introduce new name conventions in the derived class [Mey00]. For example, we can inherit *WINDOW* from *TREE* and *RECTANGLE* and rename the methods to fit the notions of *WINDOW*.

C++ does not allow renaming.

Join

It is possible, in case of multiple inheritance, that we want to merge more methods with the same name and signature.

For example, in C++ if a class inherits a **virtual void** *foo()* method from multiple base classes and redefines it, then through polymorphism and dynamic dispatching that method would be called from any references of the base classes:

```
class A{
  public:
     virtual void foo() { cout « "foo in A\n"; };
};
class B{
  public:
     virtual void foo() { cout « "foo in B\n"; };
};
class C: public A, public B {
  public:
      virtual void foo(){ cout « "foo in C\n"; };
};
```

The overridden *foo* method of *C* replaces all the inherited *foo* methods, and there would be no name conflict. In case of any references this *foo* method would be called: `C c; B* b1= $&$c;` in this case `b1->foo()` would be the same as `c.foo()`. We have a different solution if we want to choose a method from one of the base classes:

```
class C: public A, public B {
  public:
      B::foo();
};
```

This solution breaks subtyping: *C* is not a subtype of *A and B*. Because if we have a code like this: `C c; B* b1= $&$c;` then we will get the same result in case of `b1->foo()` and `c.foo()`, but in case of `A* a1 = \c; a1->foo()` we will (and we can only) reach the *foo* method of A.

It is possible that the derived class inherits methods with the same name but with different signature [Str00]. In this case, **using** declaration can help to choose corresponding method:

```
class A{
  public:
     virtual int foo(int) {. . .};
};
class B{
  public:
     virtual double foo(double) {. . .};
};
class C: public A, public B {
  public:
     using A::foo;
     using B::foo;
     virtual void foo(){. . .};
};
```

In this case *C* would have two overloaded *foo* methods.

In Eiffel, we can join features with the same name, if they have the same signature and both of them are deferred. This is the point where the keyword **undefine** becomes handy:[15] we can make features that have already been defined to become abstract again. We can redefine the functions to get signature identity, or to get identical names. The joined methods will have their preconditions connected with an *or* operation, and the post conditions are connected with an *and* operation. The derived class can keep the feature deferred or can define it. Bertrand Meyer said: "We can kill more abstract birds with one stone" [Mey00].

```
class D
  inherit A
    rename bar as foo
    undefine foo
  end;
  inherit B
    rename baz as foo
    undefine foo
  end;
  feature
    foo : T is
    ...
end -- class D
```

Join can be important in case of already **deferred** methods when we want to merge different abstractions. In this case we do not have to **undefine** anything. For example the *CHAIN* class – which represents the list of sequential structures in the standard library – inherits from two abstract classes which both have a **deferred** *item* feature, which returns the *item* at the actual position of the cursor:

```
deferred class CHAIN[T] inherit
      BIDIRECTIONAL[T] ...
      -- It has an item deferred routine in the iteration hierarchy which
      -- is the value of the actual element.
      ACTIVE[T] ...
      -- It has an item deferred routine in the access hierarchy which is
      -- the element under the cursor position.
  ...
end
```

The *CHAIN* class inherits both *item* features which would lead to a name conflict that can be resolved with renaming but now it is advised to be joined.

[15] See Section 10.7.3

Handling of diamond inheritance

A special case of multiple inheritance is repeated inheritance, when a class can reach a base class through different paths in the inheritance graph. In this case the class will inherit the same attributes more than once. How many instances should an object of a class have from these attributes and how should it reach them? This is the problem of ”diamond inheritance”.

If in the base class A we have an *attr_a* attribute and a *foo* method, than *all of the derived classes of* A should inherit them. If A has two derived classes: B and C then these both are going to contain *attr_a* and *foo*, and they can introduce new member variables like *attr_b* and *attr_c* and new methods like *bar* and *buz*. If we derive a D class from B and C, it will have *attr_b* and *attr_c* properties and *bar* and *buz* methods for sure, but how many *attr_a* and *foo* should it have? How should it reach them? How should it differentiate them?

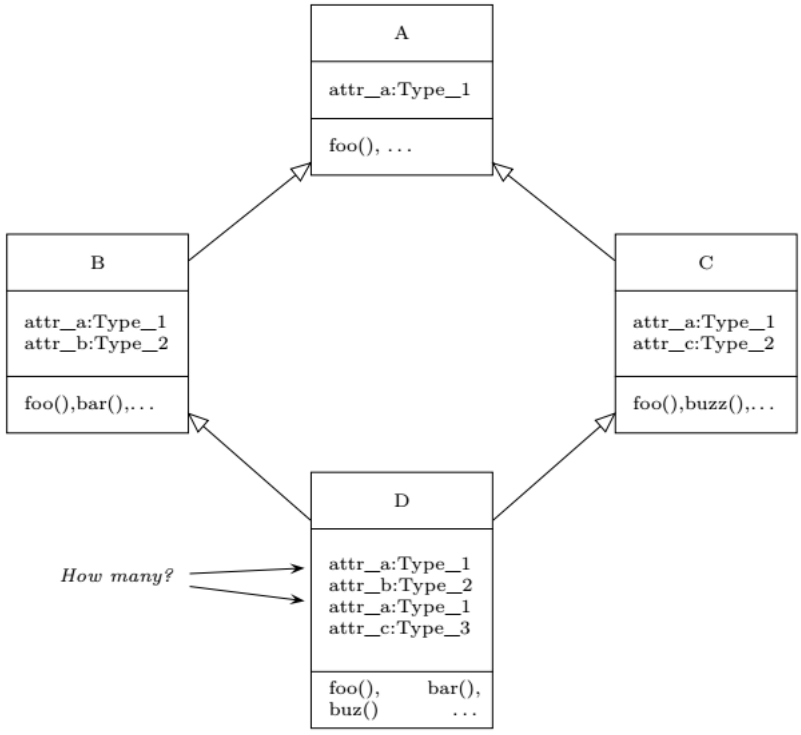


Figure 10.11: Problems of multiple inheritance

There are different solutions for this problem in the different languages. To analyze this problem we should examine the two typical languages that support multiple inheritance: C++ and Eiffel. We should examine separately what they do with multiple inherited methods and member variables.

Multiple inheritance of member variables

In C++, the default solution is that every member should appear as many times it was inherited and you can reach them through the name of the base class (like: *C::attr_a*). It can happen that there is no sense in multiplying the inherited attributes. For example, we have the *Person* class with *name* and *placeOfBirth* attributes and we have derived *Student* and *Teacher* classes from it. Imagine we have to derive the *Tutor* class which represent a *Student* who has *Teacher* responsibilities as well. There is no use of having *name* and *placeOfBirth* attributes multiple time in the object. In this case in C++, the base classes can decide to use a "virtual ancestor" which makes it possible for the common derived classes to only inherit the attributes of the "virtual ancestor" once [Str00].

It is possible that we inherit member variables from the common ancestor that we want to have only once and some that we want to have multiple times. For example it is possible that tuition fee and salary should be sent to different accounts. There is no solution for this in C++. In Eiffel the default solution is to have all attributes inherited once, and the developer can rename any of them. Renaming also means the multiplication of the inherited attribute [Mey91].

Multiple inheritance of methods

In case of diamond inheritance, the unchanged virtual methods of the common ancestor are inherited once by the common descendant. It is different if any of the derived classes redefine the inherited method. In this case the common descendant will inherit more of this method which is only resolvable in C++ if the derived class overrides the inherited method. In Eiffel we can even rename and redefine the method.

The implementation of polymorphism and dynamic dispatching raises a new question: Let there be a method *foo*, inherited from *A* class (See Figure 10.11). Suppose that *B* and *C* redefine it. We derive a class *D* and we do not introduce a new implementation in class *D* for method *foo*. We assign a *D* type instance to an *A* typed *a1* variable. If we call *foo* on reference *a1* which method should be called? The method from *B* or the method from *C*? The current problem is not the lack of implementation but the overwhelm of implementations. To solve this problem Eiffel introduced the *select* keyword in the inheritance clause.

```
class A
  feature
    foo : T is    ...
 end -- class A
 class B
   inherit A redefine foo end
  feature
    foo : T is    ...
end -- class B
```

```
class C
  inherit A redefine foo end
  feature
    foo : T is    ...
end -- class C
class D
  inherit B rename foo as bFoo select bFoo end;
  inherit C rename foo as cFoo end
end -- class D
```

In case of multiple inherited method we have to note in the inheritance clause with the keyword **select** which one to use when the instance is referenced through the base class. Most of the programming languages do not tend to solve the problems of multiple inheritance and do not allow to use it. Some experts advise not to use it at all. Some others think that most of the problems are resolvable if in case of specification inheritance we only inherit from abstract classes and we only implement representation and implementation from only one base class. This approach led to introduce interfaces as a feature of the language.

Multiple inheritance of Python

In Python the problems of multiple inheritance is solved by name resolution. If any method is requested (from outside or from an other method, where it does not count whether the method is defined in the object, in the class or in any base classes of the class) its name is resolved by the same algorithm. First it checks whether the method or variable was defined in the object, than in the class. If not, then it takes the first base class of the class. It the first super class (or its base classes) does not define the requested name, then the algorithm checks the next super class (and its base classes). This is a depth-first, left-to-right algorithm. In case of diamond inheritance Python uses *C3 linearization* which was described in Section 10.7.2.

Multiple inheritance of Scala

Scala only support single inheritance. But every class can extend multiple *trait*s. A *trait* is something like *interface*s in Java. A *trait* holds method declarations and fields. (Until Scala 2.7 these fields could be overridden in the derived classes, but field overriding results in compile time error now.) The methods can have a default implementation. This way a Scala class can inherit two or more implementation of a class. In this case that implementation will be available which belongs to the *trait* that was listed later in the class specification.

```
trait FirstTrait {
   abstract def greet() { println("Hello World!") }
}
```

```
trait SecondTrait {
   abstract override def greet() {
      println("Hello You!")
   }
}
class Greeter extends FirstTrait with SecondTrait
```

In this example `Greeter` class has a `greet` method (which is `Unit`). If we call on a `Greeter` object the `greet` method it will print `"Hello You!"` since `Greeter` will inherit the `greet` method from `FirstTrait` – it was the `trait` that was listed first, then it will be overridden with the default implementation of `SecondTrait` because this was listed later.

It would be impossible to list them in `SecondTrait` and `FirstTrait` order, because `SecondTrait` states that it `overrides` a method called `greet` but `Greeter` does not have any. The second problem is `FirstTrait` states that it will define a new method which is called `greet` but `Greeter` will already have a `greet` method inherited from `SecondTrait` (if we have resolved the compilation problem of `SecondTrait`.)

A common solution to the problem is to have a common base `trait` that is the ancestor of both `traits`. This base trait should declare the common methods thus would make `abstract override` definition of methods valid. This could make both `FirstTrait with SecondTrait` and `SecondTrait with FirstTrait` to compile.

```
trait CommonBaseTrait {
   def greet()
}
trait FirstTrait extends CommonBaseTrait {
   abstract override def greet() {
      println("Hello World!")
   }
}
trait SecondTrait extends CommonBaseTrait {
   abstract override def greet() {
      println("Hello You!")
   }
}

class Greeter extends FirstTrait with SecondTrait
class CommonGreeter extends SecondTrait with FirstTrait
```

In this example, objects of the class `Greeter` would `greet` with `Hello You!` and instances of `CommonGreeter` would `greet` with `Hello World!` since then `FirstTrait` would override the method introduced by `SecondTrait`.

Mixins of Ruby

Ruby is a dynamically-typed object-oriented programming language which was designed to focus on the user (the programmer) and not the executor (the computer). It has all the common object-oriented features, but Ruby also has *Mixin*s as a solution for multiple inheritance.

Ruby has *module*s. A Ruby module is like a Ruby class. It holds method and constant declarations, it can have module and instance methods. Modules can be embedded into classes which will automatically "inherit" the methods from the module. It is not the classical inheritance however, it is hard to distinguish from it. All the "type-check" features of the language (like `kind_of` and `instance_of`) behave the same way. The main difference is that it can be done dynamically.

```
module Flying
    def fly
       puts "I am flying"
    end
end
class Eagle
   include Flying
end
class Seagull
   include Flying
end
def showUsage
   johnTheEagle = Eagle.new
   jonathanLivingstone = Seagull.new
   johnTheEagle.fly
   jonathanLivingstone.fly
end
showUsage
```

The above code prints two lines of: `I am flying`. Both the `Seagull` and `Eagle` classes have the same `fly` method inherited from a common ancestor.

```
module WoodWork
    def pickWorms
        puts "knock, knock, yum-yum"
    end
end
class CommonBird
    # Common bird stuff
end
class Eagle < CommonBird
end
```

```
class Woodpecker < CommonBird
   def learnWoodWork
      self.class.send(:include, WoodWork)
   end
end
def noWoodWork
   e = Eagle.new
   w = Woodpecker.new
   # All of the following lines are runtime errors
   e.pickWorms
   w.pickWorms
end
def loadingWoodWork
   a = Woodpecker.new
   b = Woodpecker.new
   # this line is a runtime error:
   # a.pickWorms
   b.learnWoodWork
   # this will work:
   b.pickWorms
   # now this works also:
   a.pickWorms
   # now it works with new instances also:
   c = Woodpecker.new
   c.pickWorms
end
noWoodWork
loadingWoodWork
```

In the above example there is not much difference between an `Eagle` and a `Woodpecker` except that a `Woodpecker` can learn to pick worms from a piece of wood. A common `Woodpecker` cannot do it by default, but after `learnWoodWork` method was called on any instances all the instances have the `pickWorms` method. But it can be more special:

```
module WoodWork
   def pickWorms
      puts "knock, knock, yum-yum"
   end
end
class Woodpecker
   def learnWoodWork
      class « self; include WoodWork; end;
   end
end
```

```
def loadingWoodWork
  a = Woodpecker.new
  b = Woodpecker.new
  # this line is a runtime error:
  # a.pickWorms
  b.learnWoodWork
  # this will work:
  b.pickWorms
  # this still not works:
  # a.pickWorms
  # and it is neither applied for new instances:
  # c = Woodpecker.new
  # c.pickWorms
end
loadingWoodWork
```

In the above example only `b` will have the `pickWorm` method. Neither previously or later instantiated objects will have this method, but they can include the module as well. This dynamism can only be done by modules in Ruby. It is also possible to include multiple modules. If two modules have defined the same method than the method of the later included module will be accessible.

```
module LandLife
   def breath
      puts "I breath with lungs"
   end
end
module WaterLife
   def breath
      puts "I breath with gills"
   end
end
class Amphibian
   include LandLife
   include WaterLife
   def superBreath
      super.breath
   end
end
def testing
  a = Amphibian.new
  a.breath
  a.superBreath
end
testing
```

The above code will print: `I breath with gills` and `I breath with lungs`. The "inherited" method (`breath`) will print the first one, but we can access the other method definition with the `superBreath` method which calls the `super.breath` method. This shows that Ruby handles multiple method like ancestors and a method which is included later is like an overriding in a subclass. This module handling makes Ruby very flexible.

10.7.6 Interfaces

The *interface* is a special abstract class. There are no instance variables in it, and the methods are only declared not defined. (Sometimes interfaces can hold class constants.) The interface – as its name suggests – describes the concept of the interaction border of the object. This is a new abstraction level in the program: we can ignore the concrete implementation and we can focus to the design concept. (And it also increases changeability.)

The real usage of the interface is through its implementation. A class implements an interface if it gives a definition to all if its methods. This makes the abstract concept of the interface to a concrete class. The variables with the type of the interface can hold references to any implementers.

Interfaces can inherit between each other. Interfaces can be ordered to an inheritance hierarchy as well. Even multiple inheritance (See Section 10.7.5.) is quite straightforward because every method body is missing and there are no member variables. A class can implement any number of interfaces. If there is a function which can be separated into two separate responsibilities than it is a good design if we separate the methods of the responsibilities, put them into separate abstract classes and then inherit and implement both of them in a class.

Protocols in Objective-C

Objective-C is an extension of C following the concepts of Smalltalk [PW91]. In the language we have single inheritance and protocols.

A protocol is a collection of abstract methods. If a class adopts a protocol then it implements all of its methods. If a class implements all of the methods in a protocol we say it conforms to a protocol. A protocol can be considered as an abstract class without member variables.

For example we can define a protocol that describes that it can be saved or loaded:

```
@protocol Archivable
-read: (FILE*) f;
-write: (FILE*) f;
@end
```

We can express that the *Shape* class adopts this protocol the following way:

```
@interface Shape: Object
  <Archivable>
```

Similarly we can prepare the reference counter protocol:

```
@protocol ReferenceCounter
- increase;
- decrease;
- (unsigned) references;
@end
```

One class can adopt multiple protocols:

```
@interface Shape: Object
  <Archivable, ReferenceCounter>
```

This way we have introduced multiple inheritance. In Objective-C this can be only done with protocols.

Multiple inheritance is allowed between protocols, and a class which adopts a protocol from an inheritance hierarchy means that it conforms to all of the base protocols of the protocol.

Interface in Java

In Java, interface means a new reference type which is a place to declare abstract methods and constant values.

An interface is always abstract, we do not have to note it, but we have to declare the interface *public* or it will only be visible in the package.

We can declare an interface in the following way [Nyek08]:

```
public interface Drawable {
   public void draw();
}

public interface Sortable {
   public boolean lessThan(Sortable s);
   public boolean equalsTo(Sortable s);
}
```

There is inheritance between interfaces, which is called extending in Java. We could extend the *Drawable* interface the following way:

```
public interface ColorDrawable extends Drawable {
    public int Color = 2;
    public int whatColor();
}
```

The *ColorDrawable* interface has *COLOR* constant and two declared methods (*draw* and *whatColor*).
There is multiple inheritance between interfaces.

```
public interface ColorSortable extends ColorDrawable, Sortable {
    int SIZE = 1;
}
```

A class can inherit from an interface with the *implement* keyword when all the inherited methods should be defined:

```
class Shape implements ColorDrawable {
  public void draw() { . . . }
  public int whatColor() { . . . }
}
```

The standard library of Java contains several interfaces. They are very important for graphical contents and concurrent programming. Even the *java.util* package has lots of interfaces in Collection Framework [Nyek08].

Object Pascal

Interfaces in Object Pascal are collections of abstract methods and attributes. Every interface must have a unique identifier. Every interface is a descendant of *IUnknown* interface by default, but we can define any interface hierarchy. The methods of *IUnknown* are: *QueryInterface*, *_AddRef* and *_Release* which are implemented in *System* unit in *TInterfacedObject* class. It is advisable to derive our interface implementer classes from this class.

For example ([Can00]) we can introduce the *ISortable* and the *IDrawable* interfaces (it is a convention to start every interface name with *I*):

```
type
  ISortable = interface
    [. . .]
    function LessThan(a : ISortable): Boolean; virtual;
    function Equals(a : ISortable): Boolean; virtual;
  end;
 IDrawable = interface
   [. . .]
   procedure Draw; virtual;
 end;
```

We set the identifier between the *[. . .]* tags and also specify the *LessThan*, *Equals*, *Draw* methods:

We can introduce the class *TShape* by deriving the *TInterfacedObject* and with the implementation of *IDrawable* interface:

```
type
  TShape = class (TInterfacedObject, IDrawable)
   public
    procedure Draw; virtual;
    ...
  end;
```

A class can implement more than one interfaces.[16]

```
type
  TSortableShape = class (TInterfacedObject, IDrawable, ISortable)
   public
    procedure Draw; virtual;
    function LessThan(a : ISortable): Boolean; virtual;
    function Equals(a : ISortable): Boolean; virtual;
    ...
  end;
```

The implementer class can be abstract. In this case we have to mark the abstract methods with the **abstract** keyword.

C#

The interface solutions of C# are similar to Java, but we cannot define constants in an interface. Interfaces can hold property declarations, indexers and event listeners [Csref03]. Multiple inheritance is allowed between interfaces.

```
interface IAinterface {
   void MethA1();
   void MethA2();
}
interface IBinterface {
   void MethB1();
   int X { // Property.
      get;
      set;
   }
}
interface IA_Binterface: IAinterface, IBinterface {
   void f();
   void g();
}
```

[16] If they have similar method names we can rename them [Del99].

Interfaces can be implemented by records and classes, which should be noted in the inheritance clause. If the implementer class declares a base class we have to write it in the first place.

```
class AClass: BaseClass, IAinterface, IBinterface {
   private int myX;
   // Implementation of methods ...
   public int X {      // Implementation of property.
      get { return myX; }
      set { myX = value; }
   }
}
```

Interfaces in C# define an interface to help the developer to handle objects uniformly by their behavioral similarities.

Traits in Scala

The first and most important difference of traits in Scala to interface of Java is that a *trait* can hold default implementation of methods. (This feature will be available in Java 8.) The other important difference is that a Scala *trait* can *extend* a class. Usually if your class extends a base class and has a trait, than it is added to the class declaration with the *with* keyword. But if a class does not extend any classes than the trait should be a extended with *extend*. (Other traits still can be added with *with* keyword.)

A trait can also have a constructor, but cannot have any construction parameter. This means if a trait extends a class that class must have a parameterless constructor. If we define a class with multiple traits than our class will execute the constructors left-to-right. The constructor of the derived class will be executed after the constructor of the base class and traits.

```
trait FirstTrait {
   println("Constructor of FirstTrait")
}
trait SecondTrait {
   println("Constructor of SecondTrait")
}
trait ThirdTrait {
   println("Constructor of ThirdTrait")
}
class Extender extends FirstTrait with SecondTrait, ThirdTrait
```

Instantiation of `Extender` would print: `"Constructor of FirstTrait"`, `"Constructor of SecondTrait"`, `"Constructor of ThirdTrait"`. After these constructors would be the constructor of `Extender` executed.

Traits are excellent features in Scala to change between distinguish between class or type hierarchy and behavioral similarities. We do not have to make a common ancestor (class) for two independent types which share a functionality, and we do not have to duplicate the same behavior.

10.7.7 Nested classes, inner classes

Some programming languages allow to introduce classes inside other classes, and we call them *nested classes*.

In the following example ([Nyek08]) the class *List* has a nested class called *Element* which is invisible for the outside world, only the methods of *List* can use them:

```
public class List{
  private Element first;
  private static class Element {
    Object data;
    Element previous, next;
    Element (Object data, Element previous, Elem next){
      this.data = data;
      this.previous = previous;
      this.next = next;
    }
  }
  public void insert (Object data) { . . . }
  public void delete (Object data) { . . . }
  private void delete (Element element) { . . . }
  private Element search (Object data) { . . . }
}
```

In Java, we differentiate inner classes from nested classes. We call every class that was defined inside another class as nested class, but a nested class can be **static** and *non-static*. A non-static nested class is often called as inner class.

10.8 Working with classes and objects

It is not a priority of this book to give any programming, architectural or design advices but it can start lots of interesting thoughts what to do with the different language features. If we just start using language features, inheritance, module system, etc. than our application might fit to the logic of the language but will not be easily maintainable.

These principles and patterns are quite universal, and help to understand how and why use all the earlier mentioned features. This is not an almighty list

of object-oriented programming, because this book is not about that, but it can guide reader to the fields of common object-oriented mistakes and solutions.

10.8.1 The Roman Principle

The first thing to consider is so natural for every experienced programmer that they do not even consider it as a true principle, it is just part of the common knowledge. As Saint Ambrose said in the early medieval: "When in Rome, do as the Romans do"! This means that if we decide to work in a programming language than we have to accept the rules of that language. It is important because other (experienced) programmers would expect that from us, and they would provide us a code that fulfills the common concepts.

What does this cover? For example if we write our code in Python we should start a member name with one underscore (_) if we want to show the other developers that it was considered as private. In a language where we do not have any data hiding solution such a common rule is very important. If we ever see a code that directly manipulates variables with one leading underscore or directly calls methods like this on a variable than we can assume that code has more serious issues.

We should never mess up the concept of copy constructor in C++ and *clone()* in Java. It is quite obvious for (former) C++ developers to define copy constructors since nothing forbids it in Java to define a constructor that gets a previously instantiated object as a construction parameter. However in C++ a copy constructor is called automatically when we use assignment to a freshly defined variable (for example: *SomeClass newVariable = previouslyDefinedInstance;*). But it is only called if we assign an object and not just a pointer to an object. Every case in C++ when we do not use concrete objects but pointers to one it clearly shows the possibility of a polymorphic instance. (For example a code like this: *SomeClass newVariable = *pointerToInstance;* is a potential truncation of types.) But in Java it has no sign in code that we could make anything wrong since there everything is a pointer.

(Consider this code: *SuperClass sc =* **new** *SuperClass(subClassInstance);* here we convert a sub class instance into a super class instance, and we can have different method implementations and also invalid inner state.)

We can write code that is totally correct in the level of the type system, and cannot be found with static code analysis, but can behave erroneously in runtime. However *clone()* is a polymorphic solution for this problem.

This list could be endless. The most important thing about it we should consider is: it is not enough to know that a language has the same keywords that can even mean the same thing, we should also learn the logic and the hidden rules of the language to write maintainable code.

10.8.2 Testing doubles

It has came to light that the classical software development is not responsible enough. There is a huge gap between the idea of the client and the tools of the programmer which make it really hard involve the client to the procedure of software manufacture. This procedure is even more strange for the client if the solution will be written in an object-oriented programming language. The process of making a complex object-oriented software starts with architectural designs, object diagrams that help abstractions, than abstract classes and separation interfaces are defined and implemented. At this point month have passed, millions of lines of code has been written and nothing works yet.

To make this process more human friendly a lots of solution has been made: eXtreme Programming, Rapid Application Development, etc. All of them are based on testing that makes it possible to frequently change the code and the design without mistakenly alter an already well written behavior. These solutions make prototyping possible which is an easy way to get fast client feedback. To get any benefit of testing we have to run our tests frequently, in an ideal case in every code change. That is where testing doubles came handy.

The term ”test double” comes from Gerard Meszaros, who used it in *xUnit Test Patterns - Refactoring Test Code*, when could not find any name to a not real object in testing that was not taken already by any concept. He has borrowed it from Hollywood stunts. These doubles are – just like in Hollywood – to replace expensive objects in dangerous situations. How can an object be expensive? An object might need lots of memory to work. It might take lots of CPU time in its methods. Construction and destruction objects like these could take many resources. Why is a testing harness dangerous? In tests we directly create special cases (beside average use cases) where we know the system can fail. We separate every test case from each other which means we should build up the test system, and then we should tear it down.

In dynamic languages to make test doubles is not challenging. But it is in strongly typed languages. Why? What does a test double do? In short: nothing. Well of course, it does something: it replaces the original object. They could be categorized but these categories are not strict. *Martin Fowler* has the following categories:

- The most simple test double is *dummy* object. Dummy objects just sit there in parameter lists, and can accept method calls, but do not change anything in the workflow. They are like Null Objects – not null references – or placeholders.
- *Fake* object is an object that has some working implementation but this is not enough for a production code. For example it can be any hardwired data source instead a complicated – user friendly – implementation.
- Fowler calls *stub*s every object that has fixed answers in its methods. These are typically done with method overriding where every method is a one line long **return** value, or a simple empty implementation.

- An object that records every method call on it – time, order, parameter – is called *spy*. These can be very useful to test behavior of two collaborating objects.
- And finally he presents *mock*s, which have a preprogrammed protocol (method call order, required parameters, etc) and fail the test case as soon as one point of the protocol is missed. These are also used for behavior testing.

All of these features are only possible to use if we can have polymorphism, dynamic dispatching, etc. In languages (like C++) where we can only override methods that were marked for this they are quite hard to be used easily. In C++ we cannot extend all the classes, but only those that have a virtual destructor. If they do not have virtual destructors than in destruction only the destructor of the dynamic type will be called and we can have resource leak, we can violate type invariant, etc.

10.8.3 SOLID object hierarchy

SOLID as an object-oriented principle was first introduced by *Robert C. Martin*. This is an acronym that stands for five principles that – according to Martin – should be applied for all object-oriented software system. These are to make our system, flexible, maintainable and understandable. (And they also help to make it testable.)

Single Responsibility Principle

Single Responsibility Principle stats that every class should have only one responsibility and that responsibility should be entirely served by that class. This is not so easy to achieve as we think, since we usually represent real life things with classes which could lead to a very complex entity.

If we imagine a car, which could be represented with a *Car* class. If they only holds data of a car (like in a car sale system) it will be fine. If they take part in a driving simulation, than a single class can be few. Think about it: in a simulation we want to control the engine the steering, we want to set properties of the tyres and so on. This could lead us, to take the *Car* apart.

If we separate the different functions of the car into *BreakPedal*, *GasPedal*, *Engine*, *SteeringWheel*, *Tyre*, *WindScreen*, etc. than what would be a car? Can we remove *Car* class entirely? No, because it will be still needed to collect all of these objects into a single entity. It can be a *Mediator* which helps the parts to work together. When the *GasPedal* is pushed, it tells the *Engine* to accelerate through the *Car* and so on. The *Car* class should only have this mediator responsibility and nothing else.

Now we can think that such a complex thing as a car obviously can lead to such complicated architectures. But we can take simpler things, like a glass.

What responsibilities should a class that represents a glass have? It can hold information as capacity, content, color, etc. What can we do with it? We can fill it up. We can empty it. We can ask how much liquid does it contain. (Is it half-empty or half-full?)

But a glass can appear in a different context. A *Glass* class can represent a 3D model of a glass, and in this case it also has reflection, clarity, shadow, etc. If this 3D model also has to work in a physical context (falling, breaking, moving, etc) than it is getting more and more complex.

To identify whether our object has a single responsibility or more Martin advices to examine why should we change our object? If it should change for only one purpose than our class has a single responsibility otherwise not. If we have a set of functions that uses only a set of member variables, than this set could be a class. If we have more separate sets, than we might have more classes, and so on.

Open/Closed Principle

As Bertrand Meyer wrote in *Object-Oriented Software Construction* "software entities should be open for extension, but closed for modification". This sounds impossible but it is not. This principle only suggest us to make our system modular.

How can we make an object based system modular? We have to use abstraction and the power of polymorphism and dynamic dispatching. If all the "modules" of our system is accessed through a well defined interface than they can be replaced without touching the code that works with them.

For such a flexible implementation we have to ensure that the objects that are separated from the system are not instantiated directly else we cannot replace the "modules" freely. Every object creation should be made dynamical as well. This is achievable if we use the *Abstract Factory* design pattern.

Our code is initialized with an instance of any derived class of the *Abstract Factory* and we request all of our objects through that factory instance. This ensures the opportunity to build up our system with the correct "modules".

This shows us that every part of the system that is open should be totally separated from the working code. The instances of the "module" should come from an abstract class. We should handle the instances uniformly through their common interface. We should not make any type-checking differentiation the closed part of the system should be totally free from any information of the underlying implementation.

Every logic of the application that has to behave differently in case of different instances should be moved to the open parts of the system. Every lines of code that is inside the closed section of the code should keep behavior. Open/Closed Principle requires the closed part to be constant. Open parts of the system can expect some behavior of the closed part (like method invocation order) which

is a dangerous invisible dependency of the system that can result mysterious failures of the system.

Dynamic languages such as Ruby can solve these problems easily. Ruby has a built in module system which is useful to fully customize class features according to requirements, but it also has ”duck typing”. This funny name refers to *James Whitcomb Riley* who said ”When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.” In Ruby we can replace objects with each other in a context if both of the objects has all the methods that are used by the context with the same signature.

Strongly typed languages has to use inheritance for the same purpose, and they can only have contravariant parameters, and covariant return values. In C++ we have strong features to keep our system closed: we can only override methods that are marked virtual. In C# we can always overload methods – even non virtuals – but they would only take effect if they are referenced through their dynamic type. This closed behavior can be forced by such features as a closed class or method (like final classes and methods in Java, etc.)

Liskov Substitution Principle

The L in SOLID stands for Liskov Substitution Principle which was introduced by Barbra Liskov. She has laid down that any instance of a class should be freely replaced with any instances of any subclass of the class without altering any property of the system. It is not strictly defined what those properties are. It can be static correctness, correctness, result, etc.

Strongly typed languages usually provide us with static correctness. Every instance of a derived class has the same public interface as the base class had. Eiffel makes an exception since in Eiffel we can inherit from a class without inheriting its interface.

What else should we require? For example we can request that every overridden method should accept similar parameters and should return similar value. This is the most common application of this principle.

We can also make limitations to side effects of a method invocation. If we state that we cannot alter the side effects of any methods, than we force programmers to only extend classes or only implement abstract methods. This solutions helps to build a flat object hierarchy which makes our system more flexible.

Interface Segregation Principle

Interface Segregation Principle comes from *Robert C. Martin* who has designed a software for Xerox and realized that it became more and more expensive to change the system. In that architecture they had a central class that has played several roles between the well defined and separated modules of the system. However every time when a module needed to send a new type of message to

another this method call had to be driven through the central class. Every time when the central class was changed the whole system needed to be recompiled.

This phenomenon came from the static linking of C++ source files: once one of the *include*d sources were changed every source that was built on them – or just simply used them – needed to be recompiled to be able to link to the new content. Totally unrelated classes that had nothing to do with the source of the change needed to be recompiled, relinked, etc. which took a lots of time. This expensiveness made them to concentrate to the system design, and they have discovered the problem:

The problem was that the central class was *include*d directly to every source file. Every class had full access to every method that was no way close to their responsibilities. The problem of the physical linking led them to realize the problem of logical linking. The interface of the central object could be cut to different *role*s that could be declared by different (pure virtual) classes. These classes should be *include*d by the underlying modules. In this case the changes of the central class would only affect the classes that are logically involved in the change.

Beside linking Interface Segregation Principle has other advantages as well. If we design our system considering this principle than we get a more natural solution. Imagine that we are clients of a bank and we can handle our accounts through a website. The server which holds this website holds the access point of the director of the bank. A director can do totally different things than a client. It is possible that the same physical thing holds independent roles but we feel it better if they are accessed through different interfaces. If we would see that we can touch all the account in that bank through our interface we would not feel our money safe in that institute however all the clients can promise that they will be trustworthy we would prefer banks with higher security.

This is the case of large interfaces if a class has access to unrelated methods of an object our system can easily become fragile. Even if we do not separate the implementers to different classes we can always slice the interface to make our system less sensitive to changes.

Dependency Inversion Principle

Dependency Inversion Principle is a mixture of Dependency Injection Principle and Inversion of Control. This is a little ”marketing” to get SOLID as acronym.

Dependency Injection means that composite objects that do not represent the state of an object only grants access to some functionality should not be instantiated by the enclosing class but they should be included from outside. This injection is better to perform in creation time to ensure validity of type invariant.

These injected objects should not be referenced directly by the enclosing object only their (abstract) super type should be used by. This limitation gives us inversion of control.

Dependency Injection gives us modularity. If outer logic is not forced by the class but injected by its builder or context than our class will become more reusable. Imagine a calculation where the calculator should get input data, execute the logic and report every errors and results to the user. This class can get an *InformationPresenter* class as an outer dependency. This dependency should have a *showError(String)* and a *showMessage(String)* method. Behind this method everything can happen. If the *InformationPresenter* is a wrapper of command line messaging (writes to standard error and standard output) than we simply write messages to command line. If it is a wrapper of a complex network-based logging system than we will log our messages through a network. All the logic of message visualization is separated from the logic of our calculation.

Inversion of Control makes it possible to physically separate our application-logic from other logic. In the previous example if we would not have a common ancestor of the two different *InformationPresenter* classes than dependency injection would still be useful. We could have two variables (for example *logPresenter* and *simplePresenter*) we could centralize our information sowing logic to methods that checks whether we have logging or printing logic (for example which variable is not null) and use that to visualize information. Inversion of Control separates these two implementation from our calculation. When we directly use logging and change anything in log handling it means that our calculation class should be recompiled. If we have a common base type that is used to access information visualization objects than we can change the concrete classes freely if we do not change the base class.

We can represent dependencies as a directed graph where a class depends on an other class if it is directly using it (extending a class is a direct usage) with an abstract class we can flip direction. (That is why it is called inversion of control.)

This principle helps to build loosely coupled systems which is much more maintainable than hardwired ones.

10.8.4 The Law of Demeter

Karl Lieberherr and *Ian Holland* has first mentioned the Law of Demeter which was named after their object-oriented research project. (Demeter was the goddess of harvest in ancient Greek.) In this principle they have suggested that every method of every object should only talk to their friends and should never talk to strangers. This means that it is restricted which objects methods can they call. The Law of Demeter states that every method can freely invoke:

- methods of its own object
- methods of the direct composite objects of its own object
- methods of object that was created by the method
- methods of object that was given as input parameter to the method

To be honest the Law of Demeter only forbids to call any method on a return value. Why is it so bad? First of all it hard wires your software's architecture. If you have many calls like *a.getB().getC().getD().getE().doSomething()* than you always have to ensure that every *a* like objects have to have a *B* that has a *C* which has a *D* that has access to a *E* that can *doSomething()*. If we ever want to change *D* and from that it might not have *E* available than and returns **null** in case of *getE()* was called than we introduce run time mistakes in parts of system that are not a single bit related to *D*.

In a software system it is quite natural that objects has and provides information about their context. But this information should not be used like in the above example. Every object should work with its own context because it was designed that way. We have to build a system that sounds reasonable for any developer and not just to developers who has lots of knowledge of the architecture. This – of course – means that we have to use reasonable names for our classes, instances, methods and variables, and also means that the authority of the class should be possible to detect from its name.

Imagine the following line of code inside a class that is called *Teacher*:

getClass(0).getStudent(0).getFather().getCar().start()

First of all this line is way too long. Secondly why should a teacher of a *Student* be able to start the car of the father of the *Student*? We can say that references of the system is reasonable. A *Teacher* have access to the *Class*es that she teaches in. A *Class* knows the *Student*s that attends to it. A *Student* knows his parents. An *Adult* can have a *Car*. All these references sounds reasonable but we do not feel appropriate if a random teacher of our child just starts our car. If all these references sound reasonable we do not have any reason to get rid any of them. This means that a system which holds the possibility to write such a statements is not bad from the beginning. We just should not write such a line ever.

The problem is that it is usually not so clear who has right for what. Can an MP3 player class send message to the *pause* button or only the *pause* button can send message to the class to stop music? If we avoid these chain message calling than we end up with a question: We have to provide some paths from one object to another how can we do that?

The first answer is: we should have instant access to that object from the first one. That means that a *Teacher* can know the *Student*s directly not just the *Classes*. But should a *Teacher* have direct access to all parents who has a child that attends to any classes of the teacher? No but sometimes a teacher has to send some messages to the parents (like "Your child has behavioral disorder." or "Little Jenny has won the school programming competition.") but it is quite natural to do it through the child. It is more likely that the teacher gives a letter to the student that tells the information to the parents than she walks right up to the office of the mother to inform her.

If we follow the second rule that means new methods will appear on the interface of the *Student* some methods that helps teacher-parent communication.

The good side of this is that we can limit the rights of the *Teacher* class, but the bad side is that the interface of the student will grow. Well for this the Interface Segregation Principle gives a good solution.

The Law of Demeter also helps to find out responsibilities. If a *Teacher* has direct access to the *Car* of a parent when the whole system fits the Law of Demeter than it must have a reason. First they might share the car (the teacher of the child can also be a parent of the child), or the teacher is a driving teacher that teaches on the car of the family (maybe for financial reasons). The most important part of that is it helps to discover correlations and relations of the system.

10.9 Summary

In this chapter we have seen the disciplines of object-oriented programming and their implementations in some languages.

Object-oriented programming is the sum of communicating objects which are described with well-defined properties and responsibilities.

Objects are to model separated parts of the world. Objects are collected into classes by their properties. We have seen the implementation of the concept of class and object in different programming languages. We have seen the object and class diagrams which are common notations for these concepts. These diagrams help us to represent the connections and relationships of objects and classes without the concrete implementation solutions and language features.

Data members and methods are encapsulated together in a class. It is advised to forbid the direct access of the data members by other objects the inner state should only be manipulated by the methods of the class. This is supported by different data hiding solutions.

Classes can have members and methods that are not connected to instances of the class. These are the part of the class and they are assigned when the class is declared. These are called class variables and class methods.

The most important feature of object-oriented programming is that we can build class hierarchies by inheritance. During inheritance we can extend the base class with new variables and methods, and we can also redefine methods. In some programming languages we can even change the data hiding properties as well.

Inheritance calls for life substitution. Substitution means that the instances of the derived class can replace the instances of the base class. Polymorphism and dynamic dispatching makes substitution possible.

If we derive classes from abstract classes (or interfaces) we can implement the abstract methods differently. This solution simplifies program design and supports changeability.

Some languages support multiple inheritance which means that a class can have more than one direct base classes. Multiple inheritance has benefits and drawbacks. Most of these disadvantages can be avoided by using interfaces.

Object-oriented programming is the most popular paradigm nowadays. The main reason is that it supports code reuse, and by clear class hierarchies the development of reusable programs.

10.10 Exercises

Exercise 10.1. Describe the rules and suggestions for encapsulation!

Exercise 10.2. Give some problems where the use of the *friend* class is an advantage!

Exercise 10.3. Give problems from everyday life which illustrate inheritance!

Exercise 10.4. What are the advantages of a common ancestor in a language?

Exercise 10.5. Describe dynamic binding through an example!

Exercise 10.6. Compare the support of polymorphism in C++, Eiffel and Java!

Exercise 10.7. Compare the multiple inheritance in C++ with the possibilities of interfaces in Java!

Exercise 10.8. Compare the constructors and destructors of various programming languages!

Exercise 10.9. Compare the multiple inheritance in C++ and in Eiffel!

10.11 Useful tips

Tip 10.1. See encapsulation in Section 10.4.

Tip 10.2. See friend in Section 10.5.1.

Tip 10.3. Read useful information about inheritance in Section 10.7.

Tip 10.4. You can read about common ancestor in Section 10.7.4.

Tip 10.5. For information about polymorphism and dynamic binding see Section 10.7.2.

Tip 10.6. About multiple inheritance see Section 10.7.5.

Tip 10.7. Constructors and destructors are described thoroughly on page 477.

10.12 Solutions

Solution 10.1. The data and the methods that operate on the data should be placed in the same "module". This rule led to the idea of class. A class is a collection of data and methods over the data. Encapsulation helps to form and maintain type invariants. (Data hiding helps to ensure invariants.)

What data and method should be closed together into the same type? There is no straight answer. We can say that what logically belongs together should be handled together. But there comes the problem of names. Why should a Dog know its kennel, or certificates of the kennel? A Dog should be an entity of the world which knows its state (*isHungry()*, etc.) its services (*cache(Object)*, etc.). But a class which is called DogRegistry should hold all these information and should not care about whether the dog is currently hungry or not.

This means that we can decide what to encapsulate from the name of the class? No, it is not true. If we program an animal recording system we might want to record differently *Dogs* and *Cats* and *Horses*. Of course int this system we wont talk about actual animals, but their breed, owner, price, etc. In this case we will always have a *DogRegistry*, *CatRegistry*, etc. but it is not important to always write down this class is a registry (especially if they are all subclasses of an abstract *Registry* class). Noting registry all the time would lead to a cluttering code.

If we have a system where we have to have both information (breed of a dog and state of a dog) should we have all of them in the same *Dog* class? It is quite likely that the answer is no. These two things are two totally independent responsibilities, and should be separated. Some time we need them together, for this we should provide some connections between them and a way to find connected objects.

Should a *Dog* have a reference to its *Registry* and should a *Registry* have a reference to the actual animal? This would mean there is a ring in the object dependencies which is quite malicious. Than who should reference the other? The one which should have right to change information in the other. If we do want to have a *changeOwner* method on *Dog* than the puppy should know its registry. If we want a *giveInjection* method into the *Registry* than it should reach *Dog* directly. But it is also a valid answer to have a special linking class that provides some functionality, like an *inject(String)* method that gives the injection to the dog, and registers its name to the registry. This class can be the link between the two object, and all methods that can belong to both responsibilities should be in this class.

From this short example we can see that data can be separated by the methods, and methods should be kept with data. These two things are so strongly related that they have to be handled together for a coherent system. This is encapsulation.

Solution 10.2. The concept of friend class is a shortcut in C++ to solve similar issues like the selective visibility in Eiffel. The main difference is that Eiffel can select between inner variables and classes one-by-one, but C++ can only declare a class as a friend, and in this case every variable of that class is visible for the other class.

This shortcut is rarely needed, in the most cases it could be solved by some standard object-oriented pattern. It can be very useful for testing. If the unit tests of the class are collected into an other class (a test-class) and that test-class is a friend of the class than we can check inner state without exporting it to the public interface.

Other very useful feature to set I/O streams as friend, to write a language friendly I/O functionality.

Solution 10.3. Think about Uniform Vehicle Code where we can read about rules that apply to every vehicle on the road. But vehicles have special casts like cars and trucks. We have general rules for every vehicle like: they have to travel on a specific side of the road, they cannot cross the barrage line. We have special rules for trucks like they cannot go faster than 80 km/h even on highways. This classification is a perfect example of generalization and specialization.

Solution 10.4. Common ancestor makes possible to form general methods. In the early versions of Java they did not need parametric polymorphism to implement Collections. They could extend the List class (where we can store objects in a specific order) without defining what they expect from that object. When we want to do the same thing with everything regardless their services than we can make use of the common ancestor.

Solution 10.5. Imagine a little game made for small children. You can implement a general *Animal* class where you can define a general action what to do on mouse click. Like *getName*`()` `+ " says " +` *getSound*`()`. In the different subclasses we only have to override the *getName*`()` and *getSound*`()` methods to get a working implementation for mouse click.

```
abstract class Animal {
    public void onClick() {
        System.out.println(getName()+" says "+getSound());
    }
    protected abstract String getName();
    protected abstract String getSound();
}
class Cat extends Animal {
    protected String getName() {
        return "Cat";
    }
    protected String getSound() {
        return "meow-meow";
    }
}
class Dog extends Animal {
    protected String getName() {
        return "Dog";
    }
```

```
        protected String getSound() {
            return "woof-woof";
        }
    }
    public static void main(String[] args) {
        Animal[] animals = new Animal[] {new Cat(), new Dog()};
        for(Animal animal : animals) {
            animal.onClick();
        }
    }
```

This code prints: "Cat says meow-meow" and "Dog says woof-woof".

Solution 10.6. If we compare polymorphism of Java, C++ and Eiffel we can find that the polymorphism of Java is the most simple and common. If a class is a subclass of an other than we can reference it through a variable of the super class. (If the class implements an interface we can reference an instance through a variable with the type of the interface.)

In C++, we have the same behavior through pointers but it gets interesting when we assign an instance of the derived class to the variable of an ancestor class In this case type truncation happens and we lose all the new variables of the derived class, and we cannot access the methods of the subclass. If we call a method that was defined in the super class than no dynamic binding will happen.

In Eiffel, we can only reference our objects through pointers so we do not have to fear from type truncation like in the case of C++. But Eiffel gives the opportunity to delete a method from the interface of the class. This means that we can reference a sub-type instance with a super-type variable and call a method which is not available on the specific instance.

Solution 10.7. C++ has multiple inheritance in case of implemented methods. It means it can inherit implementation form multiple classes. In case it inherits a method from multiple ancestors it has to give an implementation for that method. The implementation can be a simple call for one of his ancestors. In C++ it can also inherit variables twice. It is really important to realize these situations otherwise our class can grow enormous and we can have methods that seem to work together but always operate on different variable.

In Java we can only inherit method declarations from interfaces which has to defined or marked as abstract.

Solution 10.8. Usually languages do not have such a straight forward destructor like C++ has. In C++ when a variable gets out of scope than it is destructed. We can specify how to do that. In languages like Java and C# we can define a method that will be called before the object is destructed but contrary to C++ we cannot know when will it be.

Most of the object oriented languages have a special method that has the same name as the class which has to be called to instantiate an object. There are languages like Python where we have a special method (*__init__()*) which is the constructor of the class. Eiffel gives the opportunity to declare any methods as a constructor.

Solution 10.9. If we compare the inheritance of C++ and Eiffel, we should start with C++ because it is simpler. C++ supports multiple inheritance. C++ has public, protected and private inheritance. In case of public inheritance we inherit the full interface of the class and we can access the protected and public parts of the ancestor. We inherit both interface and implementation. Private inheritance means that we inherit public and protected members as private members. This is also called implementation inheritance. If we use protected inheritance then we inherit all public and protected parts as protected member. This means that our derived classes can reach them but the outer world does not know about this inheritance.

Eiffel has all the possibilities. It can inherit from a class but can delete methods from its interface. It can inherit only implementation or implementation and interface together. Eiffel has very complex possibilities that even supports covariant polymorphism in method parameters.

11 Type parameters

Type parameters make it possible to write generic, flexible and reusable program components. This chapter introduces the idea of type parameters and polymorphism. A small overview is provided to the theory of polymorphism, by giving a possible classification of its types. The most prevalent properties of the defined kinds of polymorphism and the way how programming languages implement them are also discussed in the chapter.

During software development it is common to write solutions as generic as possible to develop components usable in a wide variety of programs. This is especially important in case of library development. In Section 7.1, it is shown that — even by using subprograms — this goal can be achieved in some cases, if good parameters are chosen. However, the method's applicability is limited by the fact that types cannot be passed as parameters to functions. Hence, it is not possible to write one subprogram which can work on two distinct, but similar types.

Type parameters are the solution to this problem.

A subprogram which has type parameters is called *polymorphic*. A polymorphic subprogram can handle various types of the same argument, and can be used in various type environments. Moreover, it is also possible to map different implementations for different type parameters. Polymorphic subprograms are sometimes called *templates*.

The benefit of using type parameters is that we can use the same solution for problems using two different, but sufficiently similar types. This approach results in higher reusability, so the usage of type parameters facilitates writing succinct and more exhaustively tested libraries, as well as solving whole families of problems, while keeping the effort of the programmer much lower than writing distinct solutions for the problems.

11.1 Control abstraction

When implementing an algorithm, it should work on several distinct types. For example, to implement a sorting on arrays, the only fact we have to assume is that there exists a function which can compare two elements of the array.

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

This is the signature of the implementation of the quicksort algorithm in the standard C library appeared in the C90 standard [C90]. The parameter *base* is an array of an unknown type, *nmemb* is the number of elements in the array, *size* is the size of one element in bytes and *compare* is a function to compare two elements of the array. Unfortunately, using function pointers is not always enough to solve the problem, furthermore, using **void** * is error prone and not desirable in recent implementations.

Type parameters of subprograms can be seen as a kind of *control abstraction*. That is, instead of writing subprograms specialized for different types, it is sufficient to write one general solution for all of them. It is still possible to write distinct subprograms for distinct base types, but it is a more graceful solution to use type parameters and parametrize the function by the element type of the array. The following function in Java gives an example.

```
public class Sorting {
    public static <T extends Comparable> void sort(T a[]) {
        for (int i=a.length-1; i>0; --i) {
            for (int j=0; j<i; ++j) {
                if (a[j].compareTo(a[j+1]) > 0) {
                    T tmp = a[j];// swapping a[j] and a[j+1]
                    a[j] = a[j+1];
                    a[j+1] = tmp;
                }
            }
        }
    }
}
```

The *sort* generic function is an implementation of the bubble sort algorithm. It is parametrized by the type variable T introduced by the notation <*T* **extends** *Comparable*>. Moreover, this means the parameter here must implement the interface *Comparable*. In this example, the interface *Comparable* can be defined as follows, but please note that this definition and the definition of the function *sort* will be refined later in this chapter.

```
public interface Comparable {
    int compareTo(Object o1);
}
```

That is the reason why we can use the method *compareTo* in the **if** statement. On the one hand, the definition of the generic ensures that T implements the interface *Comparable*, so it has a *compareTo* method, hence we can use it. On the other hand, whenever we use this generic function, the compiler checks that the T really implements *Comparable*.

The usage of this generic function does not differ from using a regular function, type parameters are deduced and checked against the requirements given in the template definition.

```
Integer a[] = {1,2,5,4,3,6};
Sorting.sort(a);
```

In C++, polymorphic functions are called template functions, and can be introduced by the keyword *template*. See the following function as an example [Str00].

```
template <class T>
void sort(vector<T>& v)
{
    const size_t n = v.size();

    for (int gap = n/2; 0 < gap; gap /= 2) {
        for (int i = gap; i < n; i++) {
            for (int j = i-gap; 0 <= j; j -= gap) {
                if (v[j + gap] < v[j]) {
                    T temp = v[j]; // swapping v[j] and v[j + gap]
                    v[j] = v[j + gap];
                    v[j + gap] = temp;
                }
            }
        }
    }
}
```

This example implements the *Shell sort* [Knu87] for arbitrary type T. This type can be anything which has an assignment operator (because of `T temp = v[j]`), copy constructor (`v[j] = v[j + gap]`) and the $<$ comparison operator. So the variable T can be replaced by, for instance, **int** or *string*. These operations are not specified yet, and their concrete meaning is not important at this point. Every type satisfying the mentioned requirements can be used without writing the function *sort* for each of them. Note that, unlike Java, these constraints of the type parameter are not specified in the signature.

Using the template function is as simple as in Java:

```
vector<int> v;
...
sort(v);
```

In functional languages, type parameters are rather natural and can be used without any special construct. For instance, in Haskell, type names must begin with an uppercase letter, so any type name in a type signature beginning with lowercase letter is a type parameter, just like *a* in the following example.

```
isort :: Ord a => [a] -> [a]
isort []  = []
isort (a:x) = insert a (isort x)


insert :: Ord a => a -> [a] -> [a]
insert e [] = [e]
insert e (x:xs)
   | e <= x   = e:x:xs
   | otherwise = x:insert e xs
```

In this example, there are two polymorphic functions defined. The function `isort` is an insertion sort, which uses the function `insert` to insert an element (`e`) to the right place into the ordered list (`x:xs`). In the function `insert` the operator `<=` is used to compare two elements, so there must be an instance of the type class `Ord` for the type `a`. Note that, like in Java, the constraints are explicitly written in the signature, but unlike imperative languages the type checker can infer the type of the function as well as the constraints in most of the cases.

In the aforementioned languages polymorphic subprograms worked just like regular subprograms, without the need of instantiating them. However, some languages, like Ada, require explicit instantiation.

```
generic
   type Element is private;
   type Index is (<>);
   type Vector is array (Index range <>) of Element;
   with function "<" (X, Y : Element ) return Boolean is <>;
procedure Sort (V : in out Vector );
```

Here, the specification of the procedure starts with the keyword **generic** to mark that this is a generic procedure, and after that come four parameters. A polymorphic subprogram can have multiple type parameters. However, in Ada the type parameters *Element* and *Index* are not enough, and the concrete type of the array is a parameter which is needed as well (i.e. *Vector*). This is because in Ada two arrays of the same index and element type are not necessarily the same, due to the nominative type equivalence.

The keyword **private** in the type specification of *Element* means that we have no assumptions about its concrete type (except, that it has assignment and equality comparison). *Index* must be discrete type, which fact is denoted by *(<>)*. The type *Vector* is an array of type *Element*, where the array is indexed by an interval of the type *Index*. The line beginning with the phrase **with function** `"<"` means that we need an operation to compare the elements of the array.

To use a generic subprogram in Ada we must instantiate it first (see Section 11.6), where the formal parameters of the generic are substituted with actual

generic parameters. In the example, an array type *Int_Vector* is created first, and then the subprogram *Sort* is instantiated with the name *Int_Sort*.

```
type Int_Vector is array (Integer range <>) of Integer;
procedure Int_Sort is new Sort(Integer, Integer, Int_Vector, "<");
```

After this the procedure *Int_Sort* can be used to sort an array of type *Int_Sort*. For example, if we have a variable named *A* of type *Int_Vector*, we can simply write the following.

```
Int_Sort(A);
```

11.2 Data abstraction

It is possible to give type parameters not only to subprograms, but even to types. Is is often desired to create generic, parametrized types as well. A common attribute is the possibility of formulating them without the exact knowledge of the actual type parameters. So the type is *abstract*, and the corresponding *concrete type* turns out only during instantiation – when we substitute formal parameters with concrete types. To demonstrate this, we will show how to implement the stack data type in various languages. The generic stack data structure will be implemented over an abstract type, and after instantiation it becomes a concrete stack of the given concrete type (see Section 11.6). The first example is in Java. In Java, parametric types are generic classes:

```
public class Stack <T> {
    private static class Node {
        Node(T data, Node next) {
            this.data = data;
            this.next = next;
        }
        Node next = null;
        T data = null;
    }

    private Node top = null;

    public void push(T t) {
        top = new Node(t, top);
    }
    public T top() {
        return top.data;
    }
    public void pop() {
        top = top.next;
    }
    public boolean isEmpty() {
        return top==null;
    }
}
```

Like the generic function, the term <T> introduces the type parameter, but here without restriction and it is situated after the class name. So, <T> can be any type (more precisely, *T* must extend *Object*, so it cannot be a primitive type), and it parametrizes not only one function, but the whole class. More than one parameter can be expressed as well, separating them by commas. The stack in this case is implemented using a linked list, one element of the list is represented by the nested class *Node*.

In C++, a similar solution can be given:

```
template<class T, int max>
class Stack {
  T data[max]; // The elements of the array are instantiated
  int size;    // using the default constructor.
public:
  Stack() : size(0) {}
  void push(const T& e) {
    // Copy constructor is used to put the element into the array.
    if (size < max) { data[size++] = e; }
    else { throw std::logic_error("Stack overflow"); }
  }
  T pop() { . . . }
  const T & top() const { . . . }
  bool isEmpty() const { . . . }
  bool isFull() const { . . . }
};
```

In C++, parametric types called *template* classes. Template classes are introduced by the keyword **template** like template functions, and they basically look like a regular class. This class has two template parameters, the type of the elements of the stack, and the maximal size of the stack. The second one is, however, not a type, but a value. The implementation relies on the default constructor and copy constructor of the type T, however, this restriction is not expressed in the template.

Parameter of a template can be a type (class or struct or even a primitive type), values of some types, and templates. Types and values can be used inside the template definition just like a regular type or a value. It is even possible to pass a function (as a value of a function type) as template parameter.

The specification of the stack in Ada can be written as:

```
generic
   type Element is private;
   Max : Integer;
package Stacks is
   type Stack is limited private;
```

```
   procedure Push (V : in out Stack; X : in Element );
      -- Putting an element to the top of the stack.
   procedure Pop ( V : in out Stack; E : out Element );
      -- Deleting the topmost element from the stack.
   function Top (V : Stack) return Element;
      -- Returns the value of the topmost element.
   function Is_Empty (V : Stack ) return Boolean;
      -- Returns whether the stack is empty.
   function Is_Full (V : Stack ) return Boolean;
      -- Returns whether the stack is full
   Empty, Full : exception;
      -- Exceptions are used in situations, when somebody tries to
      -- read the top of an empty list, or tries to put an element to
      -- a full list.

private
   type Elements is array (1..Max) of Element;

   type Stack is
      record
         Data : Elements;
         Size : Natural := 0;
      end record;

end Stacks;
```

Like in C++, the type is parametrized by the type of the values stored in the stack and the maximal size of the stack.[1] However, this examples defines a package, which contains the type stack. The stack type is opaque, as it is specified as **limited private**, which means we prohibited the assignment and equality check on the type *Stack* (see Section 6.3.2). The two exceptions are used to report when the user of the stack tries to access an element of an empty stack, or tries to put an element of a full stack (see Section 8). The only operations we can use are the subprograms defined in the public part of the package. The private part of the package contains the representation of the type *Stack*, but this part can only be used within the package. The representation is simple, the array *Data* holds the elements of the stack, and the index of the topmost element is stored in the field *Size*.

The corresponding package body contains an abstract implementation of the operation. Abstract, because the concrete type of the element is unknown, moreover, the type *Elements* is unknown, because the type of its elements are unknown.

[1] In Ada, a neat solution would be to use discriminated record and parametrize the type *Stack* by its size. Using generics instead is for demonstrational purposes.

```
package body Stacks is
   procedure Push (V: in out Stack; X: in Element) is
   begin
      if V.Size < Max then
         V.Size := V.Size + 1;
         V.Data(V.Size) := X;
      else
         raise Full;
            -- Trying to put an element to a full stack.
      end if;
   end Push;
   procedure Pop (V: in out Stack; E: out Element) is ...
   function Top (V : Stack) return Element is ...
   function Is_Empty (V: Stack) return Boolean is ...
   function Is_Full (V: Stack) return Boolean is ...
end Stacks;
```

Finally, in functional languages using type parameters in types is natural. Most functional languages use the concept of algebraic data types (see Section 15.4.2), which can be parametrized as easily as functions.

```
data List a = Nil | Cons a (List a)
```

This examples shows how to define a linked list in the language Haskell. Here `List` is a *type constructor* — applying it to a type yields a type. From type `Int` it creates a list of integers. A value of type List can be created by the two given data constructors: `Nil` and `Cons`. `Nil` is nullary and represents an empty list, while `Cons` has two: a value to be put into the front the list and the rest of the list. Type constructors are one form of *type polymorphism* in functional languages. In some languages, like Haskell, type constructors are even allowed to have parameters which are type constructors.

```
data Strange a = Strange (a Int)
```

It is easy to see that in this example, the parameter of the type `Strange` must be a type constructor, as it is applied to a type (namely `Int`).

Interestingly, just a special branch of functional languages (called *dependently typed* languages) allows the use of values as type parameters. These languages (ATS, Agda, Epigram) are most frequently used in proof assistants, and give an approach for proof-carrying code.

11.3 Polymorphism

Polymorphism is a rather general idea, so it is more convenient to find a classification of polymorphism and then discuss the properties of the classes one by

one. A lot of classifications have been developed, but this chapter covers only one created by Luca Cardelli and Peter Wegner [CW85]. In this classification, there are two major kinds of polymorphism, *universal* and *ad-hoc* polymorphism. There will be two subclasses of these major kinds as well. This classification can be seen on Figure 11.1.

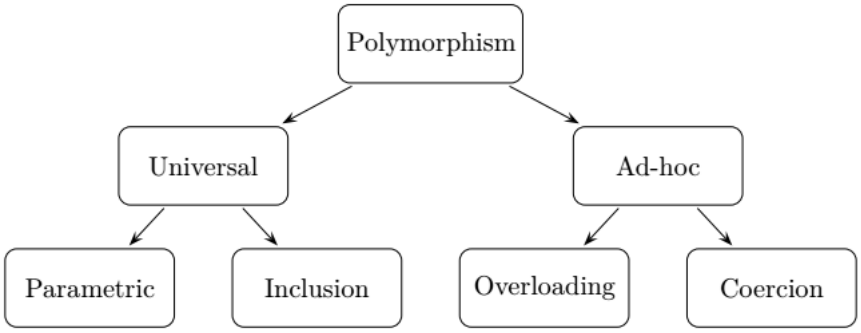


Figure 11.1: A possible classification of kinds of polymorphisms

Universally polymorphic program units usually work on an infinite number of types, where the elements share some common structures. For example, a function, which calculates the maximal element of a set should work on every type which has the comparison operator (`<`). The common structure, which restricts the set of acceptable types is the very existence of this operator. Therefore, this operator can work on potentially infinite number of types.

Why can we say that this function works on infinite number of types? The answer is that this function works not only on the types already defined, but for all the types defined in the future which has the necessary comparison operator. After defining the type *Person*, and defining the operator `<` to compare persons by age, we can surely use our parametric function with this type. It is easy to see that it is possible to define any number of new types, and the parametric function will work for all of them. The reason for this is the body of the function does not need to assume anything about the representation of the type, it is enough to know that there exists an operator `<` for that type.

The name 'universal' comes from the fact that that a universally polymorphic unit creates the same code for different type parameters, so there is a universal realization, which can be used for any type (with suitable structure). In contrast, *ad-hoc* polymorphism works on a finite number of types. For example, take a printer function. It is obvious that different types should be printed in different ways – for integer types, converting the number to a string should be enough, but for a record it is reasonably to apply some formatting:

```
Name:           John Smith,
Date of birth:  1990,
Place of birth: Anytown
```

So, it can be seen that we cannot share the same code for the two functions, and it is desired to give different implementations to different types. Because it is only possible to write an implementation for finite number of different types, we cannot state here that the function can work on an infinite number of types. In contrast to universal polymorphism, there is no common structure in the applicable types here, but these types are unrelated in most of the cases. There are distinct implementations for distinct types, which means it needs much more effort to implement, compared to universal polymorphism. But, there are also advantages, the programs specialized for the types are usually more effective than the abstract implementations.

The rest of the chapter is dedicated to discuss these kinds of polymorphism, but at first, let us summarize polymorphism support in certain languages. *Parametric polymorphism* is supported in most of the functional languages and some form of parametric polymorphism is supported in most of the modern languages (see Section 11.3.1). *Inclusion polymorphism* is a peculiar feature of object-oriented languages. *Overloading* and *coercing* is widely adopted and supported by most of the modern programming languages.

11.3.1 Parametric polymorphism

Parametric polymorphism is the most complete form of polymorphism. For example, the maximum search in the set mentioned earlier is parametrically polymorphic. Parametrically polymorphic program units work for all the types that can be matched to the type parameter. When applying, the types of *polymorphic arguments* (polymorphic argument can be anything which can be polymorphic in the body of the program unit, i.e. function parameters, types) are determined by the actual values of the type arguments. In the example above, polymorphic argument is the set, whose maximum is searched for. Its actual type depends on the type parameter, as it tells us the type of the elements set.

Depending on the language, type parameters can be *implicit* or *explicit*. It is called implicit, when the programmer does not have to provide the type parameters when the program unit is used. When the programmer must provide all the type parameters, then it is *explicit*. Usually, functional languages utilize implicit parameters, and non-functional languages mostly prefer explicit parameters. However, it is more and more common to have type inference (and hence implicit parameters) for generic functions and methods. To support parametric polymorphism, the language needs some kind of *unified representation of types* (e.g. using pointers for all the types), which can affect the performance in some cases.

Now, some example for parametric polymorphism is provided. For the sake of simplicity, we use list or array here instead of set. Its simplest implementation can probably be given in a functional language, such as in Haskell:

```
maximum :: Ord a => [a] -> a
maximum [x]  = x
maximum (x:xs)
   | x < m    = m
   | otherwise = x
   where
      m = maximum xs
```

This function works for every list, where the element type of the list is an instance of the `Ord` typeclass. The variable `a` in the type signature is a `type variable`, and it can be substituted by any type which implements the class `Ord`. The body of the function contains two cases, the first case is when the list has exactly one element, then it is the maximal element of the list. If the list contains more than one element, then the function checks whether the first element of the list is greater than the maximum of the rest of the list, if so, the first element is the maximum. The function calls itself recursively to calculate the maximum of the rest of the list. The following Ada generic subprogram provides the same functionality for arrays:

```
generic
  type Element is private;
  type Index is(<>);
  type Vector is array (Index range <>) of Elem;
  with function "<" (Left, Right: Element) return Boolean is <>;
function Maximum(V: Vector) return Element;
function Maximum(V: Vector) return Element is
   Max: Element := V(V'First);
begin
   for I in V'Range loop
      if Max < V(I) then
         Max := V(I);
      end if;
   end loop;
   return Max;
end;

with Text_Io, Maximum; use Text_Io;
procedure Maximum_Demo is
   type Int_Vector is array (Integer range <>) of Integer;
   function Int_Maximum is new Maximum(Integer, Integer, Int_Vector);
   A: Int_Vector(1..5) := (2, 8, 9, 23, 5);
begin
   Put("The maximal element of the vector is " &
        Integer'Image(Int_Maximum(A)));
end Maximum_demo;
```

This subprogram has three type parameters (*Element*, *Index* and *Vector*). To calculate the maximum of the array, we need the operation <. We also provided an example for the usage of the generic. It can be seen that we do not use the generic function *Maximum* directly, but we created a generic instance called *Int_Maximum* (see Section 11.6), which works on integers. The instantiation must be done for every actual case in Ada, the polymorphism is just a "syntactic sugar": we provide the body of the function only once, but it must be instantiated for every type.

In Java [Java13] — similar to Ada — the declaration can restrict the type parameters allowing only the types which have comparison, requirements like this are given by *interfaces*.

```
public class Max {
  public static <Element extends Comparable<Element> > Element
     max (Collection<Element> xs) {
        Iterator<Element> xi = xs.iterator();
        Element m = xi.next();
        while (xi.hasNext()) {      // loop while there is more elements
            Element x = xi.next(); // next element
            if (m.compareTo(x) < 0) { // comparing elements
                m = x;
            }
        }
        return m;
    }
}
```

Like in C++, there is no need to pass actual type parameters to the function *max* (see Section 11.6).

```
LinkedList<AType> list = new LinkedList<AType>();
// ...
AType listMax = Max.max(list);    // nem Max.max<AType>(list);
```

If the *max* function is called for the instantiated list, which implements the interface *Comparable*, we get a compiler error.

We have seen that the example written in functional language was simpler than the others. The reason for this is that modern functional languages contain parametric polymorphism by design, and it is not just a "syntactic sugar", like in Ada or in C++.

11.3.2 Inclusion polymorphism

In object-oriented languages it is common to have an object which belongs to more than one type. For instance, salmon, cod and tuna are fish, so a salmon not only belongs to the group of salmons, but the group of fish. A common property

of fish is the ability of swimming. If we define a swim operation, it should work on all fish, regardless of the actual fish type. An actual fish therefore can belong to more than one groups, it can be fish and salmon at the same time. So, tuna, cod and salmon are subtypes of fish, and the swim operation works on all of them. We call this *subtype polymorphism* or *inclusion polymorphism*.

Subtype polymorphism is similar to parametric polymorphism, but they differ enough both in theory and practice to discuss them separately. The key difference between them is that in case if parametric polymorphism, *the program unit itself is polymorphic*, while in case of subtype polymorphism, *the objects used by the program unit are polymorphic*. What makes this possible? In fact, in case of subtype polymorphism, an object is considered to belong to more than one class. These classes are subtypes of a common ancestor, so they have a common behavior. The abstract implementation of the program unit relies on these common methods. This is the polymorphism adopted by object oriented languages, where the subtype hierarchy is given by the inheritance hierarchy – as detailed in Chapter 10.[2]

Subtype polymorphism (like parametric polymorphism) is effective to solve general problems. In a given class hierarchy, it is common to define an *abstract base class* which contains the common methods and properties of the classes. This class contains abstract methods, so it cannot be instantiated. Its descendants must override all the ancestor's abstract methods, so the descendants can be instantiated. This way, the ancestor defines an abstract *interface* to handle the descendants (see Chapter 10.7.4). To implement the generic methods we can use the features provided by the common interface.

In C++, dynamic binding is provided by virtual functions, while interfaces are provided by *pure virtual classes*:

```
#include <iostream>
#include <algorithm> // for_each
#include <vector>
#include <functional> // mem_fun
struct Shape {
    virtual void draw() = 0; // virtual in all of the descendants
};
struct Circle : public Shape {
    void draw() { cout << "circle drawn."; };
};
struct Square : public Shape {
    void draw(){ cout << "square drawn."; };
};
```

[2] This is prevalent in current object-oriented languages, however, it causes serious problems in the type system of these languages. There are numerous languages – mostly for research purposes – which separate these two hierarchies, showing that it increases the safety of the language by reducing the possibilities of typing errors. For the sake of brevity, this book does not discuss this topic, more about this topic can be found in [Bru02].

```
void drawAll(vector<Shape*> shapes) {
        for_each(shapes.begin(),shapes.end(),mem_fun(&Shape::draw));
}
int main() {
    Shape* s1 = new Circle;
    Shape* s2 = new Square;
    s1->draw();             // call Circle::draw
    s2->draw();             // call Square::draw
    vector<Shape*> shapes;
    shapes.push_back(s1);
    shapes.push_back(s2);
    drawAll(shapes);
}
```

In this example, the *draw* method of the *Shape* class is pure virtual, so the method is *late bound* (e.g. it is looked up at run-time rather than compile-time). Purity is expressed by the value *0*, which also makes the class abstract and uninstantiable, but it can still be used as a type (a pointer to class *Shape*) in variable declarations. As the class cannot be instantiated, this pointer can only point to some descendant. These descendants override the method *draw*, so always the correct method will be invoked. In the example, the variables *s1* and *k2*, the methods of the classes *Square* and *Draw* are invoked, respectively. This fact is exploited by the method *drawAll* which gets a vector of shapes as parameter, and draws these shapes. The function *for_each* is a standard template library function [SL95], and iterates over the vector (the first iterator points to the first element, the second points to the element following the last element of the vector), and calls the function *Shape::draw()* on all of them, with the help of the STL function *mem_fun*. The example also demonstrates how well object-oriented and generic code complement each other.

In most of the object-oriented languages, such as Java, C#, Smalltalk, Eiffel, etc., the member functions are *virtual by default*, and always invoked according to the dynamic type.

A common use of subtype polymorphism is the implementation of container types. If there is a common ancestor class in the language (such as *Object* in Java or *ANY* in Eiffel), the implementation of the container can be built upon this class. As all other classes inherit this class, an arbitrary object can be put into the container. The drawbacks of this approach is that it usually cannot handle primitive types (In C#, despite using this approach, primitives can be used), and this approach cannot be applied if the languages do not have a common ancestor (like in C++).

In Eiffel, base class of all of the containers is the *CONTAINER[G]* generic class which defines the basic methods of containers, such as *has* and *empty*.

The language is designed to separate three different viewpoints, such as accessing, storing and iterating.

Accessing is the way the user of the container can access the elements stored in the container. In case of a stack or a queue, it is possible to access only one element, and there is no way to modify which element is the one accessed. In contrast an array or hash table, there should be an index or a key, which is used to access an element.

Storing is the way how containers are stored, how many elements they have, whether it is possible to change the number of elements. Most of the containers are finite, but there can be found infinite containers (e.g. sequence of prime numbers). Finite structures can be bounded or unbounded. For instance, *ARRAY[G]* which is the class for array of type *G* is bounded, and *LINKED_LIST[G]*, the implementations of the data type linked list is unbounded.

Iteration is responsible for the traversal of the collection, if it is possible to construct a traversal which reaches every element exactly once, in a predefined order. For instance, some collections can be sequentially traversed in one or two ways, moreover, tree data structures have pre-, post and inorder traversal.

The standard library of the language Eiffel provide one (and only one) class hierarchy for each of the aforementioned aspects. The root of the hierarchy of the accessing is the class *COLLECTION*, class *BOX* is the base for storage, and the iteration is provided by the descendants of the class *TRAVERSAL*. Both of these three classes are inherited from the class *CONTAINER*. The implementation of a new special collection – such as the class *LINKED_LIST[G]* – can be done by choosing a suitable class for all of the aspects above and by combining them using multiple inheritance. This way, we get a class which is characterized by its accessing, storage and iteration.[3]

```
deferred class SHAPE
 feature
  print_it
    deferred
  end
end
class CIRCLE
 inherit
   SHAPE
 feature
   print_it
    do
     io.putstring ("I AM A CIRCLE");
     io.new_line
   end
end
```

[3] This ultimately means multiple inheritance of the class *CONTAINER*.

```
class POLYGON
 inherit
   SHAPE
 feature
   print_it
    do
     io.putstring ("I AM A POLYGON");
     io.new_line
   end
end
class SQUARE
 inherit
   POLYGON
     redefine print_it
     end
 feature
   print_it
    do
     io.putstring ("I AM A RECTANGLE");
     io.new_line;
   end;
end



class TEST
 creation
   make
 feature
  t:  LINKED_LIST[SHAPE]
  o1, o2, o3: SHAPE
  make
   local i:INTEGER
   do
    !SQUARE! o1 -- Creating the objects.
    !CIRCLE! o2
    !POLYGON! o3
    !!t.make          -- Creating the linked list
    fill              -- Filling variable t
    from              -- Traversal of t
      t.start
     until
      t.off
     loop
```

```
        t.item.print_it
        t.forth
      end
    end
    fill
     do
      t.put_front(o1)
      t.put_front(o2)
      t.put_front(o3)
    end
end
```

In the example above, the class *LINKED_LIST[G]* is instantiated with the class *SHAPE* (which is abstract, as it is marked as **deferred**). The classes *CIRCLE* and *POLYGON* are descendants of the this class, and the feature *print_it* is already defined here. The class *SQUARE* inherits the class *POLYGON* and overrides the feature *print_it*. The variables *o1*, *o2* and *o3* are of type *SHAPE* and created by calling the appropriate constructor (*!SQUARE! o1*, *!CIRCLE! o2*, *!POLYGON! o3*), we assign them objects of type *SQUARE*, *CIRCLE* and *POLYGON*, respectively. The list of shapes, namely *t* is then filled by these objects, and we traverse the list with a loop. In the loop, the elements are printed out by calling the *t.item.print_it*, where the actual feature is chosen by the dynamic type of the current (*item*) element of the list.

11.3.3 Overloading polymorphism

In the introduction of the chapter, the example given for ad-hoc polymorphism (in Section 11.3) was actually an example of *overloading polymorphism*. The overloading polymorphism is in fact just a useful *syntactic sugar*: the same name can be used on subprograms working on different types, and the context (i.e. the types of the actual parameters of the subprogram) determines which subprogram is used. (Overloading subprograms has already been discussed in Section 7.6.)

One way to resolve overloading is to assign a distinct name for all the subprograms during preprocessing, and in the place of call, the overloaded name is replaced with the respective new one. This is exactly the case in CLU, if the program contains overloaded operators.

Comparing the ideas of the overloading polymorphism with the Ada `generic` or the C++ `template`, we can ask the question: Why have they not been categorized as overloading polymorphism? The answer is (which was already seen when ad-hoc polymorphism was discussed) that in case of ad-hoc polymorphic program units, distinct implementations are used for distinct types. In case of **generic** or C++ **template**, an abstract, universal implementation is given, which should work on all the types, hence it is a kind of universal polymorphism (most precisely parametric polymorphism as we have seen). Unfortunately, the compiler creates

more than one actual solutions, because there is no uniform representation of the types in Ada and C++ (see Section 11.3.1). So overloading polymorphism provides an abstract interface, without an abstract implementation, so it is only a limited form of polymorphism (with coercion polymorphism, see Section 11.3.4).

In Java, users can override only methods, but not operators. However, the built-in operators are overloaded. For instance, the operator `+` can be used on numbers as well as on strings:

```
2.0 + 2.0       // addition for doubles
"2.0" + "2.0" // string concatenation
```

In the standard library of C++, the *sort()* function uses the comparison operator `<` overloaded for the given type:

```
class Person { /*...*/ };

bool operator< (Person e1, Person e2)
{
    return e1.age() < e2.age();
}

int main()
{
    vector<Person> friends;
    std::sort(friends.begin(), friends.end()); // uses <(Person,Person)
}
```

Overloading polymorphism can be found in functional languages, as well. To achieve this, abstract interfaces are provided by typeclasses. For details, see Section 15.4.

11.3.4 Coercion polymorphism

Let us assume that we have defined the multiplication for real numbers. In this case, it would be useful to use the very same function for integers. Theoretically this cannot be a problem, as integers can be converted to reals without loss of precision,[4] this is what is called coercion polymorphism, if the operation is called with parameters of inappropriate types, the languages converts the parameters to the types required by the called operation.

Similar to the case of parametric and inclusion polymorphism, it is difficult to distinguish the overloading and the coercion polymorphism. However, overloading is actually a *syntactic abbreviation*, while coercion polymorphism is a *semantical operation*: the actual parameters are converted to the right

[4] In practice, however, the situation is more problematic. In Java, a `long` (long integer) is converted to `float` (single precision floating point number), but the first one is stored on 8 bytes, while the second is stored on 4 bytes, so this conversion must result some loss of data.

types, which are requested by the formal parameters. During this conversion the representation of the data is usually changed. If this conversion does not happen, a type error will be raised. Such a conversion happens in language C [Str00]:

```
double max(double x, double y)
{
  return (x > y) ? x : y;
}

int main()
{
  return max(42 * 42, 40.5 * 43);
}
```

Coercion polymorphism in C++ can work on user-defined types as well, if a constructor, which can be used for conversion, or a conversion operator is written. In C++ all constructors with one parameter are implicitly type converters unless declared "explicit" using the *explicit* keyword.

Another important example of coercion polymorphism can be found in C++. If we have a class *Derived* derived from the class *Base*, then value of type *Derived** can be assigned to a variable of type *Base**. In the background, the compiler automatically converts the pointer from *Derived** to *Base**.

In Java, coercion occurs also at method calls. Let us have a class *Base* which is a base of class *Derived*, let us also assume that there is a *C* class which has a the method with signature *m(Base)*. This method can be invoked by any object of type *Base* or its descendants. In the following code fragment, the compiler uses *implicit type cast* to the variable *derived* and converts it to the type *Base*, as the signature of the invoked method prescribed it [Rog01].

```
C c = new C();
Derived derived = new Derived();
c.m(derived);
```

Type cast can be done in two different manners. The first one is the *static type cast*, where the conversion operator is placed between the function and its parameter. The other case is when the parameters are checked *dynamically* at runtime.

We have mentioned earlier that the boundary between overriding and coercion polymorphism is obscured in some cases. Let us see an example for this [CW85].

```
3   + 4
3.0 + 4
3   + 4.0
3.0 + 4.0
```

Assuming that there is a + operation, which works for all the combinations appeared in the example above. Which polymorphism we have used to achieve this?

1. It is possible that the function is overloaded for these four different cases.
2. It is also possible, that there are only two overloaded alternatives for the function, one for integers and one for reals. When one of the arguments is integer and the other is real, the integer argument is converted to real, and the alternative with real parameters are called.
3. The third case is, when there is only one alternative, which has real parameters, and every integer parameter is converted to real.

Each of these case has its own advantages and disadvantages. It depends on the langauge, which polymorphism is realized, this question can only be answered for an actual language, with the knowledge of the actual definition of the language.

11.3.5 Implementation of polymorphism in monomorphic languages

Only a small variety of languages supports parametric polymorphism by their nature. These languages are usually from the ML language family, or similarly pure functional languages (Haskell, Clean) [CW85]. Subtype (inclusion) polymorphism can usually be found in object-oriented languages, like Eiffel, Java, C++, C# (see Chapter 10). These are the languages implementing some form of universal polymorphism, and their type system is called *polymorphic*. In contrast, *monomorphic type systems* allow only one behavior for an object. In case of parametric polymorphism, polymorphic program units have different behavior, i.e. they work for different types. In case of subtyping polymorphism, the objects themselves have different behavior, as it was seen in the fish example. Monomorphic systems do not allow such obviously diverse behavior, however, there are some exceptions in special cases. If a type system does not allow any form of polymorphism, it is called *strictly monomorphic*.

One advantage of strict monomorphism is that type errors can be found easily, at compile-time. But, the expressive power of these languages are heavily limited, as they do not allow the behavior of the objects to depend on the context. Therefore, monomorphic systems concede some limited polymorphism in some cases. These languages are called *mostly monomorphic*, which means that the language is basically monomorphic, but there are some cases where some level of polymorphism is allowed. Such a language is Pascal and C. Below, we list some of the polymorphic behavior of these languages:

- *Overloading* The same name can be used for distinct subprograms. For example, it is possible to define the operator + for matrices.
- *Coercion* For example, integer argument can be passed, where real is expected.

- *Subtyping* If a subtype is created from a type (for example, restricting the set of values of the type, see Section 5.8.1, elements of the subtype automatically belongs to the base type as well. So the subtype can be used everywhere where the base type is used.
- *Value sharing* A characteristic example is the **nil** constant, which can be used as a pointer to arbitrary type.

These are independent methods to lighten the monomorphism. Of course, even all the four cases can be found in the same language. These languages are not strictly monomorphic.

Let us see, how these exceptions fit into the aforementioned classification of polymorphism. In the first three cases, it is obvious which kind of polymorphism it belongs to, however these subtyping do not implement the whole variety of polymorphism discussed earlier. A subtype can be created by restricting the set of values of a type, or by extending a type by new features. These exceptions cover only the first kind of subtyping. The fourth case is a special case of parametric polymorphism. For the first sight, we could say that **nil** is an overloaded constant, but this is not the case: the constant **nil** can be used as a pointer for any type, even for types which we have not been defined yet. So, it must work on infinite number of types. Moreover, the representation of **nil** is the same, regardless of its type, which is uncommon in overloading. A very similar case is the class *NONE* of the language Eiffel, which descend from all the types (defined or not yet defined). The role of the value *Void* of type *NONE* is very similar to **nil**.

A careful reader can notice, that even if Ada (and C++) implement parametric polymorphism, they are not listed amongst the languages having parametrically polymorphic type system. The reason for this is that in these languages, parametric polymorphism is an exception, while in polymorphic languages, these are basic structures of the language. Another reason is the already mentioned *syntactical polymorphism*: Ada **generic**s and `template`s are actually just syntactical abbreviations to define program units with very similar body, and specialized code is created for every type they instantiate (because their semantics are macro substitution). This ultimately means that it is an ad-hoc polymorphism, so this polymorphism cannot be parametric polymorphism, as parametric polymorphism must be universal rather than ad-hoc.

11.4 Generic contract model

When using a polymorphic subprogram or type, a lot of requirements must be met. One must make sure that the actual parameters used during instantiation (or in case of functional languages the inferred types) qualify for the formal parameters. The method of agreeing the actual and formal parameters is discussed using the so called *generic contract model* [Nyek98].

Checking the legality of the declaration and body of polymorphic constructions (templates for short, that is polymorphic functions, abstract data types and packages) can be done during the compilation of the template. This makes possible to detect errors early, and illegal operations can be detected during the compilation of the template. Hence, during the instantiation, we can be sure that the template is legal of its own.

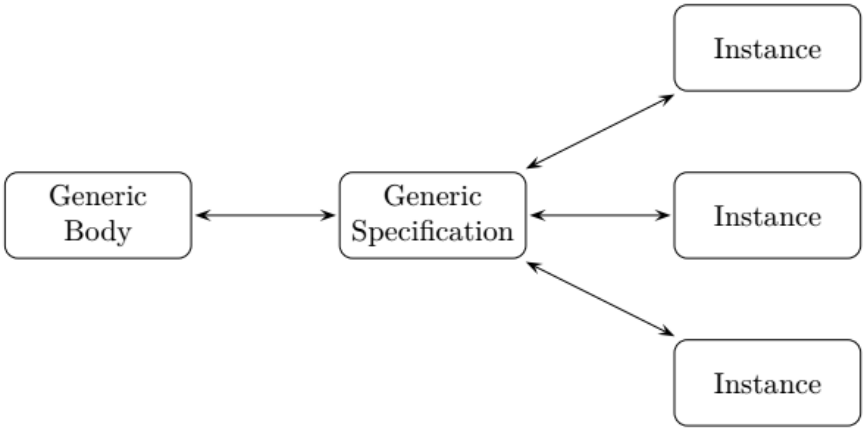


Figure 11.2: The Generic Contract Model

Why is the name generic contract model? Because the specification of the template implicitly writes a *contract* between the body of the template and the instantiation (see Figure 11.2). If the body of the template is legal, then it observes the implicit contract given by the specification. Thus, if the instantiation observes the same contract, the instance will be legal. This method makes it possible to detach the check of the template and its instantiation. Moreover, changing the template body without changing the contract (assuming that it is still legal) does not affect its instances.

When one of the parties violates the contract it is possible that problems emerge only at the time of instantiation, or even at runtime. To find the typing errors as soon as possible, it is clearly needed that all demands concerning to the instantiator should appear in the contract. For instance, such a demand can be the type class of the actual type parameter, restricting the usable types to a well-defined set. This means it is safe to use the operations of the specified type class, because if the actual type does not have the desired operation, the instantiator violates the contract, of which legality can be checked at compile time.

From the languages shown so far, it is clearly discoverable that some languages take generic contract model seriously, while others not. For example the checking of Java and Ada relies on contracts, however, C++ does not.

Using the contract model makes checking easy as well. The compiler assumes the *"worst possible"* case while checking the body of the template with respect to the specification. This is because if there is an operation used on a type in the body, but the operation is not enforced by the contract, then it is possible to instantiate the template with an actual type parameter which observes the contract, but does not have that operation. Hence, the independence of the template and the instantiation is not enforced. Then again, when checking the validity of the instantiation, the compiler should assume the *"best possible"* about the instantiation, that is, during the instantiation the actual parameter qualifies all the requirements of the contract. This is to ensure that checking the validity of the instantiation and the body can be done independently. Moreover, it is not a problem when the compiler is not familiar with the body during the instantiation of a template, because the specification is enough to do the validity checking.

In Java, when *T extends Comparable*, where *T* is a template parameter of a method, then this indicates to the body of the template that the methods of *Comparable* can be used, and it also indicates to the instantiator that this template can be instantiated only if the actual parameter implements *Comparable*.

In functional languages, generic contract model holds as well: when defining a function, all the information needed to restrict the type parameter can be given. This can be seen in Section 11.1: the type parameter *a* of both functions are restricted to be an instance of the type class *Ord*. This way, the compiler knows about the < operator of that type. In reverse, the compiler ensures that we cannot use the function when the type is not an instance of that type class.[5]

There are some languages where template instantiation does not work according to the generic contract model. In these languages, an improper template instantiation can cause a lot of inconvenience. For example, in C++ instantiating the *Stack* class template seen on page 568,if actual of the type parameter *T* does not have default constructor can result a compiler error in the default constructor of the class *Stack*, when the concrete call to the missing method occurs. This behavior can be most troublesome, when using a template class from the standard library. In case of misuse the compiler detects the error in the header of the standard library, resulting a rather cryptic error message. On the other hand this approach has some advantages as well.

The template *vector* requires its type parameter to have copy constructor. Failing to do so causes compiler errors in some of the member functions of the template, if we use them. The following code creates a vector of output streams.

```
int main() {
  std::vector<std::ostream> v;
  v.push_back(std::cout);
}
```

[5] In purely functional languages, the compiler is usually able to infer the types of functions, so in simple cases it is not necessary to write signatures for functions. Moreover, some languages, including Haskell, can also infer the type constraints for the type variables.

The type *ostream* has a base class *ios_base*, which is the base class for the entire stream library in C++. This class has a private copy constructor (How would it be possible to copy an open stream?), and this causes a compiler error when compiling some internal part of the template *vector*. It is also worth mentioning that if we do not use the member function *push_back*, the error does not appear. The reason for this will be discussed in Section 11.6.3

11.5 Generic parameters

Generic parameters can be of four different kinds: they can be objects, subprograms, types (and type classes) and modules. Different languages support different kinds of generic parameters. For example, Ada supports[6] all of the kinds, while Java only supports types as generic parameter. The rest of this section discusses the possibilities of these four kinds of parameters, the syntax of their specification and their implementation according to the generic contract model.

11.5.1 Type and type class

The most prevalent kind of generic parameter is the *type parameter*. The simplest case is when any type can be the type parameter. For instance, the function *length* takes a list as a parameter and returns the length of that list. The result does not depend on the elements of the list, moreover the elements of the list can be of any type. In the functional language Haskell (and Clean) the signature of the function can be as follows:

```
length :: [a] -> Int
```

A more difficult case is when we want to use an operation of the actual parameter in the generic body. According to the *generic contract model* we need to indicate this in the specification of the generic, so we need to give a restriction for the formal type parameter. In object-oriented languages a natural formalization of this restriction is to restrict the type parameter to subclasses of a class or an interface. Please recall the signature of the example in Java on page 564:

```
public class Sorting {
    public static <T extends Comparable> void sort(T a[]) {
        ...
    }
}
```

[6] Since Ada 95, it has been possible to create a pointer to a subprogram, so subprograms can be passed as regular parameters, while it is still possible to pass a subprogram as a generic parameter.

In the generic method *sort* of the class, the template specification of the generic parameters T tells us that the actual parameter of T must implement the interface *Comparable*. It means in the implementation of the generic, it is safe to refer to a method of the superclass as the contract guarantees that it exists. During instantiation, the compiler verifies whether the actual parameter matches the given restrictions.

The case of Eiffel is similar, to create a generic class *SET* with a type parameter G, where this parameter is restricted to be subclass of *COMPARABLE* can be defined as follows.

```
class
    SET [K -> COMPARABLE]
    ...
end
```

In C# as well:

```
public interface IComparable<in T>
{
  public int CompareTo(T other);
}

public class Set<T> where T : IComparable<T>
{
  ...
}
```

The type parameter of the generic class *Set* is restricted to implement the interface *IComparable*.[7] A careful reader will notice that the type parameter T used the type parameter T in its own contract (e.g. in *IComparable*<*T*>). This recursive generic specification (so called F–bounded quantification or recursively bounded quantification) can be also used in Java and Eiffel, but is not available in Ada.

Eiffel and C# allows to restrict type parameters with certain constructors. In Eiffel this means a type parameter can be restricted to have a creation feature called *make*:

```
class SET [K -> COMPARABLE create make end]
```

Similarly in C# it is possible to use a *Constructor constraint* to express that the type should have a default constructor:

```
public class Set<T> where T : new()
{}
```

[7] The meaning of the keyword '**in**' used in the definition of the type parameter T will be detailed in Section 11.7, and does not play any role here. It is used, because this is exactly the way how it appears in System package in C#.

It is also possible to give more than one restrictions to a type parameter. There is no multiple inheritance in Java and C#, so at most one of the constraints can be a class, but any number of interfaces can be given. In Java these restrictions are separated by ampersand and by comma in C# and Eiffel.

If a generic has more than one type parameter, they can constraint each other, however, only Eiffel allows to restrict a type parameter to itself (e.g. **class** *C[G -> G]*), but even in Eiffel it does not make much sense as it is the same as *G -> ANY*. Java and C# are more restrictive and does not allow to use cyclic constraints like:

```
public class C<U, V> where U : V where V : U
{}
```

Please note that the following code is not a cyclic constraint and is allowed in C#.

```
public class C<U, V> where U : V where V : IComparable<U>
{}
```

In Ada, it is possible to restrict a type parameter to a certain *typeclass*, such as to discrete types. This means, in the body of the generic, you can use the attributes specific to that typeclass, because the generic contract model ensures that the actual will have that attribute. The same happens when passing an array type. In the example on page 566,the generic parameters *Index* and *Vector* are good examples for that case. Here, *Index* is a discrete type, so in the implementation we can use the attributes `'Pred` and *'Succ* of this type.

In functional languages, similar result can be achieved by restricting a type to a certain typeclass.[8] To restrict type parameters by enforcing them to implement a certain typeclass has already been shown on page 565.Type classes are abstract interfaces and it is possible to create an implementation of the type class for any type.

A very similar idea in C++ is *concept*. In C++ there is no way to restrict a template parameter, hence misusing them can lead to cryptic error messages as we have seen on page 587.To overcome this problem C++ uses concepts. A concept is a set of requirements for a type, and it is used in the current and past C++ standards for documentational purposes. For instance, one of the simplest concept is *DefaultConstructible*, which means that the type should have a default constructor.

It is proposed to include concepts in C++0x [Sie05], but unfortunately it is finally left out. The idea is to create a formal description of the restrictions, for example the *DefaultConstructible* concept can be formulated as:

```
auto concept DefaultConstructible<typename T> {
    T::T();
};
```

[8] Typeclass has a bit different meaning in a functional language, see Section 15.4

This concept definition means that the type T meets the requirements of the *DefaultConstructible* concept if there exists a parameterless constructor for type T. The keyword **auto** tells the compiler that, every type which has a default constructor is automatically meets the requirements of the *DefaultConstructible* concept (similarly to structural type systems).

We can use concepts to revise our example from Section 11.2. To ensure that the type T has the used comparison operator, we can enforce that the parameter T should meet the requirements of the *LessThanComparable* concept.

```
template<std::LessThanComparable T, int max>
class Stack
{
    ...
};
```

Using concept, however, does not mean that the generic contract model, as the body of the template is not checked. It is that every function used in the body of the template is ensured by the concepts required in the template specification. It is basically a way to make an error message much less cryptic.

In some cases it is necessary to express properties which are not syntactical, but rather something behavioral. Such a concept in C++ is the concept of ForwardIterator.[9] Such concept does not contain syntactical restrictions, but cannot be automatically mapped. If the programmer knows that a type fulfills the requirements of a concept, it can be mapped explicitly. For example:

```
concept ForwardIterator<typename T>: InputIterator<T> {
};

concept_map ForwardIterator<MyIterator> {};
```

Concept maps can also be used to adapt a class to a concept, by explicitly giving a mapping from the requirements of the concepts. For example, we know that pointer types fulfill the requirements of ForwardIterators, but to be a ForwardIterator, one must fulfill the requirements of InputIterator. To be an InputIterator the type must contain a type named *value_type*, among the operators increment, decrement and dereference (++, – and *). To map pointer types to the concept of ForwardIterator, we can explicitly give the type *value_type*:

```
template<typename T>
concept_map ForwardIterator<T*> {
    typedef T value_type;
};
```

[9] ForwardIterator is an InputIterator, but it guarantees that copies of the iterator remains valid after the iterator is incremented. So an iterator which can be used in multipass algorithms. For example, an *istream_iterator* is not a ForwardIterator.

11.5.2 Template

In C++, a quite unique feature is the possibility of passing a template as template parameter. This can be done by passing its signature instead of the keyword template in the template definition.

```
template<template <typename> class F, typename T>
class C
{
    typedef F<T> applied;
};
```

Similarly, in Haskell, type constructors can have type constructor parameters as seen on page 571.

11.5.3 Subprogram

In Ada, it is possible to pass a subprogram as a generic parameter. It plays an important role to specify operations on a type parameter, just like in Section 11.1. In that example, we needed the operator $<$ to sort an array, so to ensure that two values of the specified type can be compared, we introduced a function parameter to be used as a comparison function. Moreover, in Ada 83 the only way to pass a subprogram was to use generics. The following example contains the specification of a generic function *Integrate*, which calculates the numerical integral of the function F. (You can compare this solution with the example from Section 7.3.1 on page 278,where a similar function can be found, but in Pascal.)

```
generic
  with function F(X: Float) return Float;
function Integral(Lower_Bound, Upper_Bound, Step: Float) return Float;
```

In Ada 95, it is possible to parametrize a subprogram with subprograms by using pointers to functions, similar to a lot of other languages. More details can be found in Section 7.3 (on page 278).

11.5.4 Object

It is possible to pass values and variables as generic parameters. This way, it is possible to create abstract data types or operations, which can work on types of various length. In Section 11.2 and 11.2, two implementations of the data structure Stack was shown, where the maximal size of the Stack was passed as generic parameter.

The following code shows the way how to pass a value and a variable as generic parameter in Ada.

```
with Text_IO; use Text_IO;
procedure Obj_P_Mod is
  generic
    E: in Integer;
    V: in out Integer;
  procedure G;
  procedure G is
  begin
    V := E * V;      -- E := E * V invalid, E is not an lvalue
  end G;
  X: Integer := 6;
  procedure P is new G(X, X); -- same as G(6, X)
begin
  X := X+1;
  P;
  Put_Line(Integer'Image(X)); -- X equals to 42
end;
```

The parameter E is an input value (shown by the keyword **in**), so we use its value. It means it is invalid to use E in the left hand side of an assignment. On the other hand the **in out** keywords in the specification of the parameter V means that its value can be modified in the body of the generic.

11.5.5 Module

It is common that a generic extends the services of another generic, or it simply just depends on the abstraction of another generic. A comfortable way to do this is to encapsulate the generic type and the operations over it (see Section 10.4) and pass it as a whole to the generic. In object-oriented languages (Java, C#, C++, Eiffel), it is a very natural way, because the tool of encapsulation is *class*, and these languages accept (and most of the accept only) classes as generic parameters.

In Ada 95 it is possible to do something similar. The basic construct of encapsulation in Ada is the package, so the desired effect can be done by parametrize generics by packages. Let us see an example for this!

```
generic
  type Elem is digits <>;
  type Index is (<>);
  type Matrix is array (Index range <>, Index range <>) of Elem;
package Matrix_Arithmetics is
  function "*" ( A, B: Matrix ) return Matrix;
```

```
...
end Matrix_Arithmetics;
package body Matrix_Arithmetics is ...

type Real is digits 5;
type Matrix is array (Integer range <>, Integer range <>) of Real;
package Real_MA is new Matrix_Arithmetics(Real,Integer,Matrix);
```

The package *Matrix_Arithmetics* is a basic generic to implement matrix operations, such as addition, multiplication, etc. To write a more complex package which implements *more complex* operations (like inverting matrices) it is possible to write the following.

```
generic
  type Elem is digits <>;
  type Index is (<>);
  type Matrix is array (Index range <>, Index range <>) of Elem;
  with function "*" ( A, B: Matrix ) return Matrix is <>;
  ...
package Advanced_Matrix_Operations is
  function Invert ( M: Matrix ) return Matrix;
...
```

The example shows us that all generic parameters of the used package and all the operations used are introduced as parameters of the new package. It is a rather elaborated generic definition and it can be cumbersome. Moreover, it does not form a semantic unit, which means at instantiation we do not need to use the operations of the package *Matrix_Arithmetics*, but it is possible to pass a quite different operation for the parameter * instead of multiplication. It is obvious that the use of this * operator in the body of the package *Advanced_Matrix_Operations* leads to undesired results. Using packages as generic parameters are the cure to this problem.

```
generic
  with package Arithmetics is new Matrix_Arithmetics (<>);
package Advanced_Matrix_Operations is
  function Invert ( M: Arithmetics.Matrix ) return Arithmetics.Matrix;
...
```

In this example, the specification of the formal package parameter contains the name of a generic package – the actual parameter must be an instance of this generic package.

The name of the formal package parameter (*Arithmetics* here) is a package, and not a generic. We can refer to its components, such as to *Arithmetics.*`"*"`, as well as to the formal parameters of the generic package (e.g. *Arithmetics.Matrix*). It is also possible to refer the without qualifying then with the package name:

```
generic
  with package Arithmetics is new Matrix_Arithmetics (<>);
  use Arithmetics;
package Advanced_Matrix_Operations is
  function Invert ( M: Matrix ) return Matrix;
...
```

The notation *(<>)* in the specification of the formal package parameter means that any package can be passed to the formal parameter which is an instance of the generic prior to it.[10]

11.6 Instantiation

Prior to the use of a generic, we must know what *actual parameters* are substituted to the *formal parameters* . This substitution is called *instantiation*.

In pure functional languages (ML, Clean, Haskell, Miranda, etc.), it is not required to specify the actual type generic parameters while calling a polymorphic function, because the type inference system can determine them based on the context. In these languages, the term instantiation refers to the *instantiation of type classes to specific types* (see Section 15.4.1).

In some languages, it is required to instantiate the generic before use, without this the generic is unusable.

In instantiation, we mean the process where the compiler creates the necessary declarations and definitions needed to use the generic, by parameter substitution.

The compiler takes the identifiers in generic parameters and associates concrete declarations to them. This process is called *name binding*. The obtained function and classes are called *specializations*.

Instantiation can be classified in various ways. One possible point of view is when the compiler does the instantiation: at the actual usage of the generic (called *on-demand instantiation*) or when the programmer explicitly instructs the compiler to do so (called *explicit instantiation*).

According to another classification, the compiler can be *lazy* or *eager*. In case of a composite generic (e.g. a generic class) a lazy compiler does not generate code for every part of the generic, just the parts of the generic which is necessary to compile the code. An eager compiler generates code for the whole generic, regardless of their usage.

The third question is how the substitution of parameters are done during the instantiation – whether it is necessary for the programmer to provide all the parameters of the generic, or the compiler is able to infer them.

[10] It is possible give some restriction instead of `(<>)`.

11.6.1 Explicit instantiation

In Ada, instances must be declared explicitly and it is only possible to refer to these explicitly declared instances. It was shown before in this chapter in Section 11.1. It is also possible to instantiate a template explicitly in C++, however, this possibility is mostly used only to check the restrictions of templates [MS00].

11.6.2 On-demand instantiation

In C++, the compiler instantiates a template only when it is really necessary. For example:

```
Link<int> *pointer;    // There is no need to instantiate Link here.
Link<int> intlink;     // Here, the instantiation is unavoidable.
```

Declaring a pointer (e.g. the variable *pointer*), the type which the pointer refers to (*Link*<**int**> in our case) can be incomplete, hence during the declaration of the variable *pointer* the template *Link* is not instantiated. The first place where the definition of the template is needed is called *point of instantiation*. In the example above, this is the declaration of the variable *intlink*.

In Eiffel, instantiation works in a similar manner.

In Java, the *type erasure* process of the compiler transforms generics into raw types, so technically there is no generic instantiation. The reason for this is that in Java, generics are used to perform the runtime type checks at compile-time, and no specialized code is generated when using generics.

11.6.3 Lazy instantiation

In C++, instantiation is *lazy*, the compiler does not produce the whole template instance if it is not necessary.

```
template <class T> class List {
    int size() const;
    void sort(); // operator < is used here
    ...
};
class NoComp
{ /* no operator < is defined for this class */ };
void f(List<NoComp>& ln, List<string>& ls)
{
    std::cout << ln.size();
    std::cout << ls.size();
    ls.sort();
}
```

The compiler creates the function *List*<*NoComp*>*::size*, *List*<*string*>*::size* and *List*<*string*>*::sort*, because only these are referenced in function *f*. The function

List<NoComp>::sort is, however, not created, because the function *f* does not refer to it. This is fortunate, because the function *List<NoComp>::sort* is invalid, as there is no operator < defined on the class *NoComp*, which is used in the *sort*. Note that generic contract model is violated here, during the instantiation of the template *List* with the class *NoComp*, the compiler does not check whether the actual parameter meets all the requirements the body of the template raises. The requirements are check only, when the actual function is called, that is, when the actual function is instantiated. The approach of C++ is to enable the instantiation of a template with a type which does not conform to the requirements, while no function is used which needs the unmet requirement.

Weak and strong typing

C++ provides *weak typing* in a strongly typed language [Eck00], as the template does not demand its parameters to have an exact type. Instead, C++ requires the members used by the template to be available in the types given as actual template parameters. This way, templates are more flexible. In Python and Smalltalk, a method call is weakly typed, so there is no need for **template**s. On the other hand, Ada, Java and C# can explicitly restrict their template parameters, so in these languages *strong typing* is applied. The benefit of this is that according to the generic contract model, errors in the template emerge early, in the compilation of the template, and not just at the first use of them.

11.6.4 Generic parameter matching

Matching generic arguments is much like matching parameters of subprograms (see Chapter 7). The most common way is to match by position, but some languages (including Ada) allows to match by name. In some languages, like Ada and C++, we can provide default values for parameters as well.

The language Ada uses *explicit* instantiation, meaning that during the instantiation of the generic, a new package or subprogram is created, with the passed actuals. After that, the newly created program unit can be used.

Some other languages do not requires explicit instantiation, but is enough to provide the type parameters at the place of use. In C++, it is only necessary to provide the parameters for class template which does not have default values. Parameters of function templates are inferred by the compiler (if it is possible).

```
template <class T>
void swap(T& t1, T& t2) { /* ... */ }
int main() {
    int i = 5, j = 7;
    swap(i, j);     // calling swap<int>
}
```

However, the programmer should be careful, as the compiler does not take type conversions into account. Here is an example to demonstrate this:

```
template<class T> T max(T,T);
void f() {//...
    max(3.14, 1.59);    // ok: max<double>(3.14, 1.59)
    max(3.14, 8);       // error: max<?>(double, int) does not exists
}
```

In such cases, it is still possible to explicitly provide the type parameter, so the previous example can be rewritten:

```
max<double>(3.14, 8); // max<double>(3.14, double(8))
```

In Java, the situation is similar, but the compiler tries to find the common super type for the arguments. In the example below, the type B and C are subtype of type A, so the common super type of B and C is A which conforms to the declared bound. These bounds (namely `? super T`) are detailed in Section 11.7.

```
public class Gen {
    public static <T extends Comparable<? super T> > T
            max(T t1, T t2) {
        return t1.compareTo(t2)>0?t1:t2;
    }
    public static class A implements Comparable<A> {
        public int compareTo(A a) { return 0; }
    }
    public static class B extends A {}
    public static class C extends A {}

    public static void main(String args[]) {
        System.out.println(max(2,3));
        //System.out.println(max(2,3.14)); // does not work
        System.out.println(max(new B(), new C()));
    }
}
```

11.6.5 Specialization

An advantage of using C++ *template* is that the compiler does not generate code for unused parts, because of the lazy and on-demand instantiation. In some cases the generated code can be reduced even more.

```
template <class T>
class Vector {};
```

```
Vector<int> v1;
Vector<bool> v2;
Vector<AClass> v3;
Vector<A*> v4;
Vector<B*> v5;
```

In this example, 6 classes are created. To avoid the creation of all 6 classes, it is possible, to provide different *specializations* for different type parameters. Using inline functions and inheritance it is possible to create one class handle all the types *Vector*<*T**> [Str00].

```
template<>
class Vector<void*> {
    void** p;
    void* operator[](int i); /* ... */
};

template<class T>
class Vector<T*> : private Vector<void*> {
    typedef Vector<void*> Base;
    Vector():Base() {}
    explicit Vector(int i): Base(i) {}
    T*& operator[](int i) {
      return reinterpret_cast<T*&>(Base::operator[](i));
    }
};
```

With this solution, the compiler reuses the instance *Vector*<**void***> and reuses the generated code.

Specialization is also useful to provide specialized code for certain types. In this case, the specialized version covers the general solution, so when instantiating a template for a type which has a specialization, the compiler does not compile the original template, just the specialized one.

A common specialization in C++, is the specialization of the template *vector* to **bool**. A boolean variable can be true or false, so it contains exactly one bit of information. However, technically it is not possible to store it on less than one byte of space. The generic implementation therefore uses at least one byte of space for every boolean. With the use of template specialization, it is possible to give an implementation which stores eight booleans on one byte of memory.

In C++, specialization and weak typing makes it possible to execute programs at compile-time (called *metaprogramming*) [Unr94]. The idea is quite simple, conditional operations can be rewritten to specializations, and loops can be rewritten to recursive instantiation. To calculate the factorial of n at compile type, we can write the following in C++11:

```
template<int N>
struct Fact
{
    static constexpr long value = N * Fact<N-1>::value;
};
template<>
struct Fact<0>
{
    static constexpr long value = 1L;
};
```

The static field *value* in the struct *Fact<N>* provides the factorial of the template parameter N. The calculation is done by calculating the factorial of *N-1* and multiplying it by N. To stop the recursion, the template is specialized for $N = 0$, and the value 1 is provided for that case.

11.6.6 Type erasure

In C++, the compiler creates distinct codes for distinct type parameters. The problem with it is that the template cannot be compiled without its definition, so the whole template must be in a header file, with its implementation. This means that the implementation cannot be hidden (hence violating the encapsulation). It also results that all the translation units using our template contains the code generated from the template.

In Java, generics can be compiled and used just like any other Java classes. To achieve this Java employs a procedure called *type erasure* [BOSW98]. During type erasure, type parameters are substituted by their bounds, and the compiler generates code for that instance. To preserve type safety and polymorphism necessary type casts and bridge methods are inserted. Type erasure also ensures compatibility with old Java code without generic support. In practice, this results quite a strange behavior, see the following Java and C++ classes:

```
public class A <T> {
    public static int value;
}
template<typename T>
class A {
    static int value;
}
```

In case of C++, instances for distinct type parameters are distinct types. This means that *A<int>::value* and *A<long>::value* are two different variables, and assigning a value to one of them does not change the other. However, in Java, the opposite is true. Because of the type erasure, both *A<Integer>* and *A<Double>* is implemented in the same class file. To refer to the field, the *raw*

type (the name of the generic without arguments) must be used. It is not possible to use the generic parameters in static fields and methods. It is also impossible to implement the same generic interface with two distinct type parameters:

```
public class B implements Comparable<A>, Comparable<String> {
    // error: Comparable cannot be inherited with different arguments
}
```

The main disadvantage of using type erasure is, however, the lack of information about the type parameter at runtime. This means it is impossible to retrieve the generic arguments using reflection (just the raw type), and also means that it is not possible to use type parameters with the operator **new**:

```
public class C <T> {
    public T instantiate() {
        return new T(); // T is needed at runtime.
}  }
```

A frequently used workaround is to explicitly pass the *Class* object representing the type parameter to the function:

```
public class C <T> {
    public T instantiate(Class<T> klass) {
        return klass.newInstance();
}  }
```

The C# compiler generates CIL (Common Intermediate Language [CLI12]), which provides generics at virtual machine level, so there is no type erasure, and information of the generic arguments are available at runtime. It is also possible to instantiate type parameters using the operator **new**. Generics in C# can be compiled just like any other classes, but without the drawbacks of type erasure.

11.7 Generics and inheritance

Generics are useful to create new types. When A is a subtype of B, and T<A> is a subtype of T<B> then T is *covariant* in its type parameters. If T<B> is a subtype of T<A>, then it is called *contravariant*, otherwise it is called *invariant*. In some cases covariant subtyping allows insecure language constructs, by creating a security leak in the type system of the languages. A well known example is the array type in Java. Array is covariant, so *String[]* is a subtype of *Object[]*. Using this fact, the following example creates a type error, which cannot be detected by the compiler at compile time, but causes a run-time exception:

```
Object[] t = new String[1];
t[0] = new Object();
```

In Java, *Stack* is a subtype of *Vector*, but there is no subtype relation between the types *LinkedList<Vector>* and *LinkedList<Stack>*. On the other hand, *Stack<A>* is a subtype of *Vector<A>* and *Stack<Integer>* is a subtype of *Vector<Integer>*. So this subtyping is invariant.

There is no covariant subtyping in C++ [Str00]:

```
class Shape { /* */ };
class Circle : public Shape { /* */ };
class Triangle : public Shape { /* */ };
void f(set<Shape*>& s) {
    s.insert(new Triangle());
}
void g(set<Circle*>& s) {
    f(s);// Error: no conversion from set<Circle*>& to set<Shape*>&
}
```

However, it is useful sometimes to allow some form of covariant and contravariant subtyping in parameter and return types. Please recall the method sort written in Java. It is possible to introduce a type parameter in the interface *Comparable*.

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

A naive solution to rewrite the generic method sort to utilize the generic parameter of *Comparable* is the following:

```
public class Sorting {
    public static <T extends Comparable<T> > void sort(T a[]) {
        . . .
    }
}
```

However, this requires T to implement the interface comparable, but in this case it is enough to have a superclass which implements the interface. The right solution is to write:

```
public static <T extends Comparable<? super T>> void sort(T a[]). . .
```

It means that, T implements the *Comparable* interface, where the type parameter of the *Comparable* is some supertype of T, and we used the character `?` to express this (called *wildcard*). With wildcards it is possible to express co- and contravariancy.

To compare covariancy in arrays and covariancy in generics take the following examples. In case of type *List<?* **super** *T>*, we know that the elements of the list is a supertype of T, so it is safe to add an element of type T to the list. The

other case is the type *List*<? **extends** *T*>, than it is safe to assign an element of the list to a variable of type *T*, because we know that the elements of the list are subtypes of the type *T*. This second case is how arrays work. So it is always safe to get the elements of an array, but special care is needed when modifying them. To add elements to a list and also retrieve elements, the type parameter must be invariant: *List*<*T*>.

In C# co- and contravariant parameters can be specified directly in the generic declaration. In the following example, *U* is covariant, *V* is contravariant and *W* is invariant. To keep the safety of the type system, covariant type parameters can only be used in arguments, contravariant parameters can only be used in return types.

```
public class C<in U,out V, W> {...}
```

In Eiffel, polymorphism and inheritance are closely related, but because of the lack of method overriding an interesting situation arise.

```
class A[G] feature
    f(x: G) is ... end
end

class B inherit
    A[INTEGER]
end

class C inherit
    A[REAL]
end
```

The class *A* is polymorphic, and it has exactly one method called *F*. Classes *B* and *C* inherits the instances of class *A* for *INTEGER* and *REAL*. If there is a class *D* which inherits both *B* and *C*, it will have two conflicting declarations of the method *f*. To resolve such situations, the clause **select** can be used in (see Section 10.7.5).

```
class D
 inherit
  B
     rename f as bf
  end
  C
     rename f as cf
     select cf
  end
  ...
end
```

11.8 Summary

In the previous chapter we saw how the introduction of the idea of polymorphism creates an effective way of abstraction: it gives the theoretical foundations for *control* and *data abstraction*. This chapter introduced one of the possible classifications of the kinds of polymorphism, and we get *universal* and *ad-hoc* polymorphism. Universal polymorphism is further divided into *parametric* and *inclusion*, while ad-hoc polymorphism is divided into *overriding* and *coercion*.

Different languages provide different tools to achieve polymorphism. Purely functional and object-oriented languages fully support parametric and inclusion polymorphism, respectively. The languages Ada and C++ provide *syntactical parametric polymorphism*. We have also seen that *mostly monomorphic* languages can also give tools to provide any kind of polymorphism in a certain extent, and it is also discussed that *strictly monomorphic* languages provide less expressive power.

The *generic contract model* is introduced to make the body of the generics and the instantiation of the generic independent. The generic contract model also makes the way of modular software development easier.

We saw that the use of polymorphism increases the reliability and reusability of the source code, which makes is extremely useful for *library development*.

This chapter focused on how type parametricity is implemented in modern languages, what can be a type parameter, and how instantiation works on different languages. The most important differences of the implementation in distinct languages are presented as well.

The chapter also covered the interaction of type parameters with other language features such as subtyping. Most importantly, the benefits and risks of co- and contravariant type parameters are shown, and then it is detailed, how the arisen problems are handled in the most popular object-oriented languages.

11.9 Examples

This section provides three almost complete implementations of a linked list in three different languages. Throughout the implementations, we tried to follow the coding conventions and standards of the languages, however, we tried to keep the solutions simple, even at the expense of completeness or breaking the coding conventions. For the sake of readability, some error handling is left out (in Java and C#), similarly, our list does not have *Allocator* template parameter or any removal.

11.9.1 C++

```
template <typename T>
class list
{
```

```
private:
    struct node
    {
        T value;
        node * prev;
        node * next;
    };
    node * _first;
    node * _last;

public:
    list(): _first(0), _last(0) {}
    list(const list<T>& other): _first(0), _last(0)
    {
        for(node *it = other._first; it; it=it->next)
            push_back(it->value);
    }

    template<typename InputIterator>
    list(InputIterator first, InputIterator last)
    {
        for(;first != last; ++first)
            push_back(*first);
    }

    list<T> &
    operator = (const list<T> & rhs)
    {
        if(&rhs != this)
            list<T>(rhs).swap(*this);

        return *this;
    }

    void push_back(const T& t)
    {
        if(!_last)
            _first = _last = new node {t, 0, 0 };
        else
        {
            node * tmp = new node { t, _last, 0 };
            _last -> next = tmp;
            _last = tmp;
        }
    }

    void push_front(const T& t)
    {
        if(!_first)
            _first = _last = new node {t, 0, 0 };
        else
        {
            node * tmp = new node { t, 0, _first };
            _first -> prev = tmp;
            _first = tmp;
        }
    }

    void pop_front()
    {
        node * tmp = _first;
        _first = _first -> next;
        delete tmp;

        if(_first)
```

```
        _first -> prev = 0;
    else
        _last = 0;
}

void pop_back()
{
    node * tmp = _last;
    _last = _last -> prev;
    delete tmp;

    if(_last)
        _last -> next = 0;
    else
        _first = 0;
}

void swap(list<T> &other)
{
    std::swap(_first, other._first);
    std::swap(_last, other._last);
}

const T& back() const
{
    return _last->value;
}

const T& front() const
{
    return _first->value;
}

T& back()
{
    return _last->value;
}

T& front()
{
    return _first->value;
}

class iterator
{
public:
    iterator(node * ptr): _ptr(ptr) {}

    int operator != (const iterator &rhs)
    {
        return _ptr - rhs._ptr;
    }

    iterator& operator ++ ()
    {
        _ptr = _ptr -> next;
        return *this;
    }

    T & operator * () const
    {
        return _ptr -> value;
    }

private:
```

```
        node * _ptr;
    };

    iterator begin()
    {
        return _first;
    }

    iterator end()
    {
        return 0;
    }
};
```

11.9.2 Java

```
import java.util.AbstractSequentialList;
import java.util.Collection;
import java.util.List;
import java.util.ListIterator;


public class LList<E> extends AbstractSequentialList<E> implements List<E> {
    private static class Node<E> {
        E value;
        Node<E> prev;
        Node<E> next;

        public Node(E value, Node<E> prev, Node<E> next) {
            this.value = value;
            this.prev = prev;
            this.next = next;
        }
    }

    private Node<E> first=null, last=null;
    private int size=0;

    public LList() {
    }

    public LList(Collection<? extends E> coll) {
        for(E e : coll) {
            addAfter(last, e);
        }
    }

    private Node<E> getNode(int index) {
        Node<E> next = first;
        for(int i=0; i<index; ++i)
            next = next.next;
        return next;
    }

    private Node<E> addAfter(Node<E> before, E e) {
        Node<E> node;
        if(before == null) {
            node = new Node<E>(e, null, first);

            if(first != null)
                first.prev = node;
            if(last == null)
                last = node;
            first = node;
```

```
    } else {

        node = new Node<>(e, before, before.next);

        if(before.next == null)
            last = node;
        else
            before.next.prev = node;

        before.next = node;
    }

    ++size;
    return node;
}

private Node<E> addBefore(Node<E> after, E e) {
    if(after == null)
        return addAfter(last, e);
    else
        return addAfter(after.prev, e);
}

@Override
public boolean add(E e) {
    addAfter(last, e);
    return true;
}

@Override
public ListIterator<E> listIterator(int index) {
    return new Iterator(index);
}

@Override
public int size() {
    return size;
}

private class Iterator implements ListIterator<E> {
    private Node<E> next, lastReturned;
    int nextIndex;

    public Iterator(int index) {
        nextIndex = index;
        next = getNode(index);
    }

    @Override
    public boolean hasNext() {
        return nextIndex<size;
    }

    @Override
    public E next() {
        lastReturned = next;
        next = next.next;
        nextIndex++;
        return lastReturned.value;
    }

    @Override
    public boolean hasPrevious() {
        return nextIndex>0;
```

```
        }

        @Override
        public E previous() {
            if(next==null)
                next=lastReturned=last;
            else
                next=lastReturned=next.prev;
            nextIndex--;
            return lastReturned.value;
        }

        @Override
        public int nextIndex() {
            return nextIndex;
        }

        @Override
        public int previousIndex() {
            return nextIndex-1;
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }

        @Override
        public void set(E e) {
            lastReturned.value=e;
        }

        @Override
        public void add(E e) {
            if(lastReturned == next) {
                lastReturned = LList.this.addBefore(lastReturned, e);
            } else {
                lastReturned = LList.this.addAfter(lastReturned, e);
            }
            next=lastReturned.next;
            ++nextIndex;
        }
    }
}
```

11.9.3 C#

```
using System;
using System.Collections;
using System.Collections.Generic;

public class LinkedList<T> : ICollection<T>, IEnumerable<T>, ICollection, IEnumerable {
    private class Node {
        public T element;
        public Node prev, next;

        public Node(T element, Node prev, Node next) {
            this.element = element;
            this.prev = prev;
            this.next = next;
        }
    }

    public int Count { get; protected set; }
```

```
private Node first = null;
private Node last = null;

public LinkedList() {}

public LinkedList(IEnumerable<T> other) {
    foreach(T element in other)
        AddLast(element);
}

public bool IsReadOnly {
    get { return false; }
}

public bool IsSynchronized {
    get { return false; }
}

public object SyncRoot {
    get { return this; }
}

void ICollection<T>.Clear() {
    Count = 0;
    first = last = null;
}

void ICollection<T>.Add(T element) {
    AddLast(element);
}

public void AddLast(T element) {
    Count++;
    Node node = new Node(element, last, null);
    if(last == null)
        first = node;
    else
        last.next = node;
    last = node;
}

bool ICollection<T>.Contains(T element) {
    foreach(T t in this)
        if(object.Equals(t, element))
            return true;
    return false;
}

bool ICollection<T>.Remove(T element) {
    Node node = first;

    while(node != null && object.Equals(element, node.element))
        node = node.next;

    if(node == null)
        return false;

    if(node.next == null)
        last = node.prev;
    else
        node.next.prev = node.prev;

    if(node.prev == null)
        first = node.next;
    else
```

```
        node.prev.next = node.next;
    return true;
}

public void CopyTo(T[] array, int size) {
    IEnumerator<T> e = GetEnumerator();
    for(int i=0; i<size&&e.MoveNext(); ++i) {
        array[i]=e.Current;
    }
}

public void CopyTo(System.Array array, int size) {
    IEnumerator<T> e = GetEnumerator();
    for(int i=0; i<size&&e.MoveNext(); ++i) {
        array.SetValue(e.Current, i);
    }
}

IEnumerator<T> IEnumerable<T>.GetEnumerator() {
    return GetEnumerator();
}

IEnumerator IEnumerable.GetEnumerator() {
    return GetEnumerator();
}

public Enumerator GetEnumerator() {
    return new Enumerator(this);
}

public struct Enumerator : IEnumerator<T>, IEnumerator {
    private Node current;
    private LinkedList<T> list;

    public Enumerator(LinkedList<T> list) {
        this.list = list;
        this.current = null;
    }

    public T Current {
        get { return current.element; }
    }

    object IEnumerator.Current {
        get { return current.element; }
    }


    void IDisposable.Dispose() {
        if(list == null)
            throw new ObjectDisposedException(null);
        current = null;
        list = null;
    }

    public bool MoveNext() {
        if(current == null) {
            current = list.first;
            return true;
        }

        if(current.next != null) {
            current = current.next;
            return true;
        }
```

```
            return false;
        }

        void IEnumerator.Reset() {
            current = null;
        }
    }
}
```

11.9.4 Comparing the examples

It is interesting how these languages provide a solution to the same problem. The solution is similar, creating a generic class with a type parameter giving the element type of the list. This generic always have two nested classes: one to hold the elements and one to iterate over them.

It is important to note that only the language Java supports non-static nested (inner) classes (see Chapter 10). In other languages, inner classes can be emulated by passing the outer object as a constructor parameter – like the *iterator* in the C++ and *Enumerator* in the C# solution. In the case of the elements of the list (classes named *node* and *Node*), a careful reader would notice that only the the *Node* in the language Java has type parameter. The reason for this is that everything static belongs to the class itself (and thus the raw type in Java), in correspondence to Section 11.6.6.

Another interesting fact is the way how subtyping is handled. In C++, there is a constructor which accepts a range of elements of a container, which is represented by two iterators. Being a template, nothing is assumed about the container or about the type of the elements (except the elements are passed to the *push_back* method). In Java, there is a common superinterface called *Collection*, so there is a constructor which takes a collection of elements of type E or a subtype of E (i.e. *Collection*<? **extends** E>). In C#, the superinterface is called *IEnumerable*, but this interface does not contain any mutator methods, so its type parameter is covariant (**out**).

11.10 Excercises

Exercise 11.1. Compare universal and ad-hoc polymorphism.

Exercise 11.2. What are the benefits and drawbacks of strict monomorphism?

Exercise 11.3. Why is instantiation needed and how does it take place in different languages?

Exercise 11.4. What are the types of polymorphism and which languages employ them?

Exercise 11.5. Create a polymorphic method that swaps its two parameters of the same type.

Exercise 11.6. Is it possible to use covariant or contravariant type parameter for the arguments of the swap?

Exercise 11.7. What kind of polymorphism is practical for defining a comparison operator?

Exercise 11.8. Write a generic function, which takes a predicate and a collection, and returns the first element of the collection, for which the predicate is true.

Exercise 11.9. Why do we call the parametric polymorphism of Ada and C++ syntactic?

Exercise 11.10. How could we classify the languages by the manner of instantiation?

Exercise 11.11. In Java, there are three different ways to get a *Class*`<`*T*`>` instance for a type, where the type parameter *T* is the type the class represents. The first one is to call *getClass()* method of an object, for example `"foo".`*getClass()*. The second one is to append **.class** to the name of the type, e.g. **boolean.class**. The third one is to load the class, by giving its name: *Class.forName*`("java.lang.String")`. Each of them returns a *Class* instance, but with different type parameters. What are they?

Exercise 11.12. In a generic function, how can you instantiate an object of the type parameter?

11.11 Useful tips

Tip 11.1. Which kinds of polymorphism have an universal abstract implementation? In which kind of polymorphism will the polymorphic program unit work on types defined in the future?

Tip 11.2. Which is easier to implement? A monomorphic or a polymorphic language? Which has better expressive power?

Tip 11.3. When will the specializations created from the generic? Is there a language construct to create them?

Tip 11.4. Object-oriented and functional languages both have their usual kind of polymorphism.

Tip 11.5. The implementation language must modify the variables passed, so it cannot be implemented in some languages (such as in Java). A C++ solution is provided on page 11.6.4, so Ada or C# is preferred. The solutions should be a generic procedure or a generic static method.

Tip 11.6. Is it safe to swap variables of different types? Are they in- or output variables?

Tip 11.7. In C++, it is possible to create an operator `==` with new parameters, even if this operator is already defined for primitive types. What kind of polymorphism is this? How it is possible to define the comparison operator for templates?

Tip 11.8. In C++, use an iterator range, and a functor to define a template function. In Java, a generic interface *Predicate* should be created, and a static generic function should be created. In C# the solution is similar, and predicate can be implemented using a delegate. Take into account which parameter of the function has covariant and contravariant type parameter.

Tip 11.9. Is there a universal implementation of generics in C++?

Tip 11.10. Do we need to introduce the instance by hand? The new instance is compiled at once, or only the used parts are compiled?

Tip 11.11. The difference is what can we assume about the type that the class instance represents. This knowledge must be represented by wildcards.

Tip 11.12. It depends on the language, but we discussed the must important cases, in C#, Java, Ada and C++. In C# there is an explicit syntax to do it. In Java, there is no direct way to do that because of type erasure. In Ada, the type parameter must explicitly forbid this, and in C++ type parameters can be used just like regular types.

11.12 Solutions

Solution 11.1. In contrast to ad-hoc, universally polymorphic program units have an universal, abstract implementation. This implementation will cooperate with types defined in the future.

Solution 11.2. It is much easier to find type errors in a monomorphic language, but it has much less expressive power.

Solution 11.3. During instantiation, actual parameters are checked against the contract, and formal parameters are substituted. Instantiation is important, because usable program units are created from the generics. In Ada, a new program unit must be declared explicitly, before the generic instance can be used, but most of the languages uses implicit instantiation.

Solution 11.4. There are four types of polymorphism. Inclusion and overloading polymorphism is prevalent in object-oriented languages. Parametric polymorphism is very natural in functional languages, but it can be found in most of the modern object-oriented languages as well. Coercion polymorphism is unusual in functional languages.

Solution 11.5. *C#*

```
public static class Swapper {
  public static void Swap<T>(ref T a, ref T b) {
    T tmp = a;
    a = b;
    b = tmp;
  }

  public static void Main(string[] args) {
    int a = 1, b = 2;
    Swap(ref a, ref b);
    System.Console.WriteLine(a);
  }
}
```

Ada

swap.ads

```
generic
    type T is private;
procedure Swap(A, B : in out T);
```

swap.adb

```
procedure Swap(A, B : in out T) is
    Tmp : T := A;
begin
    A := B;
    B := Tmp;
end;
```

swaptest.adb

```
with Swap;
with Ada.Text_IO;
use Ada.Text_IO;

procedure Swaptest is
    procedure SwapI is new Swap(Integer);
    A : Integer := 1;
    B : Integer := 2;
begin
    SwapI(A, B);
    Put(Integer'Image(A)); New_Line;
end;
```

Solution 11.6. Input variables can be covariant, while output variables can be contravariant. The arguments of swap are in- and output variables, so the type parameter must be invariant.

Solution 11.7. To create a comparison operator for a new type, we should create a new operator with the same name, but with different parameter types, which is called overloading. In C++, this is possible:

```
struct complex {
  double re, im;
};

bool operator == (const complex &lhs, const complex &rhs)
{
  return lhs.re == rhs.re && lhs.im == rhs.im;
}
```

To do the same with a template class, we can use parametric polymorphism as well. For example to compare two lists:

```
template<typename T>
bool operator == (const std::list<T> &lhs, const std::list<T> &rhs)
{
  typename std::list<T>::const_iterator it1 = lhs.begin();
  typename std::list<T>::const_iterator it2 = rhs.begin();
  while(it1 != lhs.end() && it2 != rhs.end() && *it1 == *it2)
  {
    ++it1;
    ++it2;
  }

  return (it1 == lhs.end()) && (it2 == rhs.end());
}
```

Solution 11.8. *C++*

```
template<typename Predicate, typename InputIterator>
InputIterator findif(InputIterator first,
        InputIterator last, Predicate f)
{
    for(;first != last && !f(*first); ++first);
    return first;
}
```

Java

```
import java.util.Collection;
import java.util.Iterator;

public class Find {
    public interface Predicate<T> {
        public boolean call(T t);
    }

    public static <T> T findif(Collection<T> collection,
            Predicate<? super T> pred) {
        Iterator<T> it=collection.iterator();
        while(it.hasNext()) {
            T t = it.next();
            if(pred.call(t))
                return t;
        }
        return null;
    }
}
```

C#

```
using System.Collections.Generic;

public static class Find {
    public delegate bool Predicate<T>(T t);
    public static bool FindIf<T>(IEnumerable<T> list,
          Predicate<T> match, ref T result) {
        foreach(T t in list)
            if(match(t)) {
                result = t;
                return true;
            }
        return false;
    }
}
```

Solution 11.9. C++ and Ada creates distinct code for distinct type parameters, and there is no universal representation. In contrast, Java uses type erasure to create a universal implementation for a generic.

Solution 11.10. Most languages uses on-demand instantiation, which means that there is no need to instantiate the generic by hand. However, in Ada, all generics must be instantiated and named explicitly. In C++, the instantiation is lazy, and only the used functions of a class is instantiated, which means that unused functions can contain type errors. In other languages, instantiation is eager, and the whole generic is compiled at once.

Solution 11.11. In case of *Foo*.**class**, we know the exact type and thus the parameters of the *Class* type. However, in case of *foo.getClass()*, the dynamic type is unknown, so we can only assume that the type of the variable *foo* is some subtype of *Foo*. When we load the class by name, we cannot assume anything, but it extends *Object*:

```
Foo foo = new Foo();
Class<?> cl1 = Class.forName("Foo");
Class<? extends Foo> cl2 = foo.getClass();
Class<Foo> cl3 = Foo.class;
```

Solution 11.12. *C++*

```
template<typename T>
T *instance()
{
  return new T;
}
```

Java

```
public class Factory<T> {
  public T instance(Class<? extends T> clazz)
      throws InstantiationException, IllegalAccessException {
    return clazz.newInstance();
  }
}
```

C#

```
public static class Factory<T> where T : new() {
  public static T instance() {
    return new T();
  }
}
```

12 Correctness in practice

As the introduction of this book reveals, the correctness of programs is an important quality factor. In this chapter we investigate the language constructs and elements available for increasing program correctness. This topic is addressed while discussing a particular example in Eiffel. We point out that to interpret (to create meaning of) the term correctness, we need somehow to define the expected behavior of the program, in addition to the syntax and semantics of a programming language in which it is written. Therefore, we first discuss the so-called correctness specification, which defines the expected program behavior. Only afterward we are able to address the notion of program correctness. Correctness specification for loops, methods, classes and programs will be defined. Furthermore, we investigate the relationship among program correctness, exception handling and inheritance. A brief overview is also presented on the issues and limitations of program validation. This chapter ends with a concise survey on contemporary languages supporting program correctness.

Present chapter describes how and to what extent the methods applied in the field of program correctness checking are applicable in the case of modern programming languages.

We deliberately use the term checking instead of verification or testing. As it is well known, there are two basic approaches to validating program correctness.

The most widely adopted approach is program testing. In the course of testing, program response is examined on predefined input sets. The program outcome is recorded and compared to the expected results with respect to actual input parameters. Thus actually checking in each case whether the program really works as it is supposed to, by giving the same outcome as expected. This process may be automated and facilitated by tools.

The other alternative, when programs are not tested by being executed, instead their correctness is verified by logical reasoning with mathematical precision. The latter approach, the logical reasoning over correctness properties of program, based on mathematical fundamentals, is called program-verification.

We strive to present practical knowledge in this chapter, nevertheless our discussion does bear resemblance to both previously mentioned alternatives to some extent.

It will be discussed how fundamentals of program-verification can be applied to programming languages. However, when applying verification theory in practice we actually diverge from theoretical fundamentals. The focus in this chapter is not on theoretical aspects of program-verification, nor we aim to present the methods of program-verification or their foundations, still we briefly discuss the relationship of theory and practice. Mathematically inclined readers who may want to consult books on the theoretical aspects of verification please refer to [GM94] and [LSS87].

We will see that extending programming languages with structures and elements specific to program verification enables us to check run-time properties of programs. This run-time monitoring is akin to traditional testing, but the significant difference is that in this case programs are validated by internal

mechanisms. Such correctness checking becomes possible thanks to the formal definition of program behavior, which is incorporated into the program itself. This formal definition or specification describing the program properties in turn is made possible by transforming program-verification fundamentals into language constructs.

12.1 Introduction

This chapter focuses on potentials of program-correctness checking elements in the Eiffel programming language. There have been several arguments for deciding on Eiffel. Eiffel is a modern object-oriented programming language, which supports numerous concepts of program-correctness validation. This is further made even more convenient with an integrated development environment. The real strength of Eiffel comes from the synergy of three main factors as follows:

- Many language constructs and tools for program-correctness checking based on the theoretical fundamentals of program-verification are available in Eiffel.
- New principles have been advocated by Bertrand Meyer, the inventor of Eiffel for components-based development, governing the relationships of components in terms of correctness specification called contracts. This approach is known as Design by Contract [Mey00], and proved to be very useful and efficient when designing and improving quality factor for systems with many components.
- The development environment (e.g. EiffelStudio) supports the methodology by enabling run-time checking of program-correctness and inspection of object properties.

Finally, the relative popularity (at least in research and education) of Eiffel also has validated our choice.

Besides Eiffel, quite a number of other alternatives have been considered. The Alphard language is one of them, in which language elements targeting program-correctness checking and verification have been present as early as the mid 1970s [Wul74]. The language concept and constructs supporting program-correctness may even seem to be more advanced than that of Eiffel in a sense that the duality and relationship of the abstract data type and its implementation is captured more clearly, also rules for iterators are unique. Unfortunately, no (efficient and fully fledged) compiler has been created and the language is barely known.

Another alternative worth considering is Sather [Gom97], which is a more recent modern programming language. Sather follows the syntactical convention of C/C++, while its semantics was influenced – including the constructs for program-correctness – by Eiffel. In our opinion due to the C-like syntax Sather programs may be more difficult to understand than their Eiffel counterparts for

those who are just being familiarized themselves with the notion of program-correctness. Furthermore, Sather does not seem to bring anything new to the table with respect to program-correctness, although its inheritance and type-system may be slightly more sound [How03].

Recently many contemporary programming languages have incorporated the notion of assertions and the approach of Design By Contract. In many cases, however, these constructs are not first-class citizens of the languages. For example, C# (and other .NET languages) have defined a framework called Code Contracts [Code13] built upon special classes emulating the behavior of Eiffel assertions (pre and postconditions, class invariants, etc.). Such assertions, so-called contracts are expressed by static method calls at method entries and exits.

Java community has also come up with similar solutions. For instance, Contract4J5 [C4Ja] is a tool supporting writing Design by Contract programs in Java. Assertions are defined with Java 5 annotations and expressed in the form of aspects in AspectJ.

There are certain limitations of such approaches. For example, Contract4J5 at the time of writing, provides only minimal support for preconditions and postconditions in the context of inheritance. This is mainly due to the fact that annotations on methods are not inherited [C4Ja].

JContractor [MHB99] is a library based framework, in which contracts are written as methods following certain naming conventions. Due to the nature of assertion implementation, JContractor is claimed to have preconditions, post-conditions, and class invariants, with full support for inheritance. Comparing to Eiffel this approach still lacks of some powerful constructs, such as loop variants and invariant to name a few.

C4J[C4Jb] is a Java framework supporting contracts for Java, it can be used as an Eclipse plugin too. Further examples are Java Modeling Language (JML), Jtest, SpringContracts for the Spring framework, iContract, etc.

A more comprehensive treatment on potentials of program validation offered by contemporary programming languages can be found at the end of this chapter.

The readers who would like to dedicate some more effort to this topic please refer to Turing ([HC83], [Hol84]) and Euclid [Lam77] programming languages, which have been the contemporaries of Eiffel.

12.1.1 Thought-provoking

Before we cover the topic of this chapter in detail, first let us stop for a moment and think about what we actually mean by correctness. In every-day's life we tend to use the expression: "the program works fine", "the program is correct", or "the program is incorrect" for that matter. Let us make an attempt to clarify the meaning of these notions. To do this, consider the following program fragment, which implements a simple function in Eiffel.

```
f : INTEGER is
do
   result := 1
end
```

It is quite futile to answer or even to raise the question whether the above program is correct or not.

First of all, for a program to be interpretable for us or for the compiler, the program must comply to a set of rules defining the formal textual and structural requirements. These requirements define the syntax of the programming language.

However, the language syntax itself is not sufficient to understand the operations of the program. We need to have clear understanding of the mechanisms and effects of language constructs. This is called semantics.

Even in possession of language syntax and semantics we still seem unable to answer the above question. This is the case, because the term 'correctness' is a relative concept. A program neither can be considered correct, nor incorrect per se. To interpret the notion of correctness we need to have the formal requirements for the program behavior and a precise description of the problem to be solved. We will refer to this description as correctness specification, or just simply as specification if it is clear from the context.

The above function clearly does not conform to the specification: "The function calculates the roots of an arbitrary quadratic equation." Nevertheless it does conform to the specification (and to many others indeed): "The function always returns a value less than ten."

12.2 Flavor of object-oriented approach

Prior to object-oriented paradigm the most important properties of structured programs could be described with some simple models. One model is based on a so-called *While* fictitious programming language [LSS87] (see Chapter 3, Section 3.3.1). Despite its simplicity it bears the important properties of third generation programming languages, and also served as a proper foundation for supporting formal verification of those program classes.

In case of object-oriented approach such simple model is not adequate. We here remark that there are theories as well describing the formal properties of object-oriented techniques, so-called object-calculi [AC98] and record calculi [GM94], but these are not covered in this chapter as it would be beyond the scope of this book.

12.2.1 Abstract data types

As opposed to functions and procedures being the quintessence of structured programming, the abstract data type is the central building block and the

focus of object-oriented model. The language mechanisms make it possible to encapsulate the properties and methods of the abstract data type and hide them from the outside world. Services of the abstract data types are exclusively available via public interface for its clients (for a more precise definition please refer to Chapter 9). While the implementation details of those services are concealed and not accessible from the outside world.

12.2.2 Type system

One of the most important fundamentals of object-oriented paradigm is the extendable type system. Not only a wider range of basic library types are readily available, but also the developer is able to define new types by combining already existing ones with various type constructs. Such high level abstraction mechanisms, being interesting on their own, are available, such as genericity. This characteristic supports creating parametric modules. Polymorphism and inheritance both further present interesting and effective possibilities.

Here we remark that as a prerequisite for the definition of program correctness the program must be type-correct. Intuitively this requires that at any time during program execution only expressions of the same or conforming types can be assigned one to another. Investigating the type-correctness is also a challenging issue on its own. This is the consequence of the aforementioned mechanisms and as a result of the dynamism, which is explored in the next section. Type-correctness issues will be addressed briefly in Section 12.5.3. From now on, it is assumed that the programs are correctly typed.

12.2.3 Dynamic properties

Dynamicity is at work increasingly for object-oriented paradigm. Implementation of object-oriented approach involves intensive dynamic storage allocation. By this we mean, that in the course of the program execution objects are created and ceased to exist. Objects are reached by reference-type variables. Since one object may be referenced by many variables, therefore the same object may be accessed and modified via multiple paths. This phenomenon is also called aliasing.

12.2.4 Object-oriented problem solving

Usually there are quite a number of classes present when addressing a problem in an object-oriented fashion. These classes are linked together to various extents. Solving the problem requires aligned operation and communication from these classes. Thus checking correctness of an object-oriented program implies checking and verifying all the classes used in the problem-solving.

Approach

Due to properties and nature of object-oriented paradigm discussed in the previous sections, verification and checking the correctness of object-oriented programs pose quite a challenge. Moreover checking program correctness requires verifying large set of classes that constitute the object-oriented program itself because of the nature of problem-solving.

Fortunately, there seems to be a simple approach, which may be taken to overcome this apparently hopeless situation. Namely, to decompose the problem (in our case the correctness checking of an object-oriented system) and split it into smaller and simpler problem domains. The basic concept is to handle the correctness of each class separately. Then, the verified components can be viewed and used as valid atomic blocks to compose more complex systems. This technique can be applied, partly due to the nature of data abstraction by making distinction between the implementation and interface of classes, thus separating them. In case, when only the implementation of a class changes, but its interface remains the same (including the semantics and correctness specification), then such change does not (directly) have impact on other classes. Hence modification and validation of the dependent classes are not (always) required. These principles are adopted by Eiffel programming language and its related methodologies having the following benefits:

- Validating the correctness of individual classes can be handled relatively easily.
- When a program is modified, it requires only re-checking and re-validation of certain parts (particular classes).
- Classes once checked and verified, can be reusable, thus reliable libraries can be built upon them. Eiffel library base classes can be considered such correct and verified modules.

Overview

In subsequent sections we first discuss the language elements making up the correctness specification. The language elements of correctness specification will be presented through an example, thus providing an opportunity for the reader to examine the advantages of such language constructs in practice.

In the possession of Eiffel correctness specification we will able to give meaning to the term of program correctness. However, it will be pointed out, that the correctness properties cannot be formulated in every case in a straightforward manner.

We briefly give an overview on the practical and theoretical issues of validating program-correctness. Finally a concise survey on contemporary languages supporting program correctness will be also presented.

12.3 The correctness specification language

The correctness specification for a short program discussed previously defined the problem's requirements in textual form, i.e. requirements were described in plain English. There are basically two issues when defining requirements in textual form using everyday language. If tools were used to support the checking of program correctness, then the textual description of requirements would be inappropriate, since it could not be interpreted by the tool in question. Another issue is that, in many cases textual definitions are not exact enough; they tend to be imprecise and ambiguous.

In the next section we present those language constructs and elements, which make up the correctness specification of programs written in Eiffel. These language elements comply with both requirements. They are meaningful for the compiler, as being part of the Eiffel programming language, furthermore with the aid of such constructs the program behavior can be defined precisely and unambiguously.

By correctness specification of Eiffel language, we mean a subset of the language. This proper subset comprises mostly logical expressions and predicate functions. These expressions describe in declarative way the behavior of the program. The correctness specification of Eiffel language aims to answer the question "What?", that is the program behavior is defined at a higher abstraction level. Naturally, the Eiffel program devoided of specification elements solves the problem in imperative way, in an attempt to answer the question "How?" similarly to other languages such as Pascal, C, C++ or Java.

We remark that the specification language is coherent with the program implementation, but at the same time is more abstract and it is a completely different notion. Specifications are not necessarily required to comprise Eiffel language constructs. There exist independent specification languages, not specific to any programming language. One of the most widely known is perhaps the Z [Spi92] and its object-oriented counterpart the Object-Z [Smi00].

There are practical reasons why Eiffel still adopts Eiffel language constructs for specification language. On the one hand, it makes the developers' task easier, since mastering a separate specification language is not required. On the other hand, it fulfills the condition so that the specification language matches the programming language. The latter condition is perfectly fulfilled, since the Eiffel correctness specification – as it will be pointed out – directly references Eiffel variables and functions.

12.3.1 Eiffel and first-order predicate Logic

Specification languages are mostly based on first-order predicate calculus. With the aid of first order predicate calculus statements describing program characteristics in declarative manner can be easily given. Therefore it is worth comparing the relationship between the first-order predicate calculus and Eiffel specification

language. Table 12.1 depicts the elements of first-order predicate logic with their respective Eiffel equivalents.

First-order logic	**Eiffel**	**Notes**
Unary and binary logical operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\equiv$	**not**, **and**, **or**, **implies** respectively	Equivalency can be defined as $(a \equiv b) \equiv (a \Rightarrow b) \wedge (b \Rightarrow a)$. non-strict equivalents of some of these operators are also available: **and then**, **or else**. **implies** itself is non-strict.
$=$	$=$, *equal*, *deep_equal*	*equal* and *deep_equal* are used to compare objects at attribute level.
Quantifiers ($\exists$, $\forall$)	*there_exists*, *for_all* used with agent construct, **all** and *some* keywords along with loop construct	Besides agents and iterators quantifiers can be modeled to some extent with functions and loops in theory. Please refer to the section on Quantifiers for more details.
Variables	Eiffel variables	
Logical constants	*True*, *False*	
Constants	Named constants and literals	
Functions	Eiffel functions	These are functions without global side effect, which do not modify object state.
Predicates	Eiffel functions with *BOOLEAN* return type	Same restrictions apply as above.
Terms	Literal, constant, variable or function	
any variable or constant is a term		
if $t_1, \ldots, t_n$ are terms, then so is f($t_1, \ldots, t_n$) where f is a function.	If $t_1, \ldots, t_n$ are literal, constant or variable and f is an n-ary function, then f($t_1, \ldots, t_n$) is also a term.	

First-order logic	Eiffel	Notes
Formulas $P(t_1, \ldots, t_n)$	Eiffel functions with *BOOLEAN* return type, where arguments are all terms	Where arguments are all terms.
$t1 = t2$ where t1 ... tn are terms and P is a predicate	If t1 and t2 are terms, then $t1 = t2$, *equal(t1,t2)*, *deep_equal(t1,t2)* are all formulas.	
$\neg F1$	If $F1$ is a formula, so is **not** $F1$	
$F1 \wedge F2$, $F1 \vee F2$, $F1 \Rightarrow F2$, where F1 and F2 are formulas	If $F1$ and $F2$ are formulas, then $F1 op F2$ is a formula as well.	*op* denotes binary logical operators such as **and**, **or**, **implies**, **or else**, **and then**.
$\exists xF$, $\forall xF$ where x is a variable and F is a formula	*there_exists*(agentP(x)), *for_all*(agent P(x)) where P is function with *BOOLEAN* return value and x is a variable. Also *across* operator with *some* and **all** keywords.	See the section Quantifiers for more details.

Table 12.1: Elements of first-order predicate logic and their Eiffel counterparts

Formulas obeying the syntactical rules outlined in the previous table define the set of so-called well-formed formulas in first-order predicate logic.

Logical operators

The observant reader may have noticed that in addition to the usual conjunction and disjunction logical operations the set of well-defined formulas contains other logical operations. In Eiffel, the **and then**, **or else** and **implies** all can be used. The **implies** operation is the logical implication as the preceding section reveals. The **and then** and **or else** and **implies** are the non-strict equivalents of conjunction and disjunction and implication respectively. Evaluation in case of these expressions is done sequentially from left to right. By this, we mean that firstly the left part of the expression is evaluated. If the evaluation of the second part does not have effect on the outcome of the expression, then it is not evaluated. For instance, in case of **and then**, if the first part evaluates to boolean value *False*, then the rest of the expression is not evaluated. Similarly, the second part of **or else** is evaluated only, if the first part equals *False*. Finally *a* **implies** *b* evaluates to **true**, if *a* has value **false**; otherwise it will have the value of *b*.

The *a* **and then** *b* logical operation can be expressed in Eiffel as follows:

```
and_then(a,b :BOOLEAN):BOOLEAN
do
  if a then
    result:=b
  else
    result:=False
  end
end
```

The *a* **or else** *b* is equivalent of the following Eiffel function:

```
or_else(a,b :BOOLEAN):BOLEAN
do
  if a then
    result:=True
  else
    result:=b
  end
end
```

Please note that if both parts are evaluated, then there is really no difference between the strict and non-strict counterparts (assuming that there is no side-effect involved). There is difference, however, if the second part cannot be evaluated (i.e. evaluation of the second part is undefined, for example, the function responsible for calculating second part does not terminate, or not defined). In case of strict boolean operations if any part is undefined, so is the outcome. Whereas in case of non-strict logical operations if first parts determines the outcome of the expression, then second part is not considered and evaluated, irrespectively of the fact that it is defined or undefined.

The **and then** logical operation can be widely used when the condition – regardless whether or not second part can be evaluated – is determined by the first part. For instance, if we would like to know if variable j is divisible without reminder by variable i, then as a first attempt we may tend to write

```
if j\\i=0 then ...
```

In the example above the \\ means the remainder of the division. The above code does work in most of the cases, but does not take account of the case, when $i = 0$. If i is indeed zero, then program execution is suspended due to exception caused by dividing with zero. To overcome this problem, we may consider writing

```
if i/=0 then
  if j\\i=0 then
...
```

The latter is correct and works as expected, but the structure is extended by a new **if**−**then**−**else** branch. While checking more complex scenarios the extensive

applications of **if**−**then**−**else** constructs makes the program more difficult to comprehend and maintain. To resolve issues like this, the non-strict versions of logical operations have been introduced. With non-strict operator the following code snippet works as expected, since the second expression is only evaluated if the first expression is true, so the division can be performed safely.

If i/=0 **and then** *j\\i=0* **then** ...

The Eiffel non-strict versions of logical operations can be applicable well in many areas of program-correctness validation. It is important, however, that the programmer is aware of their exact semantics.

Quantifiers

Universal and existential quantifiers are not directly available in Eiffel in form of language constructs. Still there is a number of approaches one can take to mimic these. The most general – also applicable in wide range of programming languages – solution would be to use loops or recursion (both without global side-effects), but there are more convenient ways in Eiffel at our disposal.

The keywords *some* and **all** can be used in loop constructs (both base and iteration form). For example, to test whether or not some property of all items of a list has a specific value can be expressed as follows with the help of the *across* operator:

across my_list **as** *l* **all** *l.item.some_property = some_value* **end**

One restriction is that the structure being traversed must belong to a descendant class of *ITERABLE*. Another one is that the structure is not changed while traversing it. Note it is also assumed that the items have the feature *some_property* conforming to the type of *some_value*.

Alternatively, the same effect can be achieved by the powerful combination of iterator (descendant of *ITERATOR*) classes and Eiffel agent mechanism. Assuming finite sets, the first-order predicates can be given as follows:

- $\forall x : P(x)$- *for_all (agent P(x))*
- $\exists x : P(x)$ - *there_exists(agent P(x))*

Where *for_all* and *there_exists* are routines of the iterator class, whereas $P(x)$ is an agent. In this example the agents act as functions with *BOOLEAN* return values, but agent may as well represent procedures. Generally speaking agents in Eiffel model operations. The fundamental difference between an agent and a routine is that although an agent represents a routine, the agent itself is an object, not a routine.

The fact that an agent expression may be either closed or open both in its argument or its target introduces even more flexibility. This is somewhat analogous to the concept of free and bound variables in the first-order predicate calculus, although it is slightly distinct.

First-order formula with quantifier	Agent with open and closed arguments
$\forall x : P(x, y)$	*for_all (agent P(?,z))*

In the above first-order formula the first argument is bound, while the second argument is free (not bounded by the universal quantifier). We say that an argument in the agent represented by the question mark is open. In such case its value is supplied by the iterator construct upon each iteration (so actually from the first-order predicate calculus perspective it is bounded by the *for_all* constructs in this particular case). In Eiffel terminology the argument z is said to be closed, since its value is determined in advance (supposing of course that z is not affected by the iterator).

We remark that besides the ones listed above, there is quite a number of iterator constructs: *do_all*, *do_if*, *do_while* just to name a few. These are mainly used for applying an operation to the structure elements meeting some criteria.

It is also worth observing that these Eiffel constructs (both loops and iterators with agents) have certain limitations. One of them is that they only work on collections with finite number of elements, such as lists, bags, sequences, trees, arrays, etc. One cannot practically transform the formulas similar to the one below into an equivalent Eiffel constructs:

$$\forall x, x \in \mathbf{Z} : P(x)$$

Assertions

Assertions are the most important elements of the correctness specification language in Eiffel. They are logical formulas (basically *BOOLEAN* expressions), which serve the basis for preconditions, postconditions of methods as well as for class and loop-invariant and check instructions to be discussed later.

An assertion may be labeled with a tag (to help identify the assertion) and may be followed by comments. Assertions are used to describe logical relationships among program variables.

These relationships may concern class attributes, local variables and return values of functions. Assertions, being the part of the Eiffel correctness specification language, strive to define the meaning of the program in a declarative manner, at the same time they help in understanding the program and the corresponding problem. As a result they provide good basis for documentation.

For instance, if in a particular context *vertex_cnt* variable is used to store the number of vertices of a polygon, and as a part of the correctness specification we would like to express that the polygon in question is either a triangle or quadrangle, then it could be done as follows:

vertex_cnt =3 **or** *vertex_cnt = 4*

or more preferably:

triangle_or_quadrangle: vertex_cnt =3 **or** *vertex_cnt = 4*

It is important to emphasize that if the program is correct with respect to its specification, then the variable *vertex_cnt* indeed should store values 3 or 4, as the assertion requires. If this condition does not hold, then it implies that the program does not conform to its specification, which in this case happens to be the single assertion above.

The two assertions are semantically equivalent. The subtle difference is the tag, which provides information with the associated boolean expression. The meaning of the first assertion may not be obvious to everyone (especially if the variable name is not descriptive), whereas the intention behind the tagged assertion is unambiguous. It is analogous to the situation when a value is used in the program text as a numerical literal as opposed to a named constant. In a program text, the value 8 may denote different things. It can be an age, an amount, etc. If, however, a named constant *spider_leg_cnt* is used with the same underlying value, it is certainly more comprehensible.

The reader may still argue that tags are superfluous, since the additional information can be given in form of comments after the assertion. This is partly true. It is possible to provide such information as comments. In fact in some cases (due to the lack of expressiveness of the specification language; more on it later) this is our last resort.

There are logical statements, which cannot be expressed (conveniently) in Eiffel. For example, we briefly discussed that quantors and quantified formulas – although with some limitations it can be circumvented by using iterators, across operator and agents – are not directly and generally supported by language mechanisms.

In other words, we cannot express formulas with ease such that ”every point is inside the polygon”. Even if Eiffel were a language with all the expression power of first-order predicate calculus, then there would be challenging situations. In such cases we have to make do with the empty assertion with comments as in the following example:

acyclic_graph: −− g graph is without cycles

The above assertion expression is empty, since the assertion is made up of the tag and comment only. Therefore this assertion does not introduce any additional value to the program correctness. In fact, such assertion with no boolean expression is considered to have value True. Thus the above assertion is equivalent to the one below.

acyclic_graph: True −− g graph is without cycles

Naturally such assertions are neither important from the viewpoint of correctness specification nor the executing environment. Still, they may serve basis for better understanding and may be useful for documentation.

We must mention that the assertions are not strictly part of the program execution flow, in a sense that they may/should not have any (direct) impact on program execution and hence its outcome.

In case of an assertion we do not evaluate the assertion expression in order to decide whether continue the program in a particular branch or another, to perform one calculation over another. An assertion can be viewed as a hypothesis on the program flow. Referring to the assertion labeled by *triangle_or_quadrangle*, if such assertion is placed at a given point of the program, then at that particular point we assume that the variable *vertex_cnt* does have value 3 or 4. This assumption is based on the knowledge of the applied algorithm.

If we would like to decide that a polygon is a triangle or a quadrangle, and if so, then continue with algorithm A, otherwise use algorithm B, then assertion is definitely not the right construct to achieve this. This simply can be done with an **if−then−else** statement in a similar way as it is done in many programming languages.

```
--triangle or quadrangle
if vertex_cnt =3 or vertex_cnt = 4 then
  algorithm A
else
  algorithm B
end
```

Nevertheless impact of assertions on program execution depends on compiling options. It is sufficient to say for now, that the assertion checking can be enabled (with some more options) and disabled. In the latter case assertions do not have any impact on program execution, since they are not even evaluated.

On the other hand, if assertions are enabled, and the assertion being checked is violated (that is, it evaluates to False) during program execution, then the program execution halts with a special type of exception.

Referring to the distinction between tags and comments used in assertions, in case such exception the executing environment will report the associated tag (if any). So in case of tags in present it is easier to identify which assertion is being violated (in other words, what hypothesis proved to be invalid). It must be pointed out that assertion evaluation generally may not have any impact on a correct program, apart from its speed of execution (provided that speed of execution is not a concern, which can be: e.g. real time systems).

12.3.2 Stack as an example

After a brief introduction to the Eiffel specification language, the elements of Eiffel supporting correctness specification will be presented through an example. We have chosen the stack, as a simple abstract data type, because this is a well-known notion and can be easily presented. Stack as an abstract data type will be

implemented in Eiffel as a class. This class will be built step by step. In each step we discuss the Eiffel programming elements supporting program-correctness.

Stack as abstract data type

Let us recap on what we mean by the concept of stack. Stack as a structure is such a container, which contains items conforming to a given type. Elements are put in a stack in LIFO fashion (LIFO stands for Last In First Out), which means the following: item to be put into the stack will be the top item. If another item is put into the stack, then that item will become the top item, whereas the item inserted before will be the second item from the top. At any given moment in time only the top (which is inserted most recently) item (if any) can be read or removed from a stack directly (first without having to remove other elements), hence the name. To manipulate the stack we need the following basic operations:

- *put* – to insert new item on the top of the stack.
- *remove* – to remove top item from the stack.

To simply read the top element, we will use the following query function:

- *item* – to read current top item from stack.

Obviously different sets of operations may be defined for manipulating stack data, but the above set also suffices. It is an easy exercise to see that manipulation of stack can be done with the above set of operations, and also more complicated operations can be defined in terms of this basic set. For instance, a *retrieve* operation, working in a destructive fashion by getting and at the same time removing the top item can be defined as a sequence of *item* and *remove* operations.

Out of curiosity we define a *reverse* method as well, which reverses the order of stack elements, so that top item will be the bottom item, second item from the top, will be the second item from the bottom and so on. For practical purposes (just to name one reason: the memory storage is limited) and also to make our class more compelling, we would like to restrict the number of elements to be inserted into the stack. Partly due to this restriction, we need the some more operations (better call queries) in addition to the ones introduced above:

- *empty* – to test whether the stack is empty. This returns **true** value if and only if stack contains no element.
- *full* – to check whether the stack is full. Returning **true** value if and only if the stack has already reached its capacity, thus no element can be inserted into the stack.

This completes the definition of our stack example.

Data representation

When implementing an abstract data type, the data type must be represented in terms of classes, components already being available in the language and the base libraries. Our stack as an abstract data type will be represented as a generic array along with two integer variables. The array will be used as a container for storing the elements. The variable *item_count* will define the current number of elements in the stack, whereas the variable *capacity* will define an upper bound for the number of elements that can be stored in the stack. The advantage of using *item_count* is the efficiency. Items will be only logically removed from stack, physically not deleted, only *item_count* will be decreased. This does not induce any problem, since items associated with greater index than the current *item_count* will not be accessed. When a new item is put into the stack, then the *item_count* variable is increased, and the old element above the top item (if any) is simply overwritten with the new one.

class *MY_STACK[G]*
feature *{NONE}*
  container : ARRAY[G]
  item_count : INTEGER
  capacity : INTEGER

Please note that the scope (in Eiffel terminology the export status) of attributes is NONE, which means that these attributes can only be directly accessed by methods of the same class. (Actually features with *NONE* export status are accessible only by those classes other than its defining class, which inherit from the class *NONE*. Since this special class is not ancestor of any classes by definition of Eiffel class hierarchy, it follows that features with such export status are only available for the defining class). It is obvious that many suitable representations of stack as abstract data type co-exist. For instance, we could use list-based representation instead of an array, or we could opt for inheritance construct (thus making our stack class a descendant of *ARRAY*) instead of using it as an attribute as in our above example.

12.3.3 Partial and total functions

It is worth noticing that some of the operations cannot be interpreted on each state of the *MY_STACK*. Calling predicates *empty* and *full* should provide consistent values for any existing stack instance, however, for example, *put* and *remove* methods cannot be interpreted, and thus executed on arbitrary stack instances. No item can be removed from an empty stack, similarly no item can be put into a stack, if the stack is already full.

A method can be viewed as a partial function. For instance, the *put* method can be seen as a function with arity two having a stack and a generic G item parameters and a return value of *MY_STACK* type.

put: MY_STACK x G => MY_STACK

The *put* operation implements a partial function, because it is not defined for all possible inputs. If the first argument is a full stack, then the function value is not defined, since no item can be put into that stack. Please notice that using partial functions is not satisfactory and practical in programs. We need total functions, which are defined, and thus provide us with result for every allowed combination of parameters. This key lies here in the "allowed" adjective. Let us have a look at the following mathematical function:

$$f(x) = \frac{1}{x} \qquad x \in \mathbb{R}$$

This function is partial too, since it is not defined for the case $x = 0$. We can, however, transform this partial function into a f' total function easily as follows:

$$f'(x) = \frac{1}{x} \qquad x \in \mathbb{R}, x \neq 0$$

We will take a similar approach when writing programs. We restrict the function domain to a set, so that the function will be defined for every element of that set, thus essentially becoming a total function on that domain.

12.3.4 Precondition

We will refer to this restriction applied to function domain as a precondition. The *precondition* is a characteristic function defined on the original domain of the (partial) function, which defines exactly those elements of the domain, for which the function is interpreted. (This characteristic function will assign *True* values for elements of restricted domain, and will assign False value otherwise.) In the above example we can consider the $x \neq 0$ predicate to be the precondition of the function f'.

It is quite straightforward to decide on the precondition of the *put* method. Since the *put* operation can be executed if and only if the stack is not full; thus giving us the precondition: **not** *full*. Analogously we can obtain the precondition of the *remove* operation: **not** *empty*.

12.3.5 Postcondition

We further would like to be able to describe somehow the outcome of the operations executed in a method or function. Such descriptions can be defined similarly to preconditions. For example, in case of the *put* method please observe that the item just being put into the stack will be the top item. Such conditions describing the effect of the method are called *postcondition*s. Let us suppose that *put* method receives the new item in the *new_item* parameter. Then part of the postcondition can be defined in Eiffel as: $item = new\_item$.

12.3.6 Pre- and postconditions in Eiffel

The precondition is introduced by the **require** keyword, while postcondition is put after the **ensure** keyword. We remark that neither the precondition nor the postcondition is actually mandatory in Eiffel. If they are not defined, then it will have the same effect as the *True* logical formulas were given.

```
put(i:G) is
require
  not_full: not full
do
  item_count:=item_count+1
  container.put(i,item_count)  -- store item
ensure
  new_item_on_top: item = i
  not_empty: not empty
  ...
end
```

We call the ordered pair <Pre, Post> the correctness specification (or specification for short) of a routine where Pre and Post denote the precondition and postcondition of the routine respectively.

Group of assertions

The attentive reader may have noticed that there are actually two assertions in the postconditions of the put method. This is, because one predicate may not be sufficient to describe the whole postcondition. Although one assertion may contain more logical expressions combined with the logical operators (**or**, **and**, **and then**, **or else**, etc.), such combination is not always desirable. It is straightforward to define the two logical expressions as one assertion.

Postcondition: item = i **and not** *empty* **and** ...

Such combination of different logical expressions is not always effective and convenient as only one label can be assigned to each assertion. In the previous postcondition of the put method the purpose of the two assertions is evident, whereas their roles are blurred in the second example. Therefore Eiffel enables us to put more assertions in precondition and postconditions (and also in class and loop invariant and check construct introduced later in this chapter).

By group of logical expressions hereby, we mean the sequence made of one or more assertions in the order they appear in the program source code. Assertions, although it is not indicated in the source code, are combined with the **and then** operator.

Old construct

When describing the effect of a method, one may frequently need to reference the value of an attribute prior to execution. This is, because only in the possess of old and new values one can compare and describe the changes. This is supported by the **old** construct, which can only be used in method postcondition. Only class attributes (also can be viewed as global variables) for the class can be used in this construct. The scope of **old** is restricted to the method and referring to the actual value of the variable prior to method invocation. Please recall that the *item_count* variable denotes the number of items currently stored in the stack. Thus the postcondition of *remove* operation can be defined as follows:

```
-- remove top item
remove is
require
  not_empty: not empty
do
  item_count:=item_count-1
ensure

  item_count_decreased: item_count = old item_count - 1
  empty_if_there_was_one_item: old item_count = 1 implies empty
  not_full:  not full
end
```

In the construct *item_count* = **old** *item_count* - 1 the **old** *item_count* denotes the value of *item_count* before the method execution. The whole construct expresses the fact that by removing one item from the stack, the *item_count* is decreased by one. We remark that **old** operator has the highest precedence, therefore the expression **old** *item_count* = 1 **implies** *empty* is equivalent with (old *item_count*) = 1 **implies** *empty*. Please notice that we also could describe two interesting consequences of the *remove* operation. The first one asserts if the stack had only one item before the *remove*, then it must be empty after. The seconds one says that once an item has been removed from a stack, the stack cannot be full anymore, although it does not say or require that it was full before.

Strip and only expressions

We have just seen how object attribute changes can be captured by **old** construct, but there are also cases when we would like to assert that some attributes are actually not affected, their values remain untouched. For example, we already mentioned the *reverse* method, which does modify the order of stack elements, but does not modify the number of items and their values stored in the stack

and the stack capacity. In the postcondition of *reverse* method this property can be expressed as follows:

same_item_count: **old** *item_count* = *item_count*
same_capacity: **old** *capacity* = *capacity*
...

In the above postcondition all the unchanged class attributes must be enumerated. The more attributes the *MY_STACK* class has, which remain unaffected by an operation, the more cumbersome it becomes to express the condition that these attributes are not updated by a method. Plus, if the class is expanded by a new attribute, then postconditions of all methods having no impact on the new attribute should be supplemented accordingly. In such cases the **strip** construct comes handy, which along with the **old** construct may be used in method postconditions. Attributes of the class are enumerated in parenthesis after the **strip** keyword. The **strip** construct defines an array (*ARRAY[ANY]*), which essentially is a subset of class attributes. This subset is the relative complement of the set – defined by class attribute elements enumerated in the **strip** expression – in the set of class attributes.

In other words this array includes exactly those class attributes, which are not appearing in between the parenthesis. Please notice that if no attribute is written after the **strip** expression, that is, we write **strip***()*, then the array will contain all attributes of that class.

Please recall that the items of our *MY_STACK* class are stored in an attribute called *container*. Thus **strip***(container)* in *MY_STACK* class defines an array, which contains all attributes. This is exactly what we need to describe the effect of *reverse* method. Postcondition can thus be given as follows:

*items_changed_only: equal(***strip***(container),* **old strip** *(container))*

The above postcondition asserts precisely that the *reverse* method does not have impact on any class variable except that the order of stack elements is changed.

ECMA-367 [ECM06] Eiffel has introduced the *only* expression, which is similar to, but slightly more general and straightforward than the strip expression discussed above. *Only* construct works on queries (the term query covers both attributes and functions) and may appear in postcondition as a last clause. Solely those queries are listed after the *only* keyword in a method postcondition, of which the return value calculated prior to and upon method execution differs as a result of the method being executed. It is assumed that all other queries are not affected by the method, that is, if q is a query not appearing in only clause in a particular method, then $q =$ **old** q. The syntax of *only* (similarly to **strip**) also permits omitting the list of queries, thus *only()* asserts that the method in which this clause appears in the postcondition leaves all queries unchanged. Such routine is known as ”pure” routine.

12.3.7 Design aspects

When specifying preconditions and postconditions we have to obey certain rules. One of them is that a logical expression can only be used in the precondition part of the routine, if all of its class attributes, functions (predicates) are also available for the caller of the routine. The reason behind this is that the caller cannot be expected to perform valid routine invocations (that is calling a routine with a class state and parameter combinations which satisfy the precondition), unless it is in a position to check actually the conditions beforehand. For instance, if the *remove* method of *MY_STACK* class the precondition makes use of the **not** *empty* predicate, then this predicate must be exported, that is, available for all classes, which can call the *remove* method. Certainly, we cannot expect to perform a consistent routine call satisfying the precondition of the *remove*, if there is no way for us to check this condition, by examining the value of predicate *empty* prior to the method call.

Note that there is no such restriction on assertions appearing in postconditions. This is, because to fulfill the routine obligations described in form of postcondition is the responsibility of the routine being invoked. If the routine terminates, the caller can expect the conditions given in postcondition to be fulfilled. It is a nice touch of the methodology that the caller, in fact, should trust the called, and under no condition should double check the result.

Local variables must not be referenced in the precondition and postcondition parts of the routine. As far as preconditions are concerned, this rule is partly the consequence of the rule that assertions must be available for the caller, on other hand, prior to executing the routine body local variables may have only default values, thus checking them in assertions would not have any practical benefit.

In case of postconditions, such restriction is due to theoretical and programming methodology reasons. Class methods introduce state transitions of objects. These transitions are characterized by changes made to class attributes, since object state is captured by them. What is important for us is to describe the effect of a method. This is solely related to how the object is transformed as a result of method execution, that is, from what initial state to what state the object is transitioned. Local variables are not suitable to capture such effect. Furthermore, local variables are assigned values from parameters and class variables, or from class functions called again with method parameters and class attributes. Therefore such values are available at the time of evaluating postcondition of methods (exceptions are functions with side-effects).

12.3.8 Class invariant

Valid objects

In the preceding section we have given one possible representation of the stack concept. Let us suppose that *container* array contains the following values: $[1, 2, 3, 4, 5]$

The *container* array alone may not be sufficient to capture the entire state of the stack, since depending on the implementation the above stack may represent more than one state or stack instance. The above array as one extreme may represent an empty stack, but as another extreme it can be viewed as a full stack. For this particular representation we need to consider *item_count* and *capacity* attributes as well to be able to assign a precise interpretation to the stack. The relationship between the actual representation (one concrete implementation of the *MY_STACK* class) and the abstract notion of stack is described by the so-called representation (also called abstraction) function. For more information on this function please refer to Section 5.1.2. The representation function maps concrete implementation to abstract implementation. In our case this function can be defined as follows:

representation:
container x item_count x capacity=>"STACK"−**as** *an abstract concept*

The actual implementation along with the representation function may still not be sufficient to describe a concrete instance of the stack. More precisely, not all possible representation instances may be valid. Please recall that *item_count* and *capacity* denote actual and maximum number of items in a stack respectively. If, for example, the value of *item_count* is 5 for a particular object instance and the *capacity* is 4, then the interpretation of this instance can problematic. Likewise a stack with negative *capacity* cannot be interpreted in a straightforward manner.

Please notice that in this case the problem is again that the representation function is a partial function. We take the same approach again by transforming the partial representation function into a total function. In addition to a concrete implementation we need to fulfill some conditions, which ensure that the representation function becomes total, which in turn, also implies that every instance of the class is valid. In other words, we do not allow any of the above examples. To maintain such consistent object state we will introduce predicates which help characterize properties of valid objects. We can make the following observations:

- *capacity* of the stack must be a positive integer
- actual *item_count* of the stack must be between zero and the *capacity* (zero *item_count* denotes an empty stack in our interpretation).

These properties of the particular abstract data type implementation can be expressed in Eiffel as follows:

invariant
valid_capacity: *capacity* *>0*
valid_item_count: *item_count* *>=0*
and *item_count* *<= capacity*
...

Such properties are introduced by the **invariant** keyword. These assertions are called class invariant. The name comes from the nature of the condition that its logical value cannot change and thus must be evaluated to *True* during the whole lifespan of an object, from object creation till object destruction, no matter what operations are performed on the object. That is, the values of these assertions are invariant or constant with respect to class operations. A class invariant can be composed of one or more assertions. These assertions are combined together with the **and then** logical operator in the order they appear in the software text.

We note that invariants interpreted for a given representation scheme (in our case *container*, *item_count*, *capacity*) unambiguously determine which instances of the representation are allowed and which are not. We also can say that class invariants characterize the instances of a particular representation scheme. For instance, the predicate labeled with *valid_item_count* asserts that for every valid object instance of this class the *item_count* must be between zero and *capacity*. This property is maintained for every possible operation at "certain moments of time".

Class invariant cannot only establish relationship among class attributes, but can also define relationship between class attributes and class functions. The invariant of *MY_STACK* can be extended with the definitions below:

definition_of_empty:(empty **implies** *item_count = 0)*
and *(item_count = 0* **implies** *empty)*
definition_of_full: (full **implies** *item_count = capacity)*
and *(item_count = capacity* **implies** *full)*

The declaration tagged with *definition_of_full*, for instance, maintains a relationship between the *full* function and the actual representation. It asserts correctly that the stack is full if and only if the *item_count* equals to the *capacity*.

We remark that an assertion appearing in class invariant may not necessarily have a direct counterpart in the abstract data type. In fact, the majority of assertions usually fell into this category. This is not surprising, since one of the purposes of introducing class invariants is to capture the properties of valid object instances making the representation function total. Such assertions directly connected to the actual realization of abstract data type are called implementation invariants in Eiffel terminology.

Evaluating class invariants

In the preceding section we used the term "certain moments of time", by which we mean the following: We cannot and do not expect actually the class invariant to hold for every possible moment of time. This is because in case of a more complex routine it may happen that these assertions are violated (that is some assertions from class invariant may be evaluated to *False*) temporarily. Later when discussing class consistency we define constructor method of the class. It is worth noting that class invariants will become *True* upon terminating

the constructor method and may not be *True* actually before and during the execution of the constructor method. This is not surprising, as we will see one of the responsibilities of constructors is to perform object initialization by considering class validity properties given in form of class invariant.

Class invariants are only evaluated before executing public method calls (exported generally or selectively), other than constructor methods, and after public methods. However, at these certain moments of time we do require class invariants to hold.

The reader may wonder why we do not require class invariants to hold before and after for private routine calls. The object is a concrete instance of the abstract data type. Their services may be accessible through its public interface. Inside of private (non-exported) methods we allow the class invariant to be temporary violated, since these methods are not directly invoked from the outside world anyway. Private methods can be viewed as extension of algorithms defined in public methods. These algorithms are factored out into separate methods because of software methodological reasons (reusability, maintenance etc.).

12.3.9 Check construct

We have seen how class invariant can be associated with the class, how pre-conditions and postconditions are defined for routines. Another possibility is in Eiffel, that assertions can be placed also inside routine and loop bodies. Such assertions are introduced by the **check** keyword. With the help of **check** construct one can assert and document non-trivial conditions. Assertions making up of check construct are similar to those used in assertions elsewhere in pre-, postconditions, etc. For example, if in a method body we are absolutely sure (because the algorithm is designed in such way) that the variable *discriminant* cannot have negative value, then we may consider writing:

```
discriminant:=b*b−4*a*c
check
discriminant_is_positive: discriminant > 0
−−because a>0 and c<0 or vice versa
end

x1:=(−b + sqrt(discriminant))/2*a
x2:=(−b − sqrt(discriminant))/2*a
```

The calculation of the roots can be simplified after the check construct, because analyzing the algorithm and the method parameters we came to the conclusion that the discriminant could not be negative, and this conclusion is defined in the form of the check construct.

We call the reader's attention to the fact again, that here using the **check** construct (just as this is the case for other assertions) does not indicate that such

condition is checked, and if so, then we calculate the roots as above, otherwise we perform some alternative calculation. The assertion part of the **check** construct is a hypothesis, which must be always true whenever the method body is executed. If such assertion is turned out to be false, then the method does not conform to its specification (specification here is used in a broader sense including check instructions). In the above example when calculating the roots of the quadratic equation, we do know (because we analyzed our class beforehand) that the signs of the coefficients of a and c are the opposite, therefore the hypothesis must hold. In such cases the discriminant obtained by the formula $b^2 - 4ac$ cannot be negative.

There are certain benefits, which come with using check constructs. To name a few:

- Assertions of check construct also serve as documentation for the program.
- Due to the use of check constructs the program logic can be simplified (for example, in the above example, an if-then-else branch can be eliminated).
- Invalidity of a hypothesis may be identified during the program execution.

12.3.10 Loops

General problem

Along with recursive calls, loops are, which pose one of the greatest challenges when writing programs. Typical issues are:

- Incorrect initialization.
- Executing the loop body one time less or more.
- Referencing elements outside the structure bounds (i.e. using greater or less index than permitted).
- Infinite-loop.

Reverse method

To make our stack class more appealing we define a method for reversing the order of items in the stack. The basic principle of the *reverse* method is the following. The method will swap two items at a time. First, it swaps the top element with the bottom element, then the second item from the top with the second one from the bottom and so on. This process goes on until all items have been swapped. This happens if the items being swapped have subsequent indices or have the same index.

Issues similar to those discussed previously may arise with this loop. It may happen that due to invalid iteration the loop does not terminate, or terminates abnormally. The method and hence the loop may terminate, but elements may not have changed as required. A classical symptom is when the majority of

items have changed appropriately, but typically either the middle elements or the top and bottom elements have not been swapped properly. Clearly, even such a simple algorithm may require focus on details and may introduce quite a number of issues.

Fortunately the Eiffel loop construct supports us in writing correct loops, and ensures that the above problems can be eliminated. The *reverse* method besides its postcondition and local variable declaration part only comprises a loop. By analyzing the example in this section we will become familiar with the Eiffel loop construct. The following variables are used in the loop:

- *lower_idx* and *upper_idx* denoting respectively the lower and upper indices of elements to be swapped.
- *lower_item* and *upper_item* representing the lower and upper elements respectively before the swap.

Iteration initialization block is given after the **from** keyword. This is responsible for setting up initial values of loop variables. In our case this is limited to setting values for lower and upper positions. The **invariant** keyword may be familiar from class definition. If it is not clear from the context then we will refer to it with its full name as loop invariant. The loop invariant is optional. This describes the properties of the loop, which are constant during the iterations. Essentially (and ideally) these formulas must define the whole purpose of the loop, the goal we are trying to achieve by the iteration.

In our loop items to be swapped identified by two indices. By analyzing the principles of the algorithm we can conclude that the *lower_idx* must be less or equal to the *upper_idx* at any moment of time (except when exiting the loop). We also know that the start value for lower index is 1 and the upper index initially has the value of *item_count*. Since the lower index is increasing by one and at the same time the upper index is decreasing by one upon each iteration, the sum of the two must be equal to the item count plus one (since the first item is indexed by one). The two indices are bounded by the interval from 1 to *item_count*, furthermore lower index is bounded by the upper index (plus one when exiting). This can be summarized in Eiffel syntax as follows:

```
invariant
  valid_range: lower_idx + upper_idx = item_count +1
  valid_index: lower_idx >=1
               and upper_idx <= item_count
               and lower_idx <= upper_idx+1
```

The above conditions make up the invariant, which asserts that elements to be swapped coming from the lower and upper part of the stack. It is also ensured that the role of indices defining the lower and upper indices of items to be swapped are symmetrical. It cannot happen that, for instance, the lower index is increased, but the upper is not decreased, since the sum of the two indices must be constant at any time. That is the lower index may be increased

by a value, which must match the value with the upper index is decreased. It is also guaranteed that indices reference stack items, which are bounded by 1 and *item_count*. Finally the loop invariant asserts that items referenced by *lower_idx* are coming from the lower range, whereas items indexed by *upper_idx* are coming from the upper range of the stack until the loop terminates.

We have seen that the formulas defined by the loop invariant describe important properties of the loop mechanism. With the support of this construct many of the aforementioned typical loop pitfalls can be avoided. Yet we still have not concluded anything in regard to loop termination, nor did we show that elements are actually swapped. Let us have a closer look at the termination problem of Eiffel loops.

The loop exit condition is given in Eiffel after the **until** keyword. Please notice that as opposed to practice in many programming languages, Eiffel loop body is executed until the exit condition becomes **false**. As far as our *reverse* method is concerned it is precisely the case, when the length of remaining interval is less than one. Which happens if lower and upper indices are the same, or lower index is greater than the upper, as expressed in the exit conditions as follows:

until *upper_idx − lower_idx < 1*

In Eiffel the **variant** expression can be used for arguing on loop termination. This keyword must be followed by an integer expression, which must have the following properties:

- It must have a positive integer value (including zero) before and after loop body is executed.
- Its value must be strictly monotonically decreased upon each loop iteration. That is, its value before executing the loop body must be greater each time, than after the body has been executed.

It is evident that the above properties are sufficient to conclude that the loop will terminate (provided that the loop body and exit conditions themselves terminate). No matter how big value the variant expression had before the loop, since this value is asserted to be positive and also being decreased strictly monotonically by the loop body with each loop cycle, this decrease cannot go on forever. Hence the loop must eventually terminate. The reader being familiar with the concept of program-verification, may have realized that the Eiffel loop variant expression is not other, than a special case of a terminating function with the well-founded set defined by partial order relation $(N, <)$.

In the *reverse* method the following expression is used as the loop variant:

variant
remaining_interval: upper_idx − lower_idx + 1

By looking at the loop algorithm we can easily verify that the above expression is a good choice. On the one hand, the initial value of loop variant is a

positive integer. On the other hand, we have already seen that the lower index is continuously increasing as the upper index is decreasing, with the net effect, that the value of the loop variant is decreased by 2 by every loop cycle.

We obviously could decide on a different loop variant, for example, the variable *upper_idx* would suffice alone. However, such variant would not be as expressive as ours, as our loop variant will hold the length of the remaining interval with items to be swapped at any given time during the loop.

With the help of the check construct we assert that elements are swapped.

```
lower_item   :=container.item (lower_idx)
upper_item   :=container.item (upper_idx)
container.put (lower_item,upper_idx)
container.put (upper_item,lower_idx)
check
  items_swapped:
    lower_item  = container.item(upper_idx)
  and upper_item = container.item(lower_idx)
end
```

This particular example may be superfluous, since swapping two items is a straightforward operation, the benefits of the check construct, however, can be justified for more complex scenarios. The full implementation of *reverse* method can be found at the end of this chapter.

Please notice the strength of the combination of Eiffel loop constructs at work. The termination of the loop guarantees that the order of items in the stack is indeed reversed. Exit condition indicates that loop will stop if the whole stack is scanned. This happens when the length of the interval to be scanned is less than one. The loop variant expression ensures that actually such situation (when the exit condition becomes true) will be reached sooner or later. Loop invariant formulas describe the symmetrical role of lower and upper indexes ensuring that elements with the desired indexes are selected. Finally, check construct inside the loop body guarantees that the appropriate stack items are actually swapped.

The whole Eiffel loop machinery may look startling and superfluous at the first glance. Do not forget, however, that only the exit condition is mandatory. Even if all other loop constructs are optional we still recommend that one should use them. The loop mechanism enriched with loop variant and invariant is more expressive and can be understood more easily. The loop variant and invariant expressions actually define the loop algorithm at a higher abstraction level.

To come up with the appropriate loop variant and invariant may take time and practice, but it helps understand and target the problem better. When the Eiffel program is compiled with appropriate assertion checking option, then as an additional benefit of the language and compiler is that the environment aborts the loop execution whenever the loop behavior is not in conformity with the loop specification described in form of loop variant and invariant. Thus typical loop issues can be identified and avoided.

12.3.11 Assertions and inheritance

In the context of inheritance many challenging issues may surface in connection with the concepts addressed so far. What has been discussed is still valid, but some of them need refinement.

Feature redefinition

Eiffel inheritance – besides many other things – supports the redefinition of features (attributes and routines). The redefinition itself can be achieved in numerous ways (using **redefine**, **rename**, **undefine**, or combination of these). For the sake of our discussion we limit ourselves to the effect, namely that during inheritance we are able to redefine features (with certain restrictions).

Let us take a closer look at inheritance from the perspective of (sub)typing. We remark Eiffel takes the approach that inheritance also induces subtyping relationship. That is, if a class C inherits from class B, and c is an instance of C, then c can be used in any context, which requires the instance of class B.

It is not trivial what should happen to method pre- and postconditions in such cases. If the precondition of a method of the new class (inherited from its base class) is more restrictive (we use the term 'stronger' informally), then object c could not be used in places, where instances of B are expected, since method of c would assume (in its preconditions) conditions which may or may not fulfilled, because the caller is expected to conform to preconditions defined in B class (which are less restrictive, or weaker). If we, however, use a weaker condition as the precondition of the method in C, then the substitution is allowed, because the precondition of method of B implies the precondition of the redefined method in C.

We can argue similarly for postconditions. Should the postcondition of the redefined method in C be weaker, then for client calling the method of C, the conditions defined in postconditions of the method in B could not be guaranteed. However, should the postcondition of the redefined method be the same or stronger, then the client can assume the postcondition (or even more) of the method in B to be satisfied.

In the above discussion we used the term "weaker" and "stronger" somewhat intuitively. We say that P_1 is stronger than P_2, if $P_1 \Rightarrow P_2$ assuming that P_1 and P_2 are different. In the same relation P_2 is said to be weaker than P_1.

We call the specification $\langle Pre', Post' \rangle$ the sub-specification of $\langle Pre, Post \rangle$, if and only if, $Pre \Rightarrow Pre'$ and $Post' \Rightarrow Post$. One of the conditions for subtyping in the context of inheritance and correctness specification is that the specifications of the methods in the class to be used in the place of its base class must be the subspecification of the respective methods in the base class.

Please observe that subspecification will be valid from the subtyping perspective. Because if a method in C has the specification $\langle Pre', Post' \rangle$, then its client assuming instance of class is obliged to ensure Pre, in turn satisfies Pre',

since $Pre \Rightarrow Pre'$ by definition. Similarly, if the client is expecting $Post$ to be satisfied upon method termination, then $Post'$ also suffices, since $Post' \Rightarrow Post$ by definition.

In this sense the Pre formula is stronger than Pre', because in every case when Pre is **true**, Pre' is also true, whereas we call $Post$ to be weaker than $Post'$ for the same reason. Fortunately, we can relatively effortlessly obtain a sub-specification for a specification. It can be easily verified, that for arbitrary $\langle Pre, Post\rangle$ specification, and A, B predicates, $\langle Pre \vee A, Post \wedge B\rangle$ is a sub-specification of $\langle Pre, Post\rangle$.

In line with the above, Eiffel makes the following restrictions on specification of redefined methods [Mey91]. Precondition is introduced by the **require else**, whereas postcondition is introduced by **ensure then** keywords instead of **require** and **ensure** respectively. Let $Pre_1, \ldots, Pre_n$ and $Post_1, \ldots, Post_n$ denote the pre- and postconditions of the methods in the ancestor classes (in most cases $n = 1$), furthermore let Pre and $Post$ denote the precondition and postcondition respectively in the method being redefined. Then the entire pre- and postcondition of the redefined method are essentially as follows ([Mey91], [Mey00]):

- if the **require else** clause is missing or it is empty, then the entire precondition is $Pre_1 \ldots$ **or else** $\ldots Pre_n$.
- If the **require else** clause is non-empty, then the entire precondition is Pre **or else** $Pre_1 \ldots$ **or else** $\ldots Pre_n$.
- if the **ensure then** clause is missing or it is empty, then the entire postcondition is $Post_1 \ldots$ **and then** $\ldots Post_n$.
- If the **ensure then** clause is non-empty, then the entire postcondition is $Post$ **and then** $Post_1 \ldots$ **and then** $\ldots Post_n$.

ECMA Eiffel[ECM06] introduces a slightly different and more rigorous approach both with respect to typing and assertions in the context of inheritance. It takes default assertion values *False* and *True* for precondition and postcondition respectively in case they were missing and it uses the following values for so-called combined precondition and postcondition:

- the full (or combined) precondition is ($Pre_1 \ldots$ **or** $\ldots Pre_n$) **or else** Pre.
- the full (or combined) postcondition is (**old** Pre_1 **implies** $Post_1$) $\ldots$ **and** $\ldots$ (**old** Pre_n **implies** $Post_n$) **and then** $Post$.

Where Pre_x and $Post_x$ are the (recursively) combined precondition and postcondition respectively of their enclosing routine in class. Moreover Pre_x and $Post_x$ denote the so-called covariance-aware forms of precondition and postcondition respectively (Please refer to Section 12.5.3).

Apart from addressing typing issues, the ECMA version of combined precondition is very similar to that of the original Eiffel language definition[Mey91], although the order of assertions and the use of logical operator (mostly or used in place of or else) are somewhat different.

It is worth analyzing the postcondition and comparing it to its original Eiffel language definition [Mey91] counterpart. This definition removes the burden of ensuring a stronger postcondition from the redeclared method at all costs. This burden is a result of the fact that the precondition for the redeclared method is actually weakened by essentially 'or'-ing its own and precursors' preconditions. Thus the postcondition its ancestors may not hold in cases when their precondition counterpart is not satisfied. Hence the ECMA version does not actually require the postcondition of its ancestors to be satisfied in such cases. Ancestor postconditions are required to hold only under their original conditions, when their precondition counterparts are satisfied.

In the subsequent sections we use the pre- and postcondition of a routine in the above sense, often referencing them as full preconditions and full postconditions respectively.

Class invariant

Let us turn our attention to the class invariant now. We already know that in Eiffel inheritance plays double role. On the one hand using inheritance we may reuse the implementation of the base class(es), on the other hand inheritance induces subtyping.

If class C inherits from class B, then C can be viewed a specialization of B. If we used inheritance properly, then conditions expressed in form of class invariants in B, should still be valid for class C. But as we specializing B into C, we may introduce additional properties specific to class C. We expect class C to include the full class invariant already defined in B, and maybe some more invariants related to additional features being present exclusively in C.

Let us define the new MY_STACK2 class inheriting from MY_STACK, having additional characteristics, that its capacity can be only even, the invariant of the new class could look like this:

```
invariant
  valid_capacity: capacity>0
  valid_item_count:     item_count >=0
          and   item_count >= capacity
  definition_of_emtpy:  ( empty implies item_count = 0
          and  ( item_count = 0 implies empty)
  definition_of_full:     ( full implies item_count = capacity
        and   ( item_count = capacity implies full)
  even_capacity: capacity \\2 = 0

end -- class MY_STACK2
```

Where \\ denotes the Eiffel modulo operator. Please note that we list all the class invariants of MY_STACK and extended it with the new assertion on even capacity.

To enlist the class invariants of base class(es) can be rather tedious and require considerable attention especially in the context of complex class-hierarchy. Fortunately, the class invariants of base classes can be obtained automatically. Eiffel language standard and the development environment do not require us to list the class invariants of base class(es). If a class C inherits from other class(es), then the class invariant of class C is automatically extended with the class invariant of the base class(es). We will refer to this extended class invariant as full invariant of the class. In case of our MY_STACK2 class it is sufficient to define only one new assertion, as illustrated below:

invariant
even_capacity: capacity \\2 = 0
end −− *class STACK2*

To generate the full class invariant is the responsibility of the Eiffel compiler and development environment.

We remark that the proper definition of the full class invariant in the context of inheritance and feature redeclaration (especially when redeclaring a function into an attribute) slightly differs from Meyer's original intention [Mey91] to current ECMA [ECM06] standard.

The definition [Mey91] substantially applies similar, but more precise (governing the order in which invariants of parent classes appear recursively) constitution of full class invariant (although the term full class invariant is not used) with the difference that it takes also care of a fine point: when a function is being redeclared into an attribute (in class C). In such case the definition [Mey91] states that the class invariant must also include "The postconditions of any inherited functions, which C redefines as an attribute, with every occurrence of *result* replaced by the attribute's final name (If there are more such redefinitions, include them in the order in which their new declaration appears in C)."

Note that preconditions of such redeclared functions are omitted, since an attribute value can be retrieved at any time. This is actually also in line with the precondition weakening principle, since essentially the precondition of such features becomes *True*. Object-Oriented Software Construction[Mey00] further argues that actually for argument-less functions it is a matter of style to express the assertion in class invariant instead of postcondition. If one opts for using the former then there will be no change to class invariants in this respect.

A function without arguments can only refer to and check in its precondition properties of global attributes and values of other argument-less features, therefore such precondition in theory can be formed as class invariant.

ECMA [ECM06] coins the term unfolded assertion in the context of assertions with inheritance. Class invariant of a class C is recursively obtained by unfolding all of its parents' (if any) class invariants. Thus the full class invariant, termed as local unfolded form by ECMA standard is given by the formula: "up_1 **and** ... **and** up_n **and then** ua, where up_1, ... up_n are (recursively) the unfolded forms of the invariants of these parents, after application of any feature renaming specified

by C's corresponding parent clauses." Where ua denotes the local unfolded form of class invariant in C and assuming that C has n parents for some $n \geq 1$. Please note that this definition is fundamentally very close to the ones already outlined above.

Class invariant, pre- and postcondition

Let us explore the relationship of the class invariant and pre- and postconditions. The interpretation of class invariant may pose interesting problems in the context of inheritance. We pointed out that the class invariant must hold before and after exported methods and after creation. We also know that the full class invariant is obtained by combining (with the **and then** operator) the class invariants of base class(es) with the class invariant defined in the descendant class.

Let B a class with the class invariant I. Let m denote a public method of B with the specification $\langle Pre, Post \rangle$. Since m is a public method of B, whenever m is called, the condition $I \ \wedge \ Pre$ must hold, and whenever m terminates the condition $I \ \wedge \ Post$ must be satisfied. At the first glance the class invariant may look superfluous, since by selecting appropriate pre- and postconditions it could be eliminated. However, this is not quite true because of two reasons. The first reason is methodological. Class invariant is meant to describe the overall characteristics of a class. The benefits of having a separate class invariant are:

- Class characteristics are more evident.
- Class is easier to maintain, because without class invariant, the assertions (making up the invariant) should be repeated in the pre and postconditions of all public methods.

The second reason follows from the argument: We will realize that pre and postconditions play a slightly different role than class invariants. Let us investigate this again from the perspective of subtyping and inheritance. Let still m denote a public method of class B with specification $\langle Pre, Post \rangle$ and I denote the class invariant defined in class B. Let us suppose that class C inherits from B. Let $I \ \wedge \ J$ denote the full class invariant of C, where J is the class invariant given explicitly in class C. Let C inherit method m from class B in its original form along with its specification. The effective conditions need to be satisfied prior to invoking m in B is $I \ \wedge \ Pre$, similarly the condition need to be fulfilled upon termination is $I \ \wedge \ Post$. For method m in class C the condition needs to be satisfied prior to execution is $I \ \wedge \ J \ \wedge \ Pre$, whereas the condition needs to be fulfilled upon termination is $I \ \wedge \ J \ \wedge \ Post$.

There seems to be an issue with the effective preconditions, since the assertion required to be satisfied prior to the execution of m in C may be stronger than for m in B. Note the same holds for the postconditions, but it does not pose a problem there, since the requirement induced by subspecification definition is satisfied, as $I \ \wedge \ J \ \wedge \ Post \Rightarrow I \ \wedge \ Post$ always holds.

Similar reasoning can be obtained if an inherited method m were redefined and its specification were modified according to the subspecification rules. In such a case the specification of the redefined m would be $\langle Pre', Post' \rangle$, which would satisfy on its own the subspecification relationship with respect to $\langle Pre, Post \rangle$.

Because of the reasons discussed above, other languages, for instance Sather [Gom97] takes a different approach. It allows only abstract classes as base classes in inheritance relation. This restriction although does not solve the issue explained above, but at least it does not impose problems in the context of subtyping, as instances of abstract classes cannot exist. A further refinement in Sather is that class invariants are only evaluated along with postconditions.

It is worth considering the above statement. In Eiffel language the program execution involves consecutive operations performed on various objects. If after termination of a public method the class invariant still holds (and it must), then the next public method may again count on the fact that class invariant is valid.

We seem to have reached a contradiction in our reasoning. Our starting point was that during inheritance – because of the nature of how the class invariant is inherited – the subspecification relation may not necessarily hold between the specification of the exported method in the ancestor and descendant classes, if we look at the precondition and the class invariant as a combined single assertion. It is true, however, that at least the subspecification relation holds for method postconditions extended with the class invariant. When taking a closer look at how computation is performed in the object-oriented paradigm we come to the conclusion that if the class invariant is satisfied upon termination of a public method, then it certainly also holds when starting the next public method.

This apparent contradiction can be resolved by the following practical reasoning. Please notice that a class (or for that matter an object) does not expect its clients to maintain its integrity in form of class invariant. On the contrary, the class (object) itself is responsible for establishing and preserving such integrity. Internal mechanisms provide that in case of valid method invocations the validity of class invariant is preserved, regardless whether the class invariant is stronger than that of its ancestor class, or whether the object is used in a context, which assumes an instance of its ancestor. Therefore we cannot simply treat class invariants as a sort of common extension of pre and postconditions. Class invariants capture deeper semantics as already have been pointed out when abstract data types and representation functions were discussed.

12.4 Program-correctness in Eiffel

In possession of language elements dedicated to correctness specification we are able to reason on the correctness of Eiffel programs. By correctness as it will be defined in the next section we will mean total-correctness.

From this on, by precondition, postcondition and class invariant we mean their full or unfolded form in ECMA [ECM06] terminology.

12.4.1 Hoare-formulas

Generally, in case of an S statement (single or compound) we use the notation of $\{Pre\}S\{Post\}$, where Pre and $Post$ predicates denote the precondition and postcondition of S respectively. This triplet is called Hoare-formula. The formula is valid if and only if, whenever S is executed in a state satisfying Pre, then S terminates in a state satisfying $Post$. We remark that this definition describes so-called total-correctness. There exists a weaker definition, which does not require S to actually terminate. In that case we talk about partial-correctness.

In this chapter we deal with the program-correctness and its validation from rather practical, not theoretical perspective. It makes a difference for us if a program does or does not terminate, therefore from now on we are only concerned with total-correctness in the above sense.

The $\langle Pre, Post\rangle$ ordered pair from the $\{Pre\}B\{Post\}$ Hoare-triplet is called the correctness specification of method M with method body B. Please observe that many correctness specification may co-exist for a given statement or method. For example, for the operation $x := x \times 2$ the $\langle x > 4, x > 7\rangle$ or $\langle x > 4, x > 8\rangle$ can be both appropriate correctness specifications. The correctness specification along with the corresponding statement defines the Hoare-triplet, for instance, $\{x > 4\}x := x \times 2\{x > 7\}$. The meaning of this formula is the following: if variable x is greater than 4 prior to executing the statement $x := x \times 2$, then the operation terminates and after the operation the value of x will be greater than 7.

A simple Hoare-formula can be defined for the *put* method of our stack:

$$\{\neg full\}\ \ put(new\_item)\ \ \{item = new\_item\ and\ \neg\ empty\}$$

To interpret this snippet is straightforward: if the stack is not full, then the *put* method terminates, and upon termination the new item will be available on the top of the stack. Also as a consequence of the operation the stack will not be empty. Please recall that the *item* function returns the top of the stack. Also notice that without the precondition and postcondition it is not guaranteed that the method will actually put the new item into the stack, in fact, on the top of the stack.

An alternative *put* method can be easily implemented, which does nothing with a full stack. It simply checks if the stack is full, it puts the new item into the stack, if it is not full, otherwise it does nothing. With this particular stack implementation the programmer relying on it may believe that a new item is put into the stack even if the stack is full as no feedback is received in such cases.

The above small example illustrates the significant role of checking preconditions. The other benefit is that if we inspect the postcondition of the *put* method, then we realize that it is guaranteed that for every allowed (in cases where preconditions asserts **true**) method invocation, the new item is actually put on the top of the stack.

12.4.2 Correctness of attributes

Correctness of attributes is not verified on their own. We assume all attributes correct because of the following reasons:

- Every attribute is a simple variable or object; fetching their values does not involve language level computation mechanism. It also follows that such value retrieval is supposed to always terminate.
- Consistency of attributes with respect to classes, other attributes, or functions is captured by class invariants defined at class level.
- Validity of attributes in a method invocation context is ensured by the precondition and postcondition of the method to be invoked.

Both latter topics will be addressed later. We note that it may happen that a feature (routine or an attribute) of a class attribute is referenced. This may happen in a particular f feature or in a class invariant. In both cases correctness of the referencing class does depend on the correctness of the class constituting the attribute. However, the correctness of the class for the attribute is not defined at attribute level in the class it is occurring, but at routine and class level of the constituting class.

12.4.3 Loop correctness

The definition of loop-correct is given in the context of routines [Mey91] as follows. A routine is loop-correct if every loop it contains satisfies the following four conditions:

- $\{REQ\}INIT\{INV\}$
- $\{REQ\}INIT\{\ VAR \geq 0\}$
- $\{INV \wedge \neg EXIT\}BODY\{INV\}$
- $\{INV \wedge \neg EXIT \wedge\ VAR = v\}BODY\{0 \leq\ VAR < v\}$

Where INV denotes the loop invariant, VAR is the loop variant, $INIT$ is the initialization part of the loop, $EXIT$ is the exit condition, finally $BODY$ is the loop body. It is further assumed that v integer variable in the above logical formulas does not appear anywhere in the routine. We note that REQ is simply taken to be $True$ ([Mey91] and [ECM06]) (more on this later).

The first formula states that the $INIT$ loop initialization block terminates in a state, which satisfies the loop invariant. As already discussed in the preceding sections this INV formula is served to capture the essence of the loop, to establish and preserve a logical relationship among variables (class or local) by expressing the properties of the goal we are trying to achieve along with the $EXIT$ condition. It is ensured by the third formula that the loop invariant remains true whenever the loop body is executed. Note that therefore the loop invariant is guaranteed to be true when exiting the loop (based on the $EXIT$

condition) after zero (it holds because of the first formula) or more iteration (it holds because of the third formula).

The rest of the formulas deal with the termination of the loop by establishing and maintaining loop variant. The second formula describes that the loop initialization must terminate in a state when the loop variant is non-negative. The last formula asserts that loop variant must be decreased for each iteration (executing the loop body until exit condition is reached) of the loop. The loop variant is required to be non-negative at all times also by these formulas. Hence the number of iterations is limited, bounded by the VAR expression, meaning that the loop will eventually terminate, provided that the entire loop (including initialization part, loop body and exit condition) conforms to its specification given in terms of the above formulas.

The above rules implicitly assert that both the loop initialization and the loop body themselves terminate, since the validity of $\{Pre\}B\{Post\}$ Hoare-formula in case of total-correctness requires that B terminates.

The correctness and correctness checking of the loop depends on how well loop mechanism can be captured by the language construct supporting correctness specification. As an extreme example the constant value *True* may be used in the loop invariant formula. It is obvious that in such a case the $True$ logical formula does not reveal any information of the loop mechanism.

It is worth taking a moment and looking at loop correctness definition more closely, especially to see why REQ is selected to be $True$ both by Meyer [Mey91] and the ECMA standard [ECM06]. First it looks surprising to take $True$ as a precondition in $\{True\}INIT\{INV\}$ and $\{True\}INIT\{VAR \geq 0\}$. In general, the $\{True\}INIT\{INV\}$ formula cannot be proven to be valid for arbitrary $INIT$ and INV (take for example $\{True\}i := -1\{i > 0\}$).

The loop-correctness definition, however, only is defined at routine level, and solely states that a routine is loop-correct, if its every loop satisfies the formulas appearing in the original definition. It does not use double implication, so theoretically this definition is correct, although somewhat weak in a sense that only a small subset of loops (which otherwise could be proven to be correct provided that appropriate REQ condition were selected) can be viewed as correct.

Furthermore, in some cases the precondition of the loop initialization part – thanks to routine precondition – can be viewed/made stronger. For example, if the loop is the first statement in a routine, then $\{True\}INIT\{INV \wedge VAR \geq 0\}$ can be assumed to be $\{PRE \wedge True\}INIT\{INV \wedge VAR \geq 0\}$ (to save space logical conjunction is used to combine formulas in postcondition), where PRE is the routine precondition. This can be done, since the routine precondition must always hold prior to executing the routine body, in this case the loop initialization. If we are talking about an exported routine, then loop initialization part may further rely on the class invariant. This makes sense, since the loop being the part of the routine is only supposed to get executed under certain conditions, for valid objects.

Informally, even if the loop statement is not the first statement in the routine, or there are more loops, or even there are embedded loops, the routine precondition and possibly class invariant still may implicitly assert much more than the weak $True$ precondition in the original definition, thus being sufficient to get the loop initialization and the loop itself is properly executed with respect to its loop correctness formulas.

To argue on loop-correctness in the context of classes and methods (not to mention embedded loops and recursion) can become quite complicated, that is why presumably the simple *True* formula was selected as a precondition in the Eiffel loop correctness definition ([Mey91] and [ECM06]).

12.4.4 Check correctness

Please recall that check constructs serving as hypothesizes on routine behavior, may appear practically anywhere in a body of a routine. Meyer [Mey91] and ECMA standard [ECM06] define this notion as follows: "An effective routine r is check-correct if, for every check instruction c in r, any execution of c (as part of an execution of r) satisfies all its assertions."

The above definition basically requires check assertions to hold only for valid routine invocations. We remark that abstract classes and features can be defined in Eiffel using the **deferred** keyword. Abstract classes must have one or more deferred features, each with preferably some specification but without particular implementation. At some point in the inheritance hierarchy a descendant may implement a deferred feature, by making it effective.

12.4.5 Exception correctness

The Eiffel programmer may write one special exception block - introduced by the **rescue** clause - at method level for handling exceptions. The exception block may have essentially two branches. The optimistic branch is ended with the **retry** keyword assuming that after certain preparations made by the block, the whole method body may be executed again (possibly by applying a different algorithm), hopefully, this time without raising an exception.

It is required that in order to rerun the method body, the precondition must be still (or again) valid (thus the block must do operations to restore precondition), and if it is a method exported selectively or generally (other than constructor), then the class invariant must be valid (again restoring class invariant if necessary) in addition. In case of private methods or constructor it is sufficient that the precondition is restored prior to re-execution.

If there is no hope for executing the method body again (by running the **retry** block and the method body zero or more times more), then operations for supporting non-handable exception are executed, finally method ends with raising an exception. Because in such case the method fails, there is no way to

guarantee the postcondition. Exported methods are still required to restore or establish class invariant.

The notion of exception correctness was introduced by Meyer.[1] His original definition is not concerned with export status of routines and does not make distinction between ordinary methods and constructors.

The ECMA standard [ECM06] takes even a simpler approach by requiring that "a routine is exception-correct if any branch of the *Rescue* clause not terminating with a *Retry* ensures the invariant."

A more complicated definition - taking account also of export status - based on Meyer's original [Mey91] can be adapted: The m exported method is said to be exception correct if and only if, for every branch of exception block B ending with **retry** keyword the $\{True\}B\{I \ \wedge \ Pre\}$ Hoare-formula, for every branch of exception block B without **retry** the $\{True\}B\{I\}$ formula is valid. The m private method or constructor is said to be exception correct if and only if for every branch of its exception block terminating with **retry** keyword the $\{True\}B\{Pre\}$ Hoare-formula is valid.

We remark that the weakest precondition *True* is applied for the exception block in the triplets $\{True\}B\{\dots\}$. This is, because after an unexpected exception we can rely neither on method precondition nor class invariant, since during exception program may be in an non-expected, unstable state.

We must stress that the exception block should be really concerned with unexpected failures that is addressing such exceptional cases, which cannot be foreseen in advance. Thus, for example, when performing a division and not paying attention to the special case when the denominator is zero is excluded from such unexpected scenarios.

As those (remaining) set of exceptions are rather tied to the execution and actual implementation related properties of programs, one may also argue whether or not exception-correctness should be included in the correctness definition. These exceptions may be related to and be the consequences of hardware failures (broken network link, etc.), physical limitations (memory or disk size, etc.), which are resulted primary from the execution model, not from the pure mathematical algorithm itself.

Finally we remark that although assertion violations are implemented in Eiffel base libraries and Eiffel implementations (e.g. EiffelStudio) as exceptions, these two notions are quite different. From a practical point of view assertion violation (except for check on void target) cannot occur when assertion monitoring is switched off or when correctness specification is completely missing or sufficiently weak. Whereas there is no such option for turning off exceptions (other than if the careless developer decides on ignoring exceptions in all exception blocks).

The fundamental difference between the two notions is, however, lies in their semantics. "A run-time assertion violation is the manifestation of a bug in the software." [Mey00] Routine preconditions and postconditions also serve

[1] [Mey91] and [Mey00]

for separating roles and responsibilities in the context of client and supplier relationship, known as contracts, in Design By Contract methodology [Mey00]. Precondition violation clearly indicates the manifestation of a bug in the client, similarly postcondition violation is a direct result of a bug in the supplier code.

Generally speaking assertion and assertion-violations are tied to the concept of correctness, as opposed to exceptions, which are related to the robustness of the software. "Robustness is the ability of software systems to react appropriately to abnormal conditions" [Mey00].

An exception may be a consequence of a non-expected internal or external condition. Here by 'internal' we mean something, which is directly dependent on and coherent with the algorithm and nothing else. As opposed to 'external', which is exclusively related to the "outer world" (suppliers, external systems, hardware elements, etc.). An exception resulting from a non-expected internal condition still indicates a bug in the code, however, external exceptions may not. Therefore informally a routine must be constructed in such a way so that no exception resulting from an unexpected internal condition should arise. Moreover all exceptions due to external conditions should be handled in accordance with exception-correctness policy. To understand and internalize the subtle difference please refer to the exercises section at the end of this chapter.

12.4.6 Class consistency

The term class consistency is introduced [Mey91] based on the notions of preconditions, postconditions and class invariant. According to the ECMA definition [ECM06]: "A class C is consistent if and only if it satisfies the following conditions:

- For every creation procedure p of C: $\{pre_p\}\ do_p\ \{INVC\ and\ then\ post_p\}$
- For every feature f of C exported generally or selectively: $\{INVC\ and\ then\ pre_f\}\ do_f\ \{INVC\ and\ then\ post_f\}$

where $INVC$ is the invariant of C and, for any feature f, pre_f is the unfolded form of the precondition of f, $post_f$ the unfolded form of its postcondition, and do_f its body." Further assuming that any missing assertion (precondition, postcondition, or class invariant) to be taken to $True$.

The above definition is one of the most important aspects of correctness definition. On the one hand, it captures well the semantics of class invariant. Class consistency requires that every creation procedure, provided that it is called with valid arguments (as expressed in the precondition), must establish the class invariant, that is, producing a valid object instance. Moreover, this invariant must be maintained by every valid public feature call. The first Hoare-triple can be considered as the base case for induction, the second triple is the inductive hypothesis, which ensures that if the hypothesis (expressed in form of class invariant) holds before the execution of the method, it will hold also afterward.

On the other hand, the formulas define the correctness of selectively or generally exported feature in terms of their correctness specification. Requiring both constructors and features that whenever invoked with valid arguments (expressed in precondition) on a valid object instance (in terms of class invariant and precondition. The former not interpreted for creation procedures.) they will terminate in a state satisfying their postcondition, and class invariant is maintained as discussed previously.

We remark that Meyer [Mey91] had given earlier a slightly different definition: "A class C is consistent if and only if it satisfies the following two conditions:

- for every creation procedure p of C: $\{pre_p\}\ do_p\ \{INVC\}$
- for every routine r of C exported generally or selectively: $\{pre_r \wedge INVC\}\ do_r\ \{post_r \wedge INVC\}$

" We tend to adopt the more recent ECMA definition, as in our opinion it is more compact. Not just because of the use of the non-strict boolean operators, but also because we feel that the use of postconditions in constructors is justified. Creating a valid object instance in the first place is clearly required, but it is usually not sufficient, since many valid object instances may co-exist within a representation schema. Postcondition of creation procedure should also determine the initial state of the object, as it is done in our constructor method of MY_STACK class.

```
-- creation: create a stack with capacity of c
make(c:INTEGER)
require
 valid_capacity: c > 0
do
  create container.make(1,c)
  item_count:=0
  capacity:=c
ensure
  space_allocated:  container /=void
  new_stack_is_empty: empty
  capacity_is_set:  capacity=c
end
```

Let us take a look at the precondition of the constructor. It requires that capacity must be positive. If such constraint on capacity were not defined, then it would not be possible to establish and later preserve class invariant. Postcondition gives the constraints of successful creation of a stack object. It asserts that space is allocated for the stack, the new stack is empty (note not an arbitrary, but a very specific object instance is created), and that it has the appropriate capacity as indicated in the argument of the constructor.

12.4.7 Class correctness

A class C is correct with respect to its correctness specification (pre- and post-condition of its methods, full class invariant, loop assertions, check constructs) if and only if it is consistent and its every routine is check-correct, loop-correct and exception correct ([Mey91] and [ECM06]).

12.4.8 Note on method correctness

It may be striking at first for the observant reader that Meyer ([Mey91] and [Mey00]) and ECMA [ECM06] standard correctness definitions do not say anything in regard to correctness of attributes and non-exported routines with respect to their correctness specification given along with method signatures. The definition includes check-correctness, loop-correctness and even exception-correctness for routines in general (regardless of their export status), but is not concerned with preconditions and postconditions of non-exported (or private for short) routines.

We have pointed out that to talk about attribute-correctness may be a little premature. As correctness of attributes are handled inside their constituting class. It is not very difficult to see that class correctness does not directly depend on private routines, since services of the class are only available via its interface. Essentially, through a chain of calls a private routine to be useful must be called from inside a public routine. If this is not the case, then the correctness of such routine is irrelevant class correctness wise, as this may be never executed. If, however, the non-exported routine is referenced by an exported one, then clearly the correctness of the referencing public routine depends on the referenced one. Hence the class consistency definition implicitly requires that private routines do terminate in a state, which is appropriate for ensuring the postcondition of the calling exported routines and maintaining the class invariant.

We remark that even in the absence of correctness definition of non-exported routines, their correctness – given in terms of their precondition and postcondition – may be continually being monitored depending on compiling options. Please notice that this has an interesting and possibly undesirable effect that run-time class correctness also depends on compiling option as illustrated by the following class snippet.

```
class C

feature{NONE}
private
require
  false
do ...
end
```

```
feature{ANY}
public
require
  pre
do ...
  private
  ...
ensure
  post
end
```

Let us assume that the class C is correct with respect to the definition given in Section 12.4.7, further assuming that *private* method terminates in a state "appropriate" for method *public*. If preconditions are not monitored, then class C works in conformity with its correctness specification. However, if precondition monitoring is switched on, then class C stops working due to precondition violation in the *private* method despite the fact that the class C is correct with respect to definition given earlier.

12.4.9 Program correctness

Before turning our attention to definition of program-correctness of object-oriented programs and the related issues, we must familiarize ourselves with some supporting notions.

Connections

In a given software context we refer to C as client and S as supplier, if class C refers to directly or indirectly to any component of S (attribute or routine). Please notice that a given class may be client in one context, and may be a supplier in another context. This relationship may be interpreted at the object instance level as well.

Dependency

We say that implementation of class A is dependent on class B, if there is a text in the implementation of class A, in which A and B are present as client and supplier respectively. We denote this dependency as follows: $B \rightarrow_d A$, to express that implementation of class A depends on the implementation of class B. This dependency relation is reflexive and transitive. The transitive closure of dependency relation is denoted by $\rightarrow_d^+$. For instance, $A \rightarrow_d^+$ denotes a set of classes, of which implementation is dependent on A, whereas $\rightarrow_d^+ A$ describes the set of classes, on which implementation of A depends. Corollary of this definition is that a descendant class always depends on its ancestor classes.

Program correctness

Let P be a program of which execution is started by invocation of constructor defined in class C. Then we say that program P is correct with respect to its specification if and only if class C is correct. We remark that for C to be correct it is further required that all other classes acting as servers (directly or recursively) in client-supplier relationship in the above sense are also correct. More precisely, program P is correct if and only if all classes defined by $\rightarrow_d^+ C$ are correct. For *MY_STACK* class this particularly means that it is required that

- the used ARRAY and INTEGER classes are correct.
- The MY_STACK class itself is correct.

12.5 Program correctness issues

In this section we briefly overview the issues and theoretical limitations of the program-correctness validation.

12.5.1 Dependencies

The previously introduced approach, which advocates that the problem of checking program-correctness can be easily decomposed into smaller problems that focus individually on correctness properties of individual classes and cannot be applied straightforwardly due to dependencies.

There could be easily multiple level of dependencies among classes and dependency can be reciprocal. Therefore correctness of a class C, cannot usually be isolated and easily verified. As we know the correctness of class C depends on the set of classes denoted by $\rightarrow_d^+ C$.

For example, in the simple scenario, when $A \rightarrow_d B$, $B \rightarrow_d C$ and $C \rightarrow_d B$, we are not in a position to argue on the correctness of a standalone class B or C. When addressing correctness, the classes B and C should be handled together as one unit. Furthermore, in order to be able to proceed, it is required that we already verified class A, since $\rightarrow_d^+ \ B = \{A, B, C\} = \rightarrow_d^+ C$.

Figure 12.1. presents a sequence diagram in UML notation of a common design pattern[Gam95]. The symmetrical dependency between the two classes, the Subject and Observer is illustrated clearly in the figure. Due to this co-dependency the correctness of these two classes can be addressed together.

- Instances of the *Observer* class, which are interested in the state of the subject, are registered at the instance of the *Subject* class by calling the *subject.register(observer1)* and *subject.register(observer2)* methods. Such registration usually comes with adding elements to a list in instance of subject class.

- Later, the observer object modifies the state of the subject, by calling *set_state* method of the subject.
- The instance of subject class notifies all registered observer instances upon state changes (*subject.notify*).
- The notify method will invoke the update method of the registered observers (*observer1.update*, *observer2.update*).
- Finally, every notified observer instance retrieves the current state of the subject by calling *subject.get_state*.

It is evident that many more complicated design patterns exist with multiple dependencies.

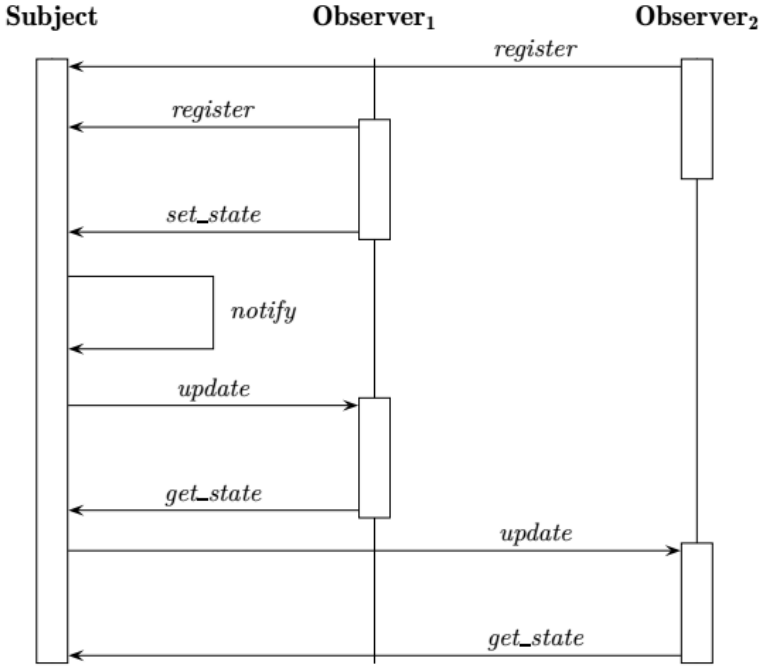


Figure 12.1: Subject-Observer sequence-diagram

12.5.2 Void-safety

The reader may have encountered the error "Java null pointer exception", or its Eiffel equivalent: "Feature call on void target". These are all clearly symptoms of non-void-safe software. It may be evident at the first glance, how and why the topic of void-safety should be tied to the notion of correctness. The potentials of void-safe constructs are worth discussing, as void-safety is a facility for improving software quality by eradicating calls on void targets [ECM06].

We informally say that a program is void-safe if it is guaranteed that no call is made on void object during run-time execution. Otherwise we categorize it as non-void-safe or void-unsafe. A language is void-safe, if it supports creating void-safe software, otherwise it is void-unsafe. With void-safety, of course, we do not, as we cannot, eliminate void references. There are certain cases, in which they are useful. It is enough to think of a linked list implementation, where the list is implemented by the generic Eiffel class $LINKED\_LIST[G]$ and its element, which are descendants of $LINKABLE[G]$ class. If a cell is the last element in a list, then it does not have a neighbor. This is expressed by having a void value in the attribute "right". Please note that void-safety only arises in connection with reference types, and does not pose problem for expanded types. The careful software engineer may write something like this to make the program void-safe:

```
if o/=void then
  ...
  o.m
  ...
else
    ...
end
```

Provided that no assignments to o are made in the interim, the feature call is safe. It is worth noting that to ensure such thing, it is required that object o above a local variable or a formal argument. Otherwise (e.g. o being an attribute) its value could be overwritten via a reference (see problem of aliasing).

Eiffel terminology calls such patterns like the above Certified Attachment Patterns (CAPs for short), when it is ensured that no call is invoked on void objects. It would be more convenient to handle such a case with more elegance and ease and also to extend this void-safety property to class attributes.

The attached syntax has been specifically introduced in Eiffel to solve this. In Eiffel a variable may be either declared (technically also depending on compiler options) as attached or detachable [ECM06].

It is ensured by the compiler that attached variables cannot be void, thus any feature call made to them is void-safe.

To enforce the validity of this property Eiffel has extended its type system with respect to conformance rules. A detachable variable cannot appear on the right hand side of an assignment, if the target is an attached variable. The reverse, however is allowed, since no harm may result from assigning a non-void value to a possibly void variable.

```
attached_var1 : attached SOME_TYPE
detachable_var1 : detachable SOME_TYPE
attached_var2 : attached SOME_TYPE
detachable_var2 : detachable SOME_TYPE
```

```
attached_var1 :=attached_var2  -- valid assignment
attached_var1 :=detachable_var1  -- invalid
detachable_var1 :=attached_var1  -- valid
detachable_var1 :=detachable_var2  -- valid
```

It is needless to say that the second assignment is invalid, whereas all the others are allowed. The same applies to routine arguments. An argument declared with some attached type only accepts attached parameters, while detachable arguments accept both.

Evidently this can only work, if variables declared as attached are immediately initialized to be non-void values. But when and how this should happen?

Eiffel has taken the same approach as in case of class invariants. The role of creation procedures is extended to also initialize attached attributes. In addition, an ordinary attribute may have a special initialization part with the following syntax, which guarantees that it is properly initialized prior to its first use.

```
my_string: STRING
        attribute
            create result.make_empty
        end
```

Note that CAPs are not required for such attached variables, but since there are still cases, when detachable variables with void references can occur, a care must be taken to handle such situations.

```
if attached o as l_o then
  l_o.feature_call
end
```

The above snippet checks if the *o* variable is attached. If it is, then it creates a local copy of it, and this local copy can be used further on, within its scope (in this case the if-then statement). Please note that by creating a fresh copy, we eliminated the potential issues that may be caused due to aliasing.

This so far looks promising but the life is not always as clear-cut as this. Sometimes a non-void value cannot be assigned to a variable right away, just after a later point in time. But it may be desirable that once a non-void value is assigned, the variable cannot become void again from that moment on. To address this issue, as a transition between attached and detachable variables there exists the notion of stable attribute.

```
my_attr: detachable SOME_TYPE
    note
    option: stable
    attribute
  end
```

Another delicate issue is void-safety in the context of genericity. Eiffel proposes constrained-genericity to assert that only attached types are allowed for a class parametrization. For example,

class *C [G –> attached ANY]*

denotes that only attached generic parameters are accepted. This works for all generic type parametrization, except for *ARRAY* [Eiff]. As long as only expanded types are used as generic parameters this construct is void-safe. When an *ARRAY* is instantiated with reference-types as generic parameters, then instead of simple *make* routine the *make_filled* must be used. The latter ensures that all array entry items are initialized properly. For instance,

a: ARRAY [STRING]
create *a.make_filled("",1,100)*

will create an array of empty strings, thus the above statement ensures that all items have non-void references. Finally, we remark that check construct has a slightly different syntax and semantics in the context of void-safety. A special variant of CAP is:

check *attached detachable_object* **as** *l_object* **then**
  l_object.feature_call
end

Here the check construct is used to assert that the *detachable_object* (possibly a stable attribute) being declared *detachable* (hence void-unsafe) is assumed to be non-void when this part is executed. To enforce void-safety, this type of check construct is always being monitored even if assertion monitoring is turned off.

12.5.3 Type safety

Type safety is the extent to which a programming language ensures that only statements and expressions (such as assignments, operators, methods, functions, etc.) of the right (conforming) type are valid in a language, furthermore such property is enforced during run-time execution [(refer within the book Chapter 5)]. Type safety can be achieved statically at compiler time or dynamically at run time or a combination of both. So far in our discussion we assumed that programs were both syntactically correct and correctly typed. Conventional non-OO imperative programming languages (such as Pascal, C, etc.) – lacking the expressive power of prominent features like polymorphism, inheritance, etc. – have relatively simple type system relying on some basic types (e.g. integer, real, string, etc.) with some possible limited form of extension such as records. Due to the rigidness of their type system it is easy to decide if a statement in a given program context is correct with respect to the typing rules. For example, the

following C program fragment illustrates that generally an assignment statement is not valid among variables of different types.

```
int i1;
int i2;
char c;
...
i1=i2; /* valid assignment */
i1=c; /* invalid assignment */
i1=(int)c; /* valid assignment due to casting */
```

However, to provide some plausible flexibility, mapping between certain types still can be made due to explicit type conversion known as casting. Due to the simplicity of such type systems type-correctness can be checked at compiler time, so that syntactically valid programs are ensured to be type-safe. Smalltalk being an early untyped OO programming language is another extreme with tremendous flexibility in typing. Due to this high level of freedom (essentially with no typing rules) Smalltalk programs may abort with the infamous "Message-Not-Understood" when an attempt is made to apply an operation to an object of the improper form (note: to put "type" here would be a paradox). Eiffel terminology refers to such run-time type violations as catcalls. Clearly, Eiffel programs must obey some typing rules but due to the complexity and flexibility of the type system the type validity cannot be statically checked at compiler-time. The sources of Eiffel catcalls have been identified[How03] to be the following:

- "A covariant argument redefinition.
- A routine argument whose type is a generic parameter.
- Descendant hiding (export restriction for an inherited feature)."

We remark that covariance formally can be defined as follows. Let A, A', B and B' denote types. We say that the x is a covariant operator (in both arguments), if $A'\ x\ B'\ <:\ A\ x\ B$ provided that $A'\ <:\ A$ and $B'\ <:\ B$, where $<:$ is the subsumption relation defined on types [AC98]. (Please also refer to Section 10.7.2 in Chapter 10.) Intuitively, $A'\ <:\ A$ indicates that A may be a more general type than A', and an instance of the possibly more specialized type A' can be used whenever instance of type A is expected. Eiffel takes the approach that the operator $\rightarrow$, hence function types are covariant ([AC98] and [SWM04]).

The type safety issue arises mainly when argument covariance is used along with polymorphism, genericity and descendant hiding. By prohibiting such features a language can remain type-safe. Notably such approach would limit the expressiveness of the language that is why Eiffel community still flavors their own approach. As we mentioned earlier Eiffel adopts the view that inheritance implies subtyping. This has numerous practical advantages, but is a clear source of potential catcalls. This is the case, because polymorphism (thus polymorphic method invocations) is made possible by this assumption that if a class C is

inherited from its base class B, then the instances of class C can also be viewed as instances of class B (however not the other way around). There are other views on whether the two notions inheritance and subtyping are related[AC98]. For instance, Sather with contravariant type system has separate implementation and type inheritance [Gom97], thus eliminating the majority of the above issues.

Correctness (sub)specification becomes even a more delicate issue in the context of covariance, and inheritance. To remedy this situation ECMA [ECM06] advocates the so-called covariance-aware form of an inherited a assertion, which is as follows:

- "If the enclosing routine has one or more arguments $x_1, \ldots x_n$ redefined covariantly to types $U_1, \ldots U_n$: the assertion
  $(\{y_1 : U_1\}\ x_1\ \mathit{and}\ \ldots\ \mathit{and}\ \{y_n : U_n\}\ x_n)\ \mathit{and\ then}\ a'$ where $y_1, \ldots\ y_n$ are fresh names and a' is the result of substituting y_i for each corresponding x_i in a.
- Otherwise: a."

Decoupling subtyping from inheritance will result in what is called implementation or nonconforming inheritance. This is proposed to be achieved in the future version of Eiffel by using the *expanded* keyword along with *inherit* clause (not to confuse this with expanded types, although the two are related to some extent.) For illustration see the table below.

Eiffel expanded type definition as per the current standard [ECM06]	Proposed Eiffel syntax [How03] for implementation inheritance
expanded class C...	class C inherit expanded B

When implementation inheritance is used the C descendant class solely reuses its B ancestor's implementation, but the two classes are not related in terms of typing. Neither $C <: B$, nor $B <: C$ holds. Therefore the operators x ,$\rightarrow$ follow so-called invariance policies. Neither instances of C can be used in a context where instances of B are expected, nor vice-versa. Therefore issues may otherwise raise under polymorphism are completely eliminated.

In places where the expanded inheritance is not used covariance still poses a problem for features exported generally or selectively. In such cases the developer must explicitly supply so-called recast functions, otherwise covariant redefinition will be rejected [How03]. If $C <: B$ and f feature is redefined in class C so that its argument type in class C is A' whereas its type is A in class B, where $A' <: A$, then the recast function must have type $A \rightarrow A'$ (in case of multiple arguments and redefinition $A_1, \ldots, A_n \rightarrow A'_1, \ldots, A'_n$). Thus providing an explicit and safe conversion for covariant arguments used in polymorphic routine calls.

As we have seen an attempt is being made to achieve a higher degree of type safety in Eiffel both statically (introducing expanded inheritance explained

earlier and enforcing some restrictions on descendant hiding) and dynamically (expecting the developer to define so-called recast functions in places of covariant redefinitions) [How03].

12.5.4 Concurrency

Concurrency in general introduces a great deal of efforts for developers. Most developers tend to think sequentially and thus designing, developing and testing even simple concurrent program may become soon a perplexing problem. Even simple predicate or precondition checking may fail (and now by this we mean to fail to serve its purpose) when used in concurrent context. Take, for example, when *remove* operation is executed in parallel on $s\ MY\_STACK$ object.

```
if not s.empty then      if not s.empty then
  s.remove                 s.remove
end                      end
```

If the two threads of code are interleaving then one of the *remove* operations may not be executed correctly. Please note that depending on timing, precondition of *remove* (which happens to be the same: **not** *empty*) may or may not be violated in the second method invocation, but certainly there could be some cases, when the second method call would fail. Eiffel has introduced the **separate** keyword to address concurrency issues and to denote that such objects will be handled by concurrent threads ([Eif13] and [Mey00]). Essentially, in such cases the correctness condition is transformed into a wait condition by the executing environment causing the *remove* operation being suspended until its precondition does not hold. Concurrency as a delicate topic on its own deserves a separate treatment. In the present section we intended only to call attention to the issues related to parallelism in the context of correctness validation. For more information on concurrency and parallel programming please refer to Chapter 13.

12.6 Correctness specification language

12.6.1 Practical limits

At the beginning of this chapter we compared the first-order logic with the Eiffel constructs supporting program-correctness checking. We pointed out that not every first-order formula can be directly captured with Eiffel program constructs. The observant reader may have noticed the *reverse* method does not require in its postcondition that the order of items should actually change. In Eiffel we cannot (always) describe easily properties such as ”all elements have changed” even with the advanced constructs like agents and iterators.

Although mimicking such quantified formulas can be relatively straightforward in some cases. For example, *LINKED_LIST* class has a predicate called has, which returns *True* if the list contains a specific element, and returns False otherwise. Thus whether a list l has element "six", can be formalized as $l.has("six")$.

Let us try to formalize in Eiffel the condition: "all items reversed". Since such predicate with universal quantor cannot be written directly, we try to achieve this by introducing a predicate function. So we extend the postcondition of the reverse method with the assertion

all_items_reversed: reversed(oc)

Where *oc* denotes a copy of the container before applying changes. So far this looks promising, but before reaching a premature conclusion let us come up with a possible implementation of this predicate.

```
reversed(o like container):BOOLEAN is
local
  item_pos:INTEGER
do
  result:=True
  from
    item_pos:=0
  invariant
    valid_item:  item_pos>=0
        and item_pos<= item_count

  until     item_pos = item_count
    or not   result
  loop
    item_pos:=item_pos+1
    result:=  result
      and container.entry(item_pos) =
            o.entry(item_count+1- item_pos)
  variant
    checked_items: item_count - item_pos
  end
end
```

The predicate *reversed* may work correctly, but it is not straightforward to see why. Universal quantifier in this example is circumvented by a loop construct. It is obvious that before using such function in other assertions one must prove that the function itself is correct and terminates. Generally speaking whenever we use functions with *BOOLEAN* return values to implement more complicated predicates, then correctness of such functions has to be verified prior to using them in assertions.

The strength of the correctness specification is that the program behavior is captured at a higher level of abstraction, focusing on the question "what" devoided of imperative elements. The *reversed* predicate does not fulfill this requirement. We remark that we have used functions with *BOOLEAN* return values as predicates before, such as empty and full. However, the correctness of these predicates can be verified easily.

full:BOOLEAN **is**
do
  result:= item_count = capacity
end

The full function seems to be quite safe, because it is virtually without any imperative elements. The *item_count = capacity* itself is a predicate, also conforming to first-order predicate calculus. The only imperative element is the assignment attached to result value, but this assignment serves purely technical purposes only and it does not alter the state of the object.

In closing, we can conclude that predicates with almost arbitrary complexity can be expressed in Eiffel, however, we must be very cautious with functions, which are complex, and their algorithm involves loops and recursive calls.

12.6.2 Model classes: an interim solution?

The precondition of the *put* routine seems to be satisfactory, however the same cannot be said about the postcondition. The postcondition asserts much less, than it should or could. While postcondition argues about the top item, it does not tell anything about the items already having been in the stack. These items could be overwritten, their order may be changed, etc. The routine does not guarantee anything useful in this regard in its postcondition.

It is observed that in general postconditions and class invariant are usually underspecified, hence ensure less than it would be desirable [SWM04]. This is, among other things, due to the absence of higher level mechanism and the lack of expressive power of Eiffel specification language.

Eiffel community wanted to address this issue in a way that Eiffel language would not be bloated with first order formulas exclusively related to specification language. Also their goal was to maintain the seamless integration between the language and specification. Furthermore, they did not want average classes to be polluted with predicates solely implemented to support predicates to appear in assertions. Finally, they wanted an efficient solution, which can be applied in practice ensuring that formulas are evaluated at reasonable cost during execution ([SWM04] and [Mey00]).

An approach was taken to rely on special classes, called model classes. These classes are deduced from mathematical concepts ([SWM04] and [Mey00]) representing elementary notions such as set, pair, relation, function, sequence, graph, powerset, etc.

Note that Eiffel thanks to its inheritance, genericity and other advanced mechanisms such as agents, is perfectly suited to implement such mathematical objects with ease.

It is evident that particular attention must be taken when developing such classes. Model class must be immutable, by minimizing side-effects and state-changes. To achieve this, a model class is allowed to have only queries (without side-effects), but not commands (which primary induce state change). This restriction is, however, for obvious reasons (to obtain the model instances in the first place) relaxed for creation routines ([SWM04] and [Mey00]). For instance, $MML\_SEQUENCE$ does not have a command extend, but instead has a query extended. Therefore the sequence $s : MML\_SEQUENCE$ itself cannot be extended (thus modified by writing $s.extend(i)$), instead $s.extended(i)$ denotes a new sequence made up by appending the sequence s with item i, whereas the original s sequence is untouched. (Analogously, when one writes the simple addition $2 + 1$, this operation does not change the value or identity of natural numbers 2 or 1, instead it denotes a new object, which represents the value 3.)

The correctness of these classes can be verified or can be taken granted by knowing that the concepts, properties they represent are proved mathematical axioms and theories. These classes can then be organized in model-libraries.

The average developer with the assistance of model libraries can argue over the properties of the class (henceforth referred as the developer class) he or she is developing. To do this, a suitable set (note for complex classes more than one model class needed) of mathematical concept (with existing implementation in model-libraries) must be selected. This choice is made so as to establish a clear relationship between the developer class and the model class. Or more precisely the choice must be such, that there exists a function (essentially a representation function) by mapping instances from developer class to that of the model class.

Instead of relying on state changes occurring in developer class, we define routine and class correctness in terms of conditions specified at the abstraction level of the model class.

For instance, the $MML\_SEQUENCE$ class seems to be a suitable model for the stack class ([SWM04] and [Mey00]). A query is with the following specification is added to the developer class:

{SPECIFICATION}
model: MML_SEQUENCE[G]

Note that *model* is exported selectively, and thus only available to classes concerned with model specification denoted by $SPECIFICATION$ in the above example. Model query is implemented so that it returns a stack of items modeled by a sequence. Each element in the sequence corresponds to an element in the MY_STACK representation in that particular order.

The postcondition of the put routine in MY_STACK thus becomes the following:

ensure
same_capacity: *capacity* = **old** *capacity*
item_count_inreased: *item_count* = **old** *item_count* + *1*
new_item_on_top: *item* = *i*
not_empty: **not** *empty*
stack_extended: *model* === **old** *model.extended (i)*

where === denotes a specialized version of equality operator defined to work well on model-classes. For more information please refer to [SWM04] and [Mey00].

The postcondition now captures the full semantics of *put* operation. It now also asserts correctly that old items are still in the stack in the exact order as they were beforehand. Note that *new_item_on_top* assertion now becomes superfluous, so that it could be dispensed.

Naturally the questions still remain:

- Model-classes can really be taken to be correct and trusted?
- What mathematical objects are the best candidates for model classes?
- How complex classes can be modeled efficiently and elegantly with such basic set of notions?

12.6.3 Theoretical limits

There is even a more important restriction on the usage of correctness formulas in Eiffel. In general the expressive power of axiomatic semantics is less than that of operational semantics [Mit98]. So even in the possess of an ideal specification language we will not be able to capture all important aspects of programs.

12.7 Languages and tools supporting Design by Contract

General importance and impact of the concept of Design by Contract to the software community is reflected by the spread of languages and tools supporting it. We would like to give a short overview of some of the interesting languages and tools. Most of them, but not all, follow the keywords of Eiffel, embedded in their own syntactic structures.[2]

12.7.1 D language

The D programming language[Dlng13] was designed by Walter Bright as a "next step" after C++. One of the main goals was the reliability. We can prescribe assert conditions, pre- and postconditions and class invariants.

[2] The description of these possibilities is based on the official references of the languages, as indicated.

Assert condition

The simplest condition is the *assert*: if it is false, an *AssertException* is thrown:

```
int i;
Symbol s = null;
for (i = 0; i < 100 ; i++)
{
  s = search(array[i]);
  if (s != null) break;
}
assert (s != null); //could we find the searched element?
```

Pre- and postconditions

The general form of a program in D with pre- and postconditions is:

```
in { . . . } out { . . . } body { . . . }
```

The rules of this construct are:

- *in* and *out* blocks are optional
- without *in* and *out* blocks, the keyword *body* can also be omitted
- *in* block checks the preconditions
- *out* block is executed after the body
- neither *in* nor *out* blocks may change the program environment

A simple example in D for the pre- and postconditions[Dlng13]:

```
long square_root(long x)
in
 { assert(x >= 0); }
out (result)
 { assert((result *result)<=x && (result+1) * (result+1) >=x); }
body
 {
   return cast(long)std.math.sqrt(cast(real)x);
 }
```

Type invariants

A special member function is the class invariant – it is not allowed to change the data members, and is automatically called after the constructor, before the destructor and before and after all public member functions to check the correct state of the actual object.

A simple example in D for the class invariant[Dlng13]:

```
class Date {
    int day;
    int hour;
 invariant()
    {
        assert(1 <= day && day <= 31);
        assert(0 <= hour && hour < 24);
    } ...
}
```

12.7.2 Cobra language

The Cobra language – designed by Chuck Esterbrook, who is well-known for his work in Python – is a general purpose, open source language, under a considerable development yet. The syntax of the language is very close to Python.

Pre- and postconditions

The concept of contracts in Cobra comes from Eiffel. We can prescribe, that certain pre-, and postconditions together with type invariants have to be true at the time of the execution of a method. We have to use the keyword *require* for the precondition and *ensure* for the postcondition of a method. These conditions can be described as lists of logical expressions. In postconditions we may refer with the keyword *old* to the original value of the attributes.

Simple examples in Cobra for the pre- and postconditions[Cob13a]:

```
class Person
    def drive(v as Vehicle)
        require
            not v.hasDriver
            v.isOperable
        ensure
            v.miles > old v.miles
        body
            ...

class ContiguousList<of T>
  implements IList<of T>
  def insert(index as int, item as T)
  require
            index >=0 and index < count
  ensure
       .count = old .count + 1
    this[index] is item
  body ...
```

Support of inheritance

The pre- and postconditions of the subprograms are inherited in Cobra according to the ”standard” rules:

- The inherited preconditions can be weakened by using the keywords *or require*;
- The inherited postconditions can be more restrictive by using the keywords *and ensure*.

Only the new requirements should be written in the redefinition of a method by the descendant (see [Cob13a] and [Cob13b]):

```
class NonContiguousList<of T> inherits ContiguousList<of T>
  """
  Allows insertions past the end of the list.
  """
  def insert(index as int, item as T)
   or require index >= 0
    body ...
```

Type invariants

The class invariant can be called using the keyword *invariant*[Cob13a]:

```
class Player
invariant
 .name.length.score >= 0
 .isAlive implies .health > 0
```

12.7.3 Oxygene language

The Oxygene language [Oxy13a], which was first called ”Chrome” language, was designed at RemObjects Software. It is based on Object Pascal and is available for .NET, Java, Android and Mono[Oxy13c]. The most important new feature in Oxygene is the support of contracts, it is based syntactically and semantically on Eiffel.

Pre- and postconditions

Pre- and postconditions can be written in Oxygene only for the methods of classes.

Preconditions should be written after the keyword *require*. It must be true when entering the method – here can one give, e.g. the constraints of the parameters.

The postconditions, given after the keyword *ensure* are evaluated before the end of the method. In the postconditions we may refer to the original value of the attributes with the keyword *old*.

A small example [Oxy13b]:

```
method MyObject.DivideBy(aValue: Integer);
require
  aValue <> 0;
begin
  MyValue := MyValue/aValue;
end;

method MyObject.Add(aItem: ListItem);
require
  assigned(aItem);
begin
  InternalList.Add(aItem);
ensure
  Count = old Count +1;
End;
```

Type invariants

The Oxygene language supports two kinds of type invariants:

- the conditions written after the keywords **public** *invariants* are checked at the end of *all public* methods, after the postcondition,
- the conditions written after the keywords **private** *invariants* are checked at the end of *all* methods.

A small example [Oxy13b]:

```
type
    MyClass = class;
  public
    ... some methods or properties
    public invariants
      fField1 > 35;
      SomeProperty = 0;
      SomeBoolMethod() and not (fField2 = 5);
    private invariants
      fField > 0;
    end;
```

12.7.4 Correctness in .NET

Microsoft Research supports reliability of programs by the *Code Contracts Library* in the .NET Framework[Code13]. Thus all .NET programming languages – C#, VB, F#, Scala, etc. – can lean on its benefits.

Assert and assume conditions

We can check an arbitrary condition in the program with the *Contract.Assert* and *Contract.Assume* functions. They have the same runtime behavior, the only difference between them is, that *Contract.Assume* can be used by static checker. Both functions have two forms, it is possible to give a string message as a second parameter, which will be written to standard output if the condition is false.

Small examples [Code13]:

```
Contract.Assert( this.privateField > 0 );
Contract.Assert( this.x == 3, "Why isn't the value of x 3?" );

Contract.Assume( this.privateField > 0 );
Contract.Assume( this.x == 3, "Static checker assumed this");
```

Pre- and postconditions

Preconditions can be checked by the function *Contract.Requires(. . .)*. In general it can be used for controlling the input parameters. In the contract, the fulfillment of the precondition is the responsibility of the caller method. The reason of the rule is, that all components of the precondition should be available for the caller. Conditions may not have side effects. If, e.g. the parameter x cannot be *null* [Code13]:

```
Contract.Requires ( x ! = null );
```

We can prescribe also the exception raised in case the condition check fails:

```
Contract.Requires<ArgumentNullException> ( x != null );
```

Traditionally the check of the appropriate parameters in the methods is described in **if**–*then*–*throw* structures. If these are at the beginning of the method, they can be transformed to precondition, but an explicit contract-method call is needed after them. (e.g.: *Requires*, *Ensures*, *EnsuresOnThrow* or *EndContractBlock*) As for example:

```
if ( x == null ) throw new . . .
Contract.EndContractBlock();
   // all preceding if statements are viewed as precondition
```

Postconditions can be checked by the function *Contract.Ensures(. . .)*. The argument of the function should be the condition as, e.g.:

```
Contract.Ensures( this.F > 0 );
```

A special possibility is that we can prescribe conditions also when exception is thrown – depending from the T type of the exception using the function *Contract.EnsuresOnThrow<T>(. . .)* as, e.g.

```
Contract.EnsuresOnThrow<T>( this.F > 0 );
```

There are some special methods which we can use only in postconditions:

- *Contract.Result<T>()* – refers the returned value of the function – it is not allowed by **void** functions, as, e.g.:

```
Contract.Ensures(0 < Contract.Result<int>());
```

- *Contract.OldValue<T>(e)* – refers to the value of the expression e at the beginning of the function. It's use has limitations:
  - the expression e may not contain another *oldValue* function
  - the expression e may have only components which had value before the call of the method

Examples of possible errors:

- It is not allowed to refer *Result* in the *OldValue* function as, e.g.:

```
Contract.OldValue(Contract.Result<float>() + sg) // ERROR
```

- *OldValue* may not depend from the *Result*, as, e.g.:

```
Contract.ForAll(0,Contract.Result<int>(),
    i => Contract.OldValue(y[i]) > 5 ); // ERROR
```

We can check the *out* parameter of a method in the postcondition with the function *Contract.ValueAtReturn<T>(out T t)* as, e.g.:

```
public void OutParam(out int x){
  Contract.Ensures(Contract.ValueAtReturn(out x) == 3);
     x = 3;
  }
```

Type invariants

Type invariants should be true before and after the call of public methods. To check this we can create a special invariant method in our class, where the body of this method contains call(s) of the function *Contract.Invariant(. . .)*;

The invariant method of a class must be of type **void**, and if it is allowed to inherit from this class, then its visibility should be *protected*.

The [ContractInvariantMethod] attribute should be used to mark a method being an invariant method.

The .NET framework checks the invariant of the class for all public methods after the execution.

A small example:

```
[ContractInvariantMethod]
  protected void MyObjectInvariant()
  {
    Contract.Invariant( this.y >= 0.0 );
    Contract.Invariant( this.x < this.y );
    . . .
  }
```

Other possibilities

- *Contract.EndContractBlock* – If the preconditions are described in **if**-*then*-**throw** form at the beginning of the method, then this indicates that these are preconditions, here is the end of the control block, as, e.g.:

```
  if ( x == null ) throw new ArgumentNullException("x");
  if ( z < 0 ) throw new ArgumentOutOfRangeException(. . .);
Contract.EndContractBlock( );
```

- *Contract.ForAll* – it is a check for elements in a collection, it can be used in a contract. It has two forms:
  - two-parameter version: the first parameter is a collection; the second parameter is a predicate. The function checks the predicate for all elements in the collection and returns true if the predicate is true for all, stops and returns false, if there is an element for which the predicate is false. A small example:

```
public int SomeEx<T>(IEnumerable<T> sg){
  Contract.Requires(
     Contract.ForAll( sg, (T x) => x != null)
     );
```

- – three-parameter version: here the first parameter is the lower limit, the second is the upper limit of the collection to be traversed given in the third predicate-parameter. The function checks whether the predicate is true for all elements between the lower and upper limit. A small example:

```
public int[] Bar(){
  Contract.Ensures(
     Contract.ForAll(0, Contract.Result<int[]>().Length,
           index => Contract.Result<int[]>()[index] > 0)
     );
```

- *Contract.Exists* – it is also a check for elements in a collection, it can be used in a contract. It also has two forms:
  - two-parameter version: the first parameter is a collection; the second parameter is a predicate. The function checks the predicate for the elements in the collection, stops and returns true if there is an element in the collection for which the predicate is true, returns false if the predicate is false for all elements in the collection.
  - three-parameter version: here the first parameter is the lower limit, the second is the upper limit of the collection to be traversed given in the third predicate-parameter. The function checks whether there exists an element in the collection between the lower and upper limit for which the predicate is true.

Contracts of interfaces and abstract classes

In case of interfaces and abstract methods of abstract classes the methods may not have a body, thus contracts can be added with the help of a special contract class, which is connected to the interface with attributes, see bellow.

- Interface contracts:

```
[ContractClass(typeof(ISomeExContract))]
  interface ISomeEx {
  int Count { get; }
  void Put(int value );
}
[ContractClassFor(typeof(ISomeEx))]
abstract class ISomeExContract : ISomeEx {
  int ISomeEx.Count {
        get {
    Contract.Ensures( 0 <= Contract.Result<int>() );
       return default( int ); // dummy return
    }
  }
  void ISomeEx.Put(int value){
    Contract.Requires( 0 <= value );
             }
}
```

- Abstract method contracts:

```
[ContractClass(typeof(SomeExContract))]
abstract class SomeEx {
  public abstract int Count { get; }
  public abstract void Put(int value );
}

[ContractClassFor(typeof(SomeEx))]
abstract class SomeExContract : SomeEx {
   public override int Count {
  get {
  Contract.Ensures( 0 <= Contract.Result<int>() );
     return default( int ); // dummy return
  }
   }
  public override void Put(int value){
  Contract.Requires( 0 <= value );
  }
}
```

Overloading of contract methods

Each contract method may have a second argument of type string. If the condition is false, the error message will be, e.g.:

```
Contract.Requires( x != null,
      " If x is null, then an error occurred! " );
```

Inherited contracts

Contracts are inherited to the subtypes of a type. The precondition of a method in the subtype must be weaker, thus if the method of the ancestor doesn't have any precondition (it means, the precondition equals with the value *true*), the descendant's method cannot have precondition. The descendant may redefine postconditions of inherited methods - they must be stronger than the original in the ancestor.

12.7.5 Java language and additional tools

Assert statement in Java

The only built-in possibility in Java for the support of Design by Contract is the *assert* statement. With the help of it one can describe the pre- and postconditions

and invariants of classes[Jass02]. Its benefit is – compared to **if** statements – that it can be switched on and off.

There are two forms of the statement:

```
assert logical_expression;
assert logical_expression: message;
```

If the logical expression is false, the statement throws an *AssertionError* and returns the message if it is given. An important rule is, that the condition in the *assert* statement may not have any influence on the execution of the program since the use of it can be switched off.

A simple example from the book Deitel and Deitel[DD05] illustrates the use of the *assert* statement:

```
import java.util.Scanner;
 public class AssertTest {
   public static void main( String args[] ) {
  Scanner input = new Scanner( System.in );
  System.out.print( "Enter a number between 0 and 10: " );
  int number = input.nextInt();
   // assert that the value is >= 0
      assert ( number>=0 && number<=10 ): "bad number: "+number;
   System.out.printf( "You entered \%d\n", number );
    } // end main
} // end class AssertTest

Enter a number between 0 and 10: 50

Exception in thread "main" java.lang.AssertionError: bad number: 50
at AssertTest.main(AssertTest.java:15)
```

Java - jContractor library

The jContractor library ([MHB99] and [Abe03]) is a pure Java support for Design by Contract. The contracts can be added to the classes as separate functions using a naming convention, the conditions are in Java.

Pre- and postconditions

The precondition can be given with a **boolean** function, where after the original function name the suffix *_Precondition* is written as, e.g.[Abe03]:

```
protected boolean push_Precondition (Object o) {
  return o != null;
}
```

The precondition is checked before entering the method body.
The rules for preconditions are:

- Contract methods, native methods and the *main* method may not have preconditions.
- The preconditions of **static** methods must also be **static**.
- The preconditions of non **static** methods are not **static**.
- The preconditions of non **private** methods must be **protected**.
- The preconditions of **private** methods must also be **private**.

The postconditions – similar to preconditions – can be given with a **boolean** function, where after the original function name the suffix *_Postcondition* is written. It is allowed to use the argument *RESULT* too, the type of it is the return type of the original function as, e.g.[Abe03]:

```
protected boolean push_Postcondition (Object o, Void RESULT) {
  return implementation.contains(o) && (size()==OLD.size()+1);
}
```

In postconditions the *OLD* refers to the value of the object at method entrance time. The use of *OLD* is allowed only in case the class implemented the interface *Cloneable*, i.e. the object must have a *clone()* method.
The postcondition is checked before leaving the method body.
The rules for postconditions are:

- Contract methods and native methods may not have postconditions.
- The postconditions of **static** methods must also be **static**.
- The postconditions of non **static** methods are not **static**.
- The postconditions of non **private** methods must be **protected**.
- The postconditions of **private** methods must also be **private**.
- The postconditions of constructors do not rely on *OLD*.

Type invariants

Type invariants belong to the class. This can be described with a **protected** boolean function – without arguments – called *_Invariant* as, e.g. [Abe03]:

```
protected boolean _Invariant () {
      return size() >= 0;
}
```

Invariant is checked before entering the public methods and before leaving the public methods and the constructors.
The rules for type invariants are:

- Type invariant is not checked for the contract, the **static** and the native methods:

- Type invariant is checked for constructors only before leaving them.
- The *_Invariant()* method has to be **protected** and non-**static**.

Inherited contracts

The inheritance of contracts defined with jContractor works as usual, inherited preconditions may be weakened, postconditions and invariants may be narrowed in the descendant classes.

Predefined exceptions

In case a contract is violated the exceptions *PreconditionViolationError*, *PostconditionViolationError* or *InvariantViolationError* are thrown respectively and the program terminates.

Use of contract classes

The previous contracts can be collected to a separate contract class. The name of such classes should be suffixed with *_CONTRACT* as, e.g. [Abe03]:

```
class Stack_CONTRACT extends Stack {
  private Stack OLD;
  private Vector implementation;
  protected boolean Stack_Postcondition (Object [] initialContents,
  Void RESULT) {
   return size() == initialContents.length;
  }
```

The jContractor will interpret the methods of contract classes as methods of the original class.

That is the way of defining contracts for interfaces and abstract classes too.

Using quantifiers in jContractor

The JaQuaL library (Java Quantification Library) can be used in jContractor contracts for description of conditions containing universal and existential quantifiers.

The following JaQuaL expressions can be used:

- With the help of the *ForAll.in(collection).ensure(assertion)* expression can be checked that the condition given in *assertion* for all elements in *collection* as, e.g.:

```
Assertion connected = new Assertion () {
  boolean eval (Object o) {
    return ((Node) o).connections >= 1;
  }
};
return ForAll.in(nodes).ensure(connected);
```

- The existential quantifier is achievable with the help of the expression *Exists.in(collection).suchThat(assertion)*.
- The function *Elements.in(collection).suchThat(assertion)* creates a new object of type Vector from those elements in *collection* for which the *assertion* is true.
- The function *Logical.implies(a, b)* is the logical implication: it returns true if the value of *a* is false or if both *a* and *b* are true.

The built-in assertions in JaQual are:

- *InstanceOf*: checks whether an object is an instance of a given class or not.
- *Equal* : checks whether two objects are equal or not.
- *InRange*: checks whether a given value is in a given interval or not.
- *Not*: is useful for negation of an expression.

Java Modeling Language (JML)

The *Java Modeling Language (JML)*[JMLc06] is a formal interface specification language, where the behavior of methods can also be prescribed.

The contract-conditions can be formulated with annotations in comments of the program code. This can be either a line, starting with //*, or a block /* . . . */. In the expressions Java notations can be used extended with special elements. This code can be compiled with *jmlc*, the *JML*'s own compiler. Since the contracts are in form of comments, the code can also be compiled with the original Java compiler.

Pre- and postconditions in JML

Pre- and postconditions can be written using *requires* and *ensures* respectively.

A small example[JMLc06]:

```
public class IMath {
  /* requires (* x is positive *);
     ensures \result >= 0 &&
          (* \result is an int approximation to square root of x *)
   */
   public static int isqrt(int x) { . . . }
```

The extension possibilities in *JML* are:

- Universal and existential quantifiers \forall and \exists
- General expressions for \sum, \product, \min, \max
- etc.

The specification of a method can look like in the example [Ald07]:

```
/* requires len >= 0
   ensures \result ==
                  (\sum int j; 0 <= j && j < len; v[j])
 */
float sum (int v[], int len) {
   float s = 0.0;
   int i = 0;
   while (i < len) {
  s = s + v[i];
  i = i + 1;
  }
    return s;
 }
```

The *old* function refers to the value of an attribute before entering a method, the *result* refers to the value returned as in the next simple example[JML13]:

```
/* requires amount >= 0;
    ensures
      balance == \old(balance)-amount && \result == balance;
*/
public int debit(int amount) {
...
}
```

Type invariants in JML

Type invariants can be given in *JML* after the keyword *invariant*, they are checked with the pre- and postconditions of the individual methods.

```
public class BankAccount {
final static int MAX_BALANCE = 10000000;
int balance;
/* invariant 0 <= balance &&
        balance <= MAX_BALANCE;
*/
...
```

Contracts for Java (cofoja)

Contracts for Java (cofoja) is a contract programming framework for Java. It is designed and implemented by Google[Mor11] based on Eiffel. It uses annotation processing and bytecode instrumentation to provide run-time checking.

The possible annotations are:

- Type Invariants – the conditions after the keyword *Invariant* describe the type invariants, these are checked on entry and exit of all normal public and package-private methods and at the end of constructors. Invariants are inherited, the invariant of the descendant is connected to the invariant of the parent with the logical *and*.
- Preconditions – the conditions after the keyword *Requires* describe the preconditions of the method, these are checked on entry of methods. Preconditions are inherited and can be weakened by the descendant.
- Postconditions – the conditions after the keyword *Ensures* describe the postconditions of the method, these are checked on normal exit. Postconditions are inherited and can be strengthened by the descendant.
- Exceptional postconditions – there is a possibility to check some conditions even on abnormal termination of method (when an exception has been thrown) using the keyword `ThrowEnsures Checked`.

There are two special keywords defined in *cofoja*:

- The keyword *old* can be used in postconditions (both normal and exceptional). It refers to the state of an expression at method entrance time, as, e.g.:

old(size()) == size() + 1

- The keyword *result* can only be used in normal postconditions, it refers to the method's return.

An example of *cofoja* in a contracted stack from [Le11]:

```
@Invariant("size() >= 0")
interface Stack<T> {
public int size();
@Requires("size() >= 1")
public T peek();
@Requires("size() >= 1")
@Ensures({
"size() == old(size()) - 1",
"result == old(peek())"
})
```

```
public T pop();
@Ensures({
"size() == old(size()) + 1",
"peek() == old(obj)"
})
public void push(T obj);
}
```

12.7.6 Ada 2012 language

A very important novelty in the new standard of the Ada 2012 programming language is its support of Design by Contract ([Ada12] and [Bar12]).

Pre- and postconditions

It is allowed to write pre- and postconditions with *Pre* and *Post* using the syntax as is in the example [Bar12]:

```
procedure Push(S: in out Stack; X: in Item)
  with
    Pre => not Is_Full(S),
    Post => not Is_Empty(S);
```

It is allowed to refer to the original value of a variable with the attribute *Old* as, e.g.:

```
Post => I = I'Old;
```

Type invariants

Type invariants are designed for private types to check the consistency of a type, as, e.g. [Bar12]:

```
type Stack is private
   with Type_Invariant => Is_Unduplicated(Stack);

type Disc_Pt is private
     with Type_Invariant => Check_In(Disc_Pt);
```

12.8 Summary

In this chapter we have discussed how program-correctness checking can be interpreted, defined, implemented and validated in practice, selecting Eiffel as our primary programming language for this purpose.

In particular, we have shown how formulas describing correctness properties can be described in Eiffel. Such formulas - also called in general assertions - are used in many contexts of Eiffel programs. Routine behavior and correctness are declared in terms of preconditions and postconditions. Class invariants serve to capture attributes of valid object instances in a given representation scheme. Loop variants and loop invariants are essential to reason over loop structures, to detect and eliminate typical pitfalls and to get the loops right in the first place. Likewise we have pointed out that check construct is an effective way of expressing different hypotheses at various places of the programs. These powerful concepts have been introduced step by step going through a concrete and practical example when implementing a LIFO stack class. The full source code of this class can be found as a reference at the end of this chapter.

Formal definitions of class consistency, check, loop, exception and class correctness have been also defined based on the work by Bertrand Meyer ([Mey91] and [Mey00]) and the current ECMA definition [ECM06]. The relationship of exception and correctness as well as the interpretation and definition of correctness in the context of inheritance have also been subject to discussion.

A number of source of issues – such as component dependencies, void- and type-safety, concurrency, etc. – which may lead to difficulties when arguing over correctness have also been identified and briefly discussed. Finally, how and to what extent language elements targeting program-correctness are present in contemporary programming languages also have been studied. Numerous examples were given for a wide range of languages including D, Oxygene, .NET languages, Java, etc.

The principles of program-correctness checking have been applied in practice for over one and a half decades. Along with the Design By Contract methodology, it has been proven to be the most useful in industrial environments. We could only encourage the reader to study and internalize the fundamentals and start applying them in practice. A clearly must-read classic in this topic is Bertrand Meyer's Object-Oriented Software Construction [Mey00].

12.9 Example source code

Finally we present the complete Eiffel source code of *MY_STACK* class.

```
class
  MY_STACK[G]
  create
    make
  feature{NONE}       -- representation
    container: ATTACHED ARRAY[G] -- to store elements
    item_count:     INTEGER -- number of elements
    capacity:       INTEGER -- maximal number of elements
```

```
  feature{ANY}
-- Creates a stack with capacity c. Capacity must be positive integer.
  make(c:INTEGER)
  require
    valid_capacity:  c>0
  do
    create container.make(1,c)
    item_count :=0
    capacity  :=c
  ensure
    created:     container /= void
    empty:    empty
    capacity_set:  capacity=c
  end
-- Puts a new item on the top of the stack
-- provided that the stack is not full.
  put(i:G)
  require
    not_full:  not full
  do
    item_count:=item_count+1
    container.put (i,item_count)
  ensure
    not_empty:      not empty
    new_item_on_top:  item      = i
    item_count_increased: item_count  = old item_count + 1
    same_capacity:    capacity  = old capacity
  end
-- Removes the top item provided that the stack is not empty.
  remove
  require
    not_empty: not empty
  do
    item_count:=item_count - 1
  ensure
    not_full:        not full
    item_count_decreased: item_count = old item_count - 1
    same_capacity:    capacity  = old capacity
  end
-- Reverses the order of items in the stack.
  reverse
```

```
  local
    lower_idx    :INTEGER
    upper_idx    :INTEGER
    lower_item   :G       -- lower item being swapped
    upper_item   :G       -- upper item being swapped
  do
    from
        lower_idx :=1
        upper_idx :=item_count
    invariant
      valid_range:    lower_idx + upper_idx = item_count +1
      valid_index:    lower_idx >=1
            and upper_idx <= item_count
            and lower_idx <= upper_idx+1

    until  upper_idx - lower_idx < 1
    loop
      lower_item   :=container.item (lower_idx)
      upper_item   :=container.item (upper_idx)
      container.put (lower_item,upper_idx)
      container.put (upper_item,lower_idx)
      check
        items_swapped:
            lower_item  = container.item(upper_idx)
          and upper_item = container.item(lower_idx)
      end
      lower_idx:=lower_idx+1
      upper_idx:=upper_idx-1
    variant
      remaining_interval: upper_idx - lower_idx + 1
    end
  ensure
  item_reversed:old container.item(1)=container.item(item_count)
      and old container.item (item_count) = container.item(1)
  end
-- queries
-- Obtains the top item of a non-empty stack.
  item:G
  require
    not_empty: not empty
  do
    result:=container.entry (item_count)
  end
-- predicates
```

```
-- Checks if the stack is empty.
  empty:BOOLEAN
  do
    result:= item_count = 0
  end
-- Checks if the stack is full.
  full:BOOLEAN
  do
    result:=item_count = capacity
  end
invariant
  valid_capacity:          capacity > 0
  valid_item_count:          item_count >=0
          and    item_count <= capacity
  valid_representation:  capacity   = container.capacity
  empty_definition:   (empty   implies  item_count=0)
        and (item_count =0 implies   empty)
  full_definition:      (full   implies  item_count=capacity)
        and (item_count =capacity implies full)
end
```

12.10 Exercises

Exercise 12.1. This exercise is originated from Bertrand Meyer [Mey00]. We have deliberately chosen it as a starting example, because it is very intuitive. Let us assume that a company in order to have a more efficient recruitment process, publishes its vacant positions along with Hoare-formulas in their usual form $\{Pre\}J\{Post\}$. This also helps the applicant select from jobs and find the most suitable one. The Pre precondition describes that the job J under what conditions is expected to be performed, whereas the $Post$ postcondition puts restriction on the result of the job. Describe the easiest and the most difficult job description up to J using Hoare-triples! Also provide some more alternatives for easy and difficult jobs with an explanation.

Exercise 12.2. Try to formalize a correctness specification of a Nespresso coffee machine!

Exercise 12.3. Why is the following correctness specification not correct?

```
{washing powder, water softener, pack of cigarettes, rinse water,
 electricity, water}
    washing program
        {clean clothes}
```

Exercise 12.4. Identify the problem with the routine below, which attempts to calculate the roots of a quadratic equation. Parameters are denoted by a, b and c following the usual convention. $x1$ and $x2$ are global attributes used to store the roots.

```
x1:REAL
x2:REAL
calculate_roots(a,b,c: like x1)
require
  discriminant_positive: b*b − (4*a*c)> 0
local   d:REAL;
do
  d:=sqrt(b*b)−(4*a*c))
  x1:=(−b+d)/(2*a)
  x2:=(−b−d)/(2*a)
ensure
  correct_roots:
    x1+x2 = − (b/a)
  and x1*x2 = + (c/a)
end
```

Exercise 12.5. Define the relationship of the Eiffel classes below from inheritance perspective, where formulas with the same name denote the same assertions in pre- postconditions and class invariants, and formulas do not contain references to functions. Furthermore formulas everywhere denote the full preconditions, postconditions and class invariants, and assuming that formulas differ from the simple $True$ and $False$ logical constants.

```
class A     class B     class C          class D
...         ...         ...              ...
m is        m is        m is             m is
require     require     require          require
 pre         pre         pre and pre'     pre or pre'
do          do          do               do
...         ...         ...              ...
ensure      ensure      ensure           ensure
 post        post and post' post          post and post'
end         end         end              end
...         ...         ...              ...
invariant   end         invariant        invariant
  inv                     inv and more   inv
end                     end              end
```

Exercise 12.6. The following function implements a simple addition routine. Is this routine loop-correct in a sense of Meyer [Mey91] and ECMA [ECM06] stan-

dard definition? Does the function work correctly with respect to its correctness specification?

```
sum(a,b:INTEGER):INTEGER
  require
    positive_b:   b>=0
  local
    i:INTEGER
  do
    from
      result:=a
      i:=0
    invariant
      result=a+i
    until  i=b
    loop
      result:=result+1
      i:=i+1
    variant
      b−i
    end
  ensure
    added:  result=a+b
  end
```

Exercise 12.7. Please fill the last two columns of the table below based on the information provided in context and condition columns. Specifically, in the "Expected Result" enter the result you expect, in the last column indicate the cause of the result and also make some observation on the routine being executed. An example is given in the first row.

```
class SOMETHING
feature{ANY}
do_something
require
  pre_something   -- 1.
do
  ...             -- 2.
  do_other_thing  -- 3.
  ...             -- 4.
  do_another_thing -- 5.
  ...             -- 6.
ensure
  post_something  -- 7.
```

```
rescue
  ...
  retry          -- 8.
  ...            -- 9.
end
feature{ANY}
do_other_thing
require
  pre_other_thing
do
  ...            -- 10.
ensure
  post_other_thing
end

feature {NONE}

do_another_thing
require
  pre_another_thing
do
  ...            - 11.
ensure
  post_another_thing
rescue
  ...
  retry          -- 12.
  ...            -- 13.
end

invariant
  some_invariant
end
```

Context	Conditions occurs/ State	Expected Result	Reason/Observation
Calling do_something	pre_something does not hold	Precondition violation	Calling module is incorrect by not fulfilling precondition of do_something
Calling do_something	post_something is not fulfilled		
Calling do_something	Class invariant does not hold		
After calling do_something at point 7	Class invariant does not hold		

Context	Conditions occurs/ State	Expected Result	Reason/Observation
At point 3, when calling do_other_thing	pre_other_thing evaluates to false		
At point 3, when calling do_other_thing			
At point 3, when calling do_other_thing			
Right after point 10	Class invariant does not hold		
Right after point 12	Class invariant does not hold		
At point 4, after calling do_other_thing			
At point 4, after calling do_other_thing			
At point 5, when calling do_another_thing	Class invariant does not hold		
At point 6, after calling do_another_thing	Class invariant does not hold		
At point 8	Class invariant does not hold		
At point 8	pre_something does not hold		
At point 9	pre_something does not hold		
Right after point 9	post_something does not hold		
Right after point 6	post_something does not hold		
Right after point 6	post_something holds		
After point 11	Pre_another_thing does not hold		
At point 12	Pre_another_thing is satisfied		
At point 12	Pre_another_thing is not satisfied		
Right after point 11	Class-invariant does not hold		

12.11 Useful tips

Tip 12.1. A job with a weaker precondition is likely to be performed under many more circumstances than the one with a stronger precondition. On the contrary, a weak postcondition makes the job easier, whereas a strong postcondition is bad news for the applicant.

Tip 12.2. Try to think in terms of inputs and outputs. What inputs (meant here in a broader sense) are required to make a cup of coffee?

Tip 12.3. Are the conditions specified in the precondition part necessary and sufficient?

Tip 12.4. Is the precondition strong enough? Or is it too restrictive?

Tip 12.5. Please recall how the sub-specification relationship on correctness specification of methods is defined. Please do not forget either how full invariants of classes are generated during inheritance. See Section 12.3.11 for reference.

Tip 12.6. Please review the loop-correctness definition described in the relevant Section: 12.4.3. Try to verify the validity of all formulas appearing in the definition. Can this routine be proven to be correct if the preconditions in the loop-correctness formulas are made stronger?

Tip 12.7. Think of whose responsibility is to satisfy the precondition, and whose is to ensure the postcondition? Who, when and how should satisfy the class-invariant? Please revisit the sections dedicated to class-invariant (12.3.8) and exception-correctness (12.4.5) if necessary.

12.12 Solutions

Solution 12.1. The easiest job is:

$\{False\}\ J\ \{Anything\}$

This is because under no condition the job J is to be performed. Notice that the postcondition can be anything (even *False*), since the job itself cannot be started.

Still a very easy job specification is:

$\{Anything\ but\ \textbf{not}\ False\}\ J\ \{True\}$

The job is performed, but irrespective of how it is done, the employer is always satisfied. Please note, however, in this case the employee must at least do something (no matter what) sometimes (based on the precondition) as opposed to the previous case when the employee does not have to do anything (possibly not even showing up at the employer's site).

The following job description looks quite easy as well, but in fact to do such job may be very uncomfortable and even sometimes dangerous:

$\{True\}\ J\ \{True\}$

Notice that the employee must perform the job always (even under conditions far from optimal, for example in an extreme case when the site is on fire). At least the employer is not picky and does not care about how the job is performed.

The most difficult is:

{True} J {False}

The job must be performed in any condition, but no matter how it is done, the result is never good enough.

Solution 12.2. A possible correctness specification of a Nespresso coffee machine:

{plugged in, water filled, capsule inserted, cup placed}
coffee making
{cup of hot coffee}

Solution 12.3. The precondition is missing the clothes part. The specification incorrectly promises clean clothes in its postcondition without putting any restriction on clothes in its precondition. Moreover a pack of cigarettes may make sense for a smoker to do the washing, but clearly it is not a must, therefore it should be removed from the precondition. By adding dirty clothes in its precondition, thus stating:

{washing powder, water softener, dirty clothes, rinse water, electricity, water}
washing program
{clean clothes}

resolves this issue. Note any type of clothes will not suffice - despite the utopian promise of washing powder manufacturers -, as we cannot expect a washing machine to produce clean and fresh clothes out of dad's dirty ones.

Solution 12.4. The method does not account for the case when *a=0*. Also it excludes the case when discriminant is zero. Either the precondition must be extended with a restriction ensuring that the equation is really quadratic. The modified precondition is as follows:

require
*discriminant_positive: b*b − (4*a*c)>= 0*
quadratic_equation: a<>0

Or the algorithm of method should be enhanced to handle special cases. To do so, the original method may be split into three methods as follows:

```
calculate_roots(a,b,c:like x1)
require
  quadratic_or_linear:  a/=0  or  b/=0
do
  if  a=0  then
    calculate_linear_roots(b,c)
  else
    calculate_quadratic_roots(a,b,c)
  end
ensure
  same_roots_if_linear:    a=0     implies x1=x2
```

```
  same_roots_if_disc_zero:  b*b-4*a*c   implies x1=x2
end

calculate_linear_roots(b,c:like x1)
require
  b_non_zero:  b/=0
do
  x1:=-c/b
  x2:=x1
end

calculate_quadratic_roots(a,b,c:like x1)
require
  quadratic:          a /= 0
  discriminant_non_negative:   b*b - (4*a*c)>= 0
do
  d:=sqrt((b*b)-(4*a*c))
  x1:=(-b+d)/(2*a)
  x2:=(-b-d)/(2*a)

ensure
  correct_quadratic_roots:
      x1+x2  = - (b/a)
    and  x1*x2  = + (c/a)
end
```

Alternatively, the method can be modified to be able to calculate complex roots. In such case no restriction applies to the discriminant.

Solution 12.5. No relationship between class A and B. Class A cannot inherit from B, because its postcondition is weaker. Neither can class B inherit from class A, as invariant of class B is weaker.

No relationship between class A and C either. Precondition of m in C is stronger, therefore class cannot inherit from A. But as class invariant of class A is weaker than that of class C, A cannot be a descendant of C either.

D may be a descendant of class A, since they have the same class invariant. Furthermore the specification of m in D is a valid subspecification of m in A. This is because

pre **or** *pre'* $\Rightarrow$ *pre*
post **and** *post'* $\Rightarrow$ *post*

Class C cannot be a descendant of class B, because the postcondition of m in C is weaker than that of m in B. Class C cannot be an ascendant of B either, because its class invariant is stronger.

Class D may inherit from B, because the specification of m in D is a valid subspecification of m in B. Moreover the class invariant is strengthened in D. The inverse relationship clearly may not hold.

Class D cannot be a descendant of class C due to the weaker invariant in class D. The inverse relation cannot hold either, because the postcondition of m in C is weaker than that of m in D.

Solution 12.6. The loop must fulfill the following conditions to be loop correct in the sense of [Mey91] and [ECM06]:

$\{True\}INIT\{INV\}$
$\{True\}INIT\{VAR>=0\}$
$\{INV$ **and then not** $EXIT\}BODY\{INV\}$
$\{INV$ **and then not** $EXIT$ **and then** $(VAR=v)\}BODY\{0<=VAR<v\}$

Where

$INIT := result:=a;\ i:=0$
$INV := result = a+b$
$VAR := b-i$
$EXIT := i=b$
$BODY :=result:=result+1;\ i:=i+1$

The routine is not correct with respect to the correctness definitions, because the second formula

$\{True\}result:=a;\ i:=0\{b-i>=0\}$

is not valid. Take, for example, $b=-1$. Note, however, that

$\{True\}result:=a;i:=0\{result=a+i\}$

- as described by the first formula - holds.

Nevertheless, thanks to its routine precondition such case(s) are excluded, resulting in the

$\{b>=0\}result:=a;i:=0\{b-i>=0\}$

alternative formula, which is valid.

$\{INV$ **and then not** $EXIT\}BODY\{INV\}$

also holds, since

$\{result=a+i$ **and** $i<>b\}result:=result+1;i:=i+1\{result'=a+i'\}$

$\{result=a+i$ **and** $i<>b\} => \{result+1=a+i+1\}$

Same can be proven for

$\{INV$ **and then not** $EXIT$ **and then** $(VAR=v)\}BODY\{0<=VAR<v\}$

First the decreasing part ($VAR<v$) is justified:

$\{result=a+i$ **and** $i<>b$ **and** $VAR=b-i\}$
$result:=result+1;i:=i+1$
$\{b-i'<b-i\}$

Clearly, $b-(i+1)=b-i-1<b-i$.
Moreover, 0<=b-i can also be seen.

Thanks to the precondition again, we can argue that the initial variant value is greater than or equal to zero.

$\{b>=0\}result:=a;i:=0\{b-i>=0\}$

with $i=0$ we obtain:

$\{b>=0\} => \{b>=0\}$

Assuming that the invariant $b-i>=0$ before executing the loop body, then because i is increasing from zero up to the value of b, the loop body $(result:=result+1;i:=i+1)$ is only executed if $i<b$ (loop exit condition $i=b$). Since i and b are both integers, thus follows that $i+1<=b$. Hence omitting the *INV* and *VAR* parts we get:

$\{i+1<=b\}result:=result+1;i:=i+1\{b-i'>=0\}$

Substituting i' with $i+1$:

$\{i+1<=b\} => \{b-(i+1)>=0\}$

$\{i+1<=b\} => \{i+1<=b\}.$

All in all, the routine works correctly.

Solution 12.7.

```
Context                    Conditions occurs / State
  Expected Result          Reason/Observation
=================================================================

Calling do_something       pre_something does not hold
  Precondition violation     Calling module is incorrect by not
                             fulfilling precondition of do_something

Calling do_something       post_something is not fulfilled
  Postcondition violation    do_something is incorrect not fulfilling
                             its postcondition

Calling do_something       Class invariant does not hold
  Class invariant violation  Class SOMETHING is incorrect

After calling do_something
at point 7                 Class invariant does not hold
  Class invariant violation  do_something is incorrect, it does not
                             preserve class invariant

At point 3,
when calling do_other_thing  pre_other_thing evaluates to false
  Precondition violation     do_something is incorrect

At point 3,
when calling do_other_thing  post_other_thing evaluates to false
  Postcondition violation     do_other_thing is incorrect

At point 3,
when calling do_other_thing  Class invariant does not hold
  Class invariant violation   do_something is incorrect since it does
                              not preserve class invariant

Right after point 10       Class invariant does not hold
```

```
  Class invariant violation    do_other_thing is incorrect

Right after point 12         Class invariant does not hold
  Normal completion            do_another_thing is not required to
                               maintain class-invariant

At point 4,
after calling do_other_thing   Internal exception occurs
  n/a                            do_something is incorrect

At point 4,
after calling do_other_thing   External exception occurs
  n/a                            Nothing wrong so far

At point 5,
when calling do_another_thing   Class invariant does not hold
  Non-issue, do_another_thing is non-exported

At point 7,
after calling do_another_thing   Class invariant does not hold
  Non-issue class invariant must be re-established upon exiting
do_something

At point 8                       Class invariant does not hold
  do_something is not exception-correct it should restore class invariant

At point 8                       pre_something does not hold
  do_something is not exception-correct, it should restore precondition

At point 9                       pre_something does not hold
  non-issue

Right after point 9            post_something does not hold
  non-issue.                     Exception is propagated.

In such cases postcondition is not required to be satisfied.

Right after point 6            post_something does not hold
  do_something fails to comply to its specification.

Right after point 6            post_something holds
  Normal completion

After point 11               Pre_another_thing does not hold
  Normal completion            Precondition is required to be satisfied
                               only upon routine entry

At point 12                  Pre_another_thing is satisfied
  Routine re-execution

At point 12                  Pre_another_thing is not satisfied
  Precondition violation      Pre_another_thing is not exception-correct

Right after point 11         Class-invariant does not hold
  Non-exported routines are not required to preserve class invariant.
```

13 Concurrency

In this chapter we concentrate on the most important language constructs for concurrent software development. We demonstrate and describe the most common problems with concurrency through a set of abstract examples, then introduce the elementary models of communication and synchronization: busy waiting, mutual exclusion, the issues with critical sections, the semaphore and the monitor. We also include a taxonomy of concurrent applications, programming languages and the advantages of and main issues in concurrent software. Widely used execution models are also covered. Furthermore, we introduce the available toolbox in a set of selected languages and frameworks including Ada, CSP, Occam, Java, C#, MPI and Scala.

"Concurrency is hard and boring. Unfortunately, my favored technique of ignoring it and hoping it will go away doesn't look like it's going to bear fruit." – A common verdict about concurrency

Writing concurrent code with today's library and tool support might seem to be easy, but writing *correct* concurrent code is even harder. Concurrent software development has its own issues, approaches, design patterns and solutions on top of the usual toolbox programmers have to deal with. It is surprisingly easy to put the pieces together and create software that is executed in a concurrent way and "almost" does the right thing – but inherently flawed at the same time and reveals serious problems through the eyes of a domain expert. These programs usually work only "by accident", and become extremely unstable often at the worst possible time: in a production environment, under heavy workload when it would be critical to keep the services up and running.

But what is the reason for that? In the next sections of this chapter we give a general overview of today's definition of concurrency, and of the selection of language constructs that were built to handle it; then we show the basic concepts and some common pitfalls; and finally we demonstrate how easy it may be to introduce additional program errors, but how hard it is to investigate them and find them, and how non-trivial it is to fix them.

So, does concurrency deserve attention at all? If dealing with it requires careful design and thoughtful implementation, is it worth the effort? Will a software developer ever encounter it in production code? The answer is definitely yes. As an illustration, let us see a few cases where concurrency, rather than anything else, solves the problem.

- *For desktop applications,* the user interface must be responsive even during computationally intensive operations as well (such as loading or parsing huge binaries, updating the software from the Internet, loading data from a database through sophisticated queries and so on). This is usually

done as a background process (i.e. by a different thread) and its progress is usually shown in a progress bar – seeing one is a good sign of something happening in the background.

- *For web applications,* the multi-user environment is inherent: a website is used by multiple users who simultaneously create, read, update or delete similar – or even the same! – resources, such as database entries, forms or online articles. Even the backend of these applications (think of web/REST services or web applications) must be prepared to work in a multithreaded environment. The reason is simple: the response time of the website must be kept at an acceptable level. Thus, it is unacceptable to serve the user requests sequentially in a queue.
  Note that the underlying database may often hide the issue of concurrency (in the form of *transactions*[1] for example), but as soon as any state is associated with the users in the forms of sessions, for example, the problems of concurrency comes to the fore.
- *Multicore systems* utilize the existence of multiple processing units, such as threads, cores or central processing units. Since even the cheapest computers (including mobile phones) tend to have multiple processing units, it is strongly recommended to be familiar with their peculiarities. Manufacturers, such as Intel for example, offer full-feature integrated development environments to aid programmers write platform-specific code easier, where the full potential of the system can be utilized.
- The idea that *databases must be sharded* is ever more popular. This means that they are stored semi-redundantly on several computers to make the storage physically possible (e.g. consider China where there are banks with more than 3 billion users) or to improve the performance how the data is used. Social sites, like Facebook, for instance, must build on this technique heavily.
- *Scalability* is also an important factor, getting a lot of attention nowadays. Grid and cloud systems (like what Amazon, Google App Engine, Windows Azure and Heroku offer at the moment) are becoming more and more popular because the computational power they offer can cover occasional load waves (peeks) on a service at a relatively cheap price.

We could continue the list with specific examples where programmers must be prepared to encounter concurrency, such as C++ compilation farms and

[1] According to Wikipedia, a transaction symbolizes a "unit of work" which is typically performed within a database. It is treated in a coherent and reliable way, independently of other operations. One of their greatest advantage is that they provide an *"all-or-nothing"* proposition: each work-unit performed must either complete in its entirety or have absolutely no effect. Transactions ensure that the database cannot get into an inconsistent state. Think of two users of a web application editing the same entry: the first user starts editing an article through a web form while the second simply deletes it from the database. Then, when the first user tries to save the data which may result in the alteration of several tables, all those modifications will be rolled back since none of them could have been committed.

test or 3D rendering grids whose roles are connected to a specific job. Most of these applications of concurrency help to drastically reduce the required processing time to an acceptable level, and being familiar with the basic concepts of concurrency is a huge advantage when it comes to using these tools.

But, instead of going into details, let us discuss the reasons why industrial application developers turned their attention to concurrent code in the first place.

A bit of history

At the very beginning of computer science, we had enormous computers at research institutes and universities that were capable of executing a single software at a time, for a mere 15 minutes at best. Technology quickly developed, and soon we had personal computers in almost every household which were capable of executing individual programs in separate processes (like the operating system and different user softwares). These processes had different stacks in the memory, and their execution was interrupted and rescheduled by the operating system in order to *show the illusion of concurrent execution* (think of writing a simple textual document and listening to music on the same single-core computer at the same time). There were several factors that motivated this enhancement, the most important probably being *resource utilization* and *convenience*. An example for resource utilization is when one of the processes are waiting for some expensive external I/O operation other processes might do some useful work on the CPU, convenience on the other hand, means that it is easier to create and maintain small software components that do simple tasks on their own than to write one multi-functional "god component" that deals with all sorts of tasks repeatedly. Having multiple processing units that can handle multiple processes concurrently is a more intuitive approach, especially if we take into account that developing stronger processing units has become a physically impossible task recently.

Processes were usually completely independent from each other; however, over time facilities have evolved to support the communication between them. We still utilize these mechanisms such as communication through sockets, files, signals and signal handlers, shared memory and so on.

After a while, it turned out that having a *degree of asynchronicity* in separate processes might also be useful. Think of today's applications: downloading a file within a browser should not interrupt the user in browsing the Internet by disabling the graphical interface during the process. This created the requirement for spawning *sub-processes* (such as threads, tasks or processes)[2] from an existing one.

In the field of research, the price of supercomputers shifted the attention to cheaper solutions. It is possible to approximate the computational power of

[2] These concepts are defined in Section 13.8.1. Here we only mean to illustrate that there are several different approaches to define a *sub-process*, but to avoid distraction from the topic, think of all of them as the elementary units of concurrent code execution: each of them being capable of running code concurrently.

such a computer through creating *distributed systems* by joining several cheaper machines with lower capacity into the same network, and by forming a grid or cloud system.

Today, even the low-range portable computational devices (smart phones or tablets) have multiple multicore processors, so the architecture is given to support this kind of development. The question is what do we have on the software side?

13.1 Reasons for concurrency

We can state that *parallel programming* helps us solve different parts of a task at the same time. The most common real-life situations where this might help us are the following:

- **It is more intuitive to approach the solution of a problem in a concurrent way** (e.g. server architectures, distributed services or graphical applications). In these situations there are usually at least two different tasks that have to be performed at the same time. In general, it might be easier and more intuitive to model every separate subtask with a different processing unit that can be executed with the other tasks in parallel – like defining *database reader* and *writer* processes separately.
- **The software has to be running on physically different devices at the same time** (like in the case of most Internet applications). For these applications concurrency is inherent as the software components must be explicitly modeled as separate entities.
- **It is more efficient to solve the given problem in a parallel manner.** In these cases the program can usually be solved in a sequential manner as well, but in cases there is a strict restriction on the throughput of the system that cannot be satisfied by using a single process. Utilizing multiple processors or grid architectures might result in a considerable performance gain – if done correctly.

13.2 An abstract example

"Most programs are so rife with concurrency bugs that they work only by accident." – Josh Bloch, author of the comprehensive book *Java Concurrency in Practice*

In this subsection, we highlight some common issues related to concurrency ad illustrate them through an abstract example. We use a simple pseudo-code in order to focus on key concepts without distracting attention away by various language constructs.

The problem we investigate here is a refined version of [KV06], which is based on the work of [AO97].

The problem

Let us assume that there is a number of different processes (n) and each is working on its own data structure, an infinite stream (noted by $s_1, s_2, \ldots, s_n$). The streams are not shared between the processes. The input of the program is a conditional function *cond* that evaluates the elements of the streams as $true$ or $false$. The processes must search in the streams for the first element that evaluates $true$ in the streams. The output should be the *first* index of the element in the ith stream. More specifically, we expect the first j index for which $cond(s_i[j])$ holds, where $1 <= j <= n$.

This is a somewhat trivial problem: it can be done easily in a sequential manner. However, if concurrency comes into the picture, the problem becomes much more complicated.

For the sake of simplicity, let us suppose that (1) the elements in the streams are all different, (2) the element can be found in one of the streams, (3) the element occurs only once, and (4) there are only two processes ($n == 2$). That is enough for demonstration purposes, still, enough to encounter potential problems. This is the smallest number of preconditions we need to demonstrate potential concurrency issues.

The first attempt for finding the solution

The pseudocode shown in Listing 13.1 is our first attempt to solve the problem. First, we define a global variable which is shared among all the different processes. This is a simple boolean variable that indicates if we have found the required e element already or not. The loop variable, which is smaller or equal to n, is the searched position. Then, we start two similar processes which work as shown below.

```
found = false          A    found = false
i = 0                  B    j = 0
while (!found) {       C    while (!found) {
  found = cond(s1[i]) D         found = cond(s2[j])
  i++                  E          j++
}                      F    }
```

Listing 13.1: The first Solution

This approach is intuitive, as we do what we would do in the case of a sequential program: we set the initial conditions ($found$ for $false$, the local i, j indices for the threads to 0), iterate through the elements of each stream and set the $found$ flag until we find the required element. Pretty straightforward.

However, here we run into our first problem: in some of the execution paths the program may result in an *infinite loop* (in the case of infinite input streams, or an index out of bounds error with finite data structures)!

The reason is the following: let us suppose that one of the processes get a higher priority in the beginning. When the p_1 process starts, executes the loop statements several times, and then finds the required element at line D.

If meanwhile the operating system interrupts it in this specific instant of time, takes the right of execution from p_1, and starts the p_2 process. It simply reinitializes the shared value to $false$, causing the infinite loop or an index out of bounds error.

The problem with this case is the multiple initialization of the shared variable. Let us try to eliminate it through a second attempt.

The second attempt for finding the solution

In the second attempt, as it is shown in Listing 13.2, before creating any processes, first we create the global variable which is shared among all of the processes, and then perform the initialization separately:

```
found = false
```

Listing 13.2: The second attempt - Initialization

Otherwise we leave the activity of the processes unchanged, as shown in Listing 13.3.

```
i = 0                A    j = 0
while (!found) {     B    while (!found) {
  found = cond(s1[i]) C     found = cond(s2[j])
  i++                D      j++
}                    E    }
```

Listing 13.3: The second attempt

The initialization issue is resolved. However, when we start running the code repeatedly, we encounter our second problem: in some of the executions, the program may result in an **infinite loop** again.

The reason is the following: let us assume that the p_1 process initiates, executes the loop statements a few times, then finds the required element at line C. It sets the global $found$ variable to $true$, and right after that, the operating system that schedules the ordering of process executions gives the right of execution to p_2. After it has begun, it evaluates its first element, and

sets the $found$ flag back to $false$ again. In the end, neither p_1 nor p_2 terminates, since we have skipped the single occurrence of the searched e element.

This error is very special, since it does not often cause program errors, only at specific execution paths. These kinds of errors are common in concurrent code, happen frequently in production, are easy to make, and may be extremely hard to find and resolve. We will discuss this issue in details, later in Section 13.4 through a widely known example.

The third attempt for finding the solution

The drawback of the previous version has been that we need to update the state of the same variable in every iteration. Let us examine what happens if we add a guard condition before the assignment as it is shown in Listing 13.4:

```
i = 0                  A    j = 0
while (!found) {       B    while (!found) {
  if (!found)          C      if (!found)
    found = cond(s1[i]) D      found = cond(s2[j])
  i++                  E      j++
}                      F    }
```

Listing 13.4: The third solution

This is a concrete example for a common pattern called a *check-then-act* compound statement, which is commonly encountered in production code (as in the case of lazy initialization for example).

The problem with this solution is similar to the previous one, we still get infinite loops. A critical program execution may be the following.

Process p_1 and p_2 starts executing their actions. Let us consider a case when the element is not found yet and both of them executing the loop statements are at line C. Since both conditions are met, both of them executes the update of the $found$ variable. In the case p_1 finds the element at position i, setting $found$ to $true$, and right after that p_2 also updates the $found$ variable from $true$ to $false$, resulting in an infinite loop again.

The fourth attempt for finding the solution

To fix the previous problems we may eliminate the usage of the shared variable. Let us try to update its state once, and only in the case the element is found as shown in Listing 13.5.

Surprisingly, this program works.

What we can conclude is that we need an ability to define a set of instructions to be *atomic*: uninterruptable by other processes until all of them are executed. This way the problematic situations where a shared resource (e.g. the $found$

```
i = 0             A    j = 0
while (!found) { B    while (!found) {
  if (cond(s1[i])) C    if (cond(s2[j]))
    found = true   D      found = true
  i++              E    j++
}                  F    }
```

Listing 13.5: The fourth solution

variable in the example) is accessed and updated simultaneously by multiple processes (i.e. they *interfere* with each other) is avoided. Let us define such a new language element as shown in Listing 13.6.

```
await (b) {
  stmt_1
  stmt_2
  ...
}
```

Listing 13.6: A new language element

The semantics of this new construct is that the process is on halt (becomes *blocked*) until the specified b logical condition is met. Then, it executes the $stmt_1, stmt_2, \ldots$ statements as uninterruptable, atomic operations, independently of other processes.

However, there is another theoretical issue with the example above, namely the issue of *unfair scheduling*. Since we have not defined how the actual execution of the processes happens, it might be that one of the components does not progress. Let us suppose that we divided the input streams based on some sort of heuristics (e.g. one of them contains only the positive numbers, while the other contains the negative ones; or one of the streams contains the odd numbers, while the other the negative numbers), and search for a specific number.

If one of the processes gets higher priority over the other one, the program may not terminate since the other process may not have even started. An example is when we search the first number that is greater than ten in two infinite streams where the first one contains the positive numbers, while the second one contains the negative numbers. If the process iterating through all the negative numbers always gets an advantage over the other one, we get an infinite loop again. This phenomenon is similar to *starvation* which is covered in details in Section 13.7.2.

What we can do here, however, is explicitly force the system to execute the processes periodically after one of them has been executed for a given number

T of iterations. Let us try it as our next attempt. To simplify the problem let us assume that $T = 1$, but the solution can be generalized easily.

The fifth attempt for finding the solution

In order to make the runtime system execute the defined threads with approximately the same speed, we can introduce an explicit scheduling. Let us define a new shared global variable *next* which defines which process should be executed.

Before the processes are initialized, we declare this variable with the value of 1, indicating that process p_1 is allowed to take the next action (Listing 13.7).

```
found = false
next = 1
```

Listing 13.7: The fifth solution - Initialization

Then, in the body of the processes (shown in Listing 13.8), let us use a new notation to block the process which is not allowed to progress while the other is acting, and then pass the right of execution to the other one.

```
i = 0                A  j = 0
while (!found) {     B  while (!found) {
  await (next == 1) { C    await (next == 2) {
    next = 2         D      next = 1
  }                  E    }
                     F
  if (cond(s1[i]))   G    if (cond(s2[j]))
    found = true     H      found = true
  i++                I    j++
}                    J  }
```

Listing 13.8: The fifth solution

We now encounter another trap of concurrent programming called *deadlock*. The code above behaves as the examples before – it is working most of the time, but surprisingly, not in every case. Take a look on the following execution path.

Let us consider the case when p_1 and p_2 processes work on their own for a while without any problems, both of them are at line C, and the value of *next* is 1. As a result, p_1 progresses to G while setting *next* to 2. Right after that p_2 is also activated, it sets *next* to back again to 1 while also reaching line G. Now let us assume that p_2 processes forward to C again, while passing back the execution to p_1. If p_1 finds the searched e element, it terminates as expected. However, p_2 process never terminates, as it is waiting for *next* to be

2 again, which condition is never met. This special state is a common problem called a *deadlock*: the program cannot progress further because it is waiting for a condition (acquisition of a resource or an external signal) that is never satisfied.

The sixth attempt for finding the solution

The last piece of the puzzle is to add one additional statement at the end of each process that allows the progress (and termination) of the other process as follows. It is illustrated in Listing 13.9.

```
i = 0                A    j = 0
while (!found) {     B    while (!found) {
  await (next == 1) { C     await (next == 2) {
    next = 2         D       next = 1
  }                  E     }
                     F
  if (cond(s1[i]))   G     if (cond(s2[j]))
    found = true     H       found = true
  i++                I     j++
}                    J    }
next = 2             K    next = 1
```

Listing 13.9: The sixth solution

This solution is an acceptable implementation for the original problem at last.

What we can conclude, is that even a simplified problem we have addressed in this section that is trivial to implement in a sequential manner may lead to several unintuitive and previously unexpected problems. Programmers who write concurrent code should bear in mind several additional issues (for example correct termination, while avoiding liveness hazards like deadlocks and livelocks are also important) in addition to providing anotherwise bug-free software.

13.3 Fallacies of concurrent computing

There are some common misbeliefs and misconceptions about concurrency which spread through word-of-mouth. It is practical to mention a selection of these myths and misbeliefs when speaking about concurrency for a better understanding.

- **If it is concurrent, it is quicker (partially true)** A common misbelief about concurrency is that just because we execute the same program in a concurrent way it is going to be quicker. Unfortunately, the efficiency is heavily task-specific, and in some cases it may turn out after running a

benchmark that regardless of our serious efforts, *the parallelized version of a software is actually slower*. The reason is that *concurrency comes with an overhead* (creating processes requires scheduling and managing them on the level of the operating system, switching contexts between them requires intensive modifications of the stack, and each new thread created has a small memory footprint which may accumulate, thus cause additional measurable load on the computer and so on).

As discussed in Section 13.4.1 in detail, there is a theoretic upper bound for the maximum performance gain by executing a task concurrently.

Speed is influenced by many factors like the architecture on which a multithreaded software is running (undoubtedly, a single computer having multiple CPU cores handles them better). To get an objective result, make the software parametrizable, make benchmarks several times and apply statistics. Believe only what you can measure - and of course, handle it carefully.

- **The structure of the program does not have to be change to run concurrently (almost a complete lie)** Many of the junior programmers fall into this trap. They get an assignment to write a concurrent software component which they first try to accomplish in a sequential manner. However, when they produce the final prototype of the software and try to apply concurrency, they realize that it does not work, and that the whole program must be completely rewritten from scratch, because the codebase becomes so complicated thus hard to maintain. They realize most of their production time has been wasted, because all their work ends up in the thrash.

  Concurrent softwares have different issues, they require different building blocks, approaches, thus, and thus different design patterns to apply. It requires a cautious and detailed design phase to build such a system, and it is often easier to rewrite malfunctioning components from scratch than to refactor existing sequential ones. As we will see, there is an emerging set of tools that targets this issue, but even their usage need a general understanding of elementary concepts about concurrency. They can also introduce serious software or performance problems if used blindly.
- **It is easier to write a sequential prototype, and then rewrite it to a parallelized version (almost a complete lie)**. See the previous reasoning.
- **I do not care about concurrency, since I will not encounter it in my work (complete lie)** Writing production code does raise the need of at least basic understanding of concurrent programming. We hope in the introductory sections we were able to demonstrate this. See Section 13.4 for a short summary of situations where mere programmers may typically encounter concurrency.
- **Concurrency is easy as a pie. Even if I make any mistakes, I will debug it easily (complete lie)** Producing parallel applications and keeping the

number of bugs at an acceptable level is a hard and time consuming task. However, it is far from impossible: several industrial/scientific case studies exist from grid/cloud systems to microkernel-based operating systems, through services like e-mailing which show that these tasks can be accomplished if done correctly. They require extra work and carefully managed components. The lack of carefulness may result in several hard to find, non-deterministic (i.e. impossible to debug) program errors. The reason is explained in the next section.

As a message take this home from this chapter: concurrency comes with a price, it is far from trivial, needs structural changes in the software to support concurrency, and the introduced problems are non-deterministic.

13.4 Possible number of execution paths

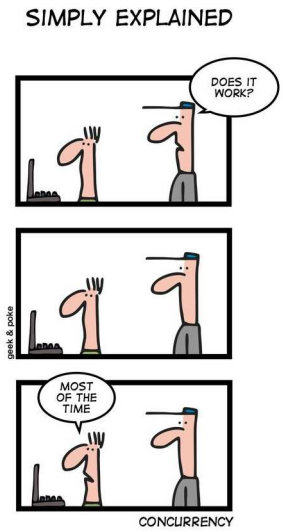


Figure 13.10: A *Geek and Poke Webcomic* issue about concurrency by Oliver Widder. `http://geekandpoke.typepad.com/`

We already mentioned non-determinism several times by now, let us consider a concrete example. It is a simple but shocking example of Brett L. Schuchert and Martin Fowler [Mar08] that perfectly illustrates the basic problems with concurrent programming.

Let us consider the following trivial class definition in Java:

```
public class Counter {
  private int ctr = 0;
  public void increaseAndPrint() {
    ctr++;
    System.out.println(ctr);
  }
}
```

This example has nothing Java-specific, the results of the discussion can be generalized for a wide set of languages. Java is one of the most widespread languages according to the TIOBE index[3] and easy to understand even if the Reader has no prior experience with the language.

As shown in Listing 13.11, here we are set to use the same instance of this class only from two different parallel execution units (these are called `Threads` in Java, but we could use POSIX threads in C with the `fork()` function, `Threads` in the .Net platform, and so on). A simple demo application and is shown in Listing 13.11.

```
public class Main {
  public static void main(String[] args) {
    Counter counter = new Counter();

    class Incrementer extends Thread {
      public void run() {
        while (true) {
          counter.increaseAndPrint();
        }
      }
    }

    Incrementer process1 = new Incrementer();
    process1.start();

    Incrementer process2 = new Incrementer();
    process2.start();
  }
}
```

Listing 13.11: Using the Counter from different threads

The output might be something unexpected,[4] see Listing 13.12.

[3] `http://www.tiobe.com` The TIOBE Programming Community Index is an *"indicator of the popularity of programming languages updated monthly. The ratings are based on the number of skilled engineers world-wide, courses and third party vendors found by popular search engines. Observe that the TIOBE index is not about the* best programming language

```
1
3
4
4
5
6
...
```

Listing 13.12: Sample output

Now that is interesting, isn't it? The first unexpected result is that 2 for example is missing; the second one is that some of the values (like 4) is appearing multiple times. How is this possible? What is the reason behind it? In order to answer those questions, we have to discuss a bit about *atomicity*.

An *atomic operation* is an action which happens all at once. If it consists of multiple operations, the state transitions between the embedded instructions are not visible to any other processes.

If we analyze the actual bytecode,[5] we will see something like shown in Listing 13.13.

```
0: aload_0
1: dup
2: getfield      #2
5: iconst_1
6: iadd
7: putfield      #2
...
```

Listing 13.13: Bytecode representation of the source

As Listing 13.13 illustrates, some operators, which are seemingly one statement in a high level language (the ++ in this case), are actually compiled into multiple statements (4 bytecode statements in this case). Roughly speaking a read method, that gets the value of the given field, moving the integer constant 1 to the register, performing the addition, and putting the value back to the field. Detailed description of these instructions can be found in [LYBB13].

or the language in which most lines of code *have been written"*

[4] Your mileage may vary on different hardware, operating system, Java version, system load and the interstellar constellation of the stars, but the same trends should definitely appear in any output.

[5] This can be obtained by running the *javap* tool which can be found in all versions of the Java Development Kit. With compiled languages such as C++, similar results may be obtained in the actual assembly code.

If we consider two threads, while the operating system is executing them, it has the right to interrupt the execution of a thread *at any point*, to immediately continue with the one that has been interrupted before, and to resume from the state wherever it was stopped.

The problems raise here by a common construct called *read-modify-write*. Let us say both threads are interrupted just after calling `increaseAndPrint`, and the `counter` variable has the value of n. The operating system allows execution of reading the value n for *process*1, and immediately passes the right of execution to *process*2. Now, the value it reads will also be n and updates the content of the variable to $n+1$, so it prints $n+1$ to the standard output as a result. Right after this, the execution is passed back to *process*1 which also writes $n+1$ into the common variable, and also prints $n+1$. This is the first and foremost issue with concurrent programming humorously illustrated on Figure 13.10.

It can be proven[6] that the total number of execution paths for n statements through T threads is:

$$\frac{(nT)!}{n!^T}$$

Evaluating the formula for the current example results in 924 different execution paths, which is rather frightening for the first time. This illustrates that even a trivial example consisting of 5 lines of code can cause an exponential blow in its complexity when executed concurrently. Another interesting issue is that if we replace *int* with *long*: a trivial change that changes only the primitive type of the variable increases the total number of execution paths to 12.870, as both reading and writing a 64 bit variable takes 2-2 statements to execute.[7]

This is the reason for non-deterministic program execution and for the faulty program behaviors. It is relatively easy to introduce such errors into more complex examples, and extremely difficult, if not impossible, to localize a reported program error from the billions of possible execution paths (even to reproduce the problematic state to analyze it for the source of errors in standard manners like debugging is a hopeless task in most of the cases).

The question of the *missing number* is now up to you. Can you provide a reason for this issue? A hint: try reasoning when both threads are after the state of reading the value of the common variable.

13.4.1 Amdahl's law

There is a widely known theoretical upper bound for the possible performance gain by concurrency which is often apostrophed as Amdahl's Law[8] after [Amd67].

[6] Based on [Mar08].

[7] Do not think this is only a Java-based problem. The .NET platform has the same issue with the CLI platform, moreover, reading the EAX register in assembly also requires the same amount of statements, effecting all low level compiled languages like C/C++.

[8] `http://en.wikipedia.org/wiki/Amdahl's_law`

The performance gain by paralellizing an application is heavily bounded by the ratio of concurrently executable parts that are independent of each other. A few examples are illustrated on Figure[9] 13.14.

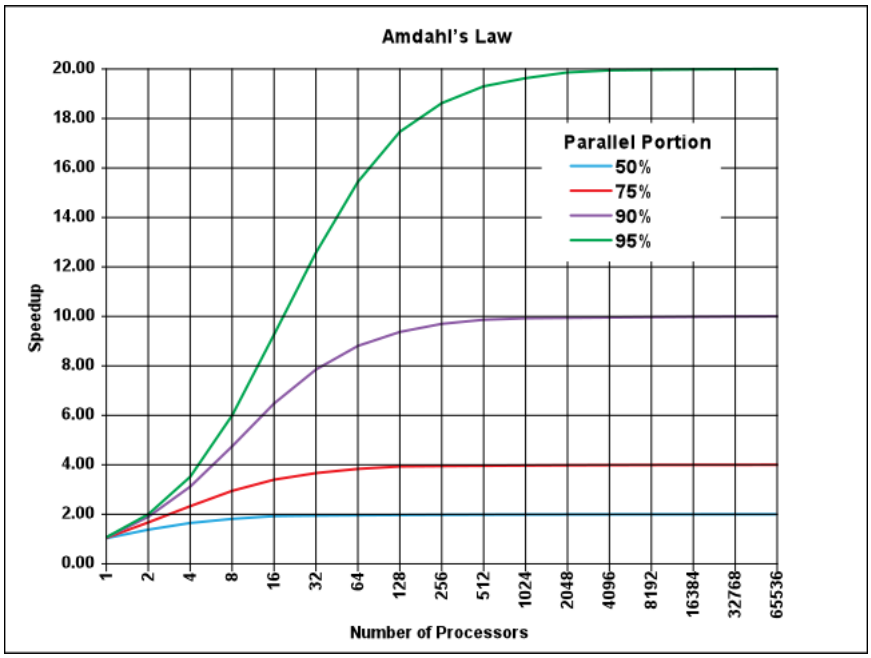


Figure 13.14: Illustration of how performance gain changes by Amdahl's Law

As an example, consider a software that has a running time of 20 hours when run sequentially on a single computer. In a very optimistic case when there is only 1 hour of running time that cannot be executed in a parallel manner, but the remaining 19 hours of work (95%) can be potentially executed concurrently. This is an exceptional case that is almost never encountered in real life. This shows that not even in an ideal case can we reduce the total running time below 1 hour: thus, regardless of the number of additional parallel processes, by increasing them to any further extent the maximum performance gain cannot be greater than the factor of 20.

More formally, if P is the portion of the program code that can be executed concurrently, then $(1 - P)$ must be executed sequentially (i.e. it cannot be parallelized). The total maximum performance gain that can be achieved by using the number of N different execution units is:

$$\frac{1}{(1-P) + \frac{P}{N}}$$

[9] Image is adapted from the Wikipedia article about Amdahl's Law.

In the case of $N \to \infty$, the result converges to $\frac{1}{1-P}$. This is the indicator of the maximum performance gain achieved when exploiting parallel architectures.

13.5 Taxonomy of concurrent architectures

There is a widely known classification of computer architectures commonly referred to as *Flynn's taxonomy*,[10] proposed by Michael J. Flynn in 1966 [Fly72]. Since the effective hardware architecture poses an interface for the software compiled to that specific architecture, it has a great influence on the software.

Flynn proposed a simple taxonomy based on the number of instruction streams and the number of associated data streams:

	Single instruction stream	Multiple instruction streams
Single data stream	SISD	MISD
Multiple data streams	SIMD	MIMD

These architectures have their own specialties.

- *Single Instruction, Single Data Stream (SISD)*
  A simple sequential computer which does not exploit concurrency at assembly level. It has a single processing unit which fetches only one instruction and is able to operate on a single data stream at a time. A single-core personal computer is a perfect example of this hardware category, however, these architectures have almost disappeared by today. The interesting thing is that even this architecture was able to *imitate parallel software execution*. The solution was that the operating system was able to switch between the processes (e.g. with a Round Robin scheduling), thus providing the illusion of parallel program execution - of course, the context switching of different processes was an expensive operation: all the local variables, stack, registers had to be saved for the previously executed process and loaded for the next one.
- *Single Instruction, Multiple Data Streams (SIMD)*
  In some specific cases, parallelism is more convenient on the level of data. Examples include array processors or Graphical Processing Units where the same algorithm is commonly executed but, for instance, on different images. One of the most popular SIMD parallel computing platform is NVIDIA's CUDA. This kind of processing is often called *data parallelism*.
- *Multiple Instruction, Single Data Stream (MISD)*
  This is a somewhat uncommon (yet sometimes very important) architecture that usually characterizes mission critical systems. The main reason why it might be useful to execute different instructions at the same time for the same data is fault tolerance. The Space Shuttle flight control

[10] `http://en.wikipedia.org/wiki/Flynn's_taxonomy`

system for instance was built on this architecture [MSH11][Kno93]. It was called a *fly-by-wire system*: sensor data were preprocessed by 5 different processors whose output was given to a *voting system* before showing them to the astronauts. The voting system then evaluated these values and ignored data that seemed to be erroneous (either because of measurement errors or faulty hardware components).

- *Multiple Instruction, Multiple Data Streams (MIMD)*
  Probably the most common architecture, where multiple processing units execute different instructions on different data stored either in a shared memory or their own distributed memory space. Today's devices (even handheld devices) usually fall into this category.

In a concurrent software, parallelism can be implemented at different abstraction levels. The most essential is at the level of the instructions which means that two or more instructions are executed at the same time (e.g. in the case of multicore systems when coding in a low level language such as assembly or C). Parallelism can also be implemented on the layer of subroutines (these are commonly referred to as *tasks*) which encapsulate parallelism of instructions or other subroutines. Another abstraction layer may be the parallel execution of different components of a software (as in the case of a distributed system) or different softwares. In this chapter, our main focus is the parallelism of instructions and subroutines (i.e. on the level of instructions or methods).

13.6 Communication and synchronization models

As we have seen in the introductory sections of this chapter, two of the main problems that have to be solved when creating concurrent applications are *communication* and *synchronization* between the components.

During communication one of the components can access information from another, concurrently executed component. This communication can be performed through several media, like shared memory, message passing, files, sockets, signal handlers and so on.

When using a shared memory space, some of the variables of the application are accessible for multiple processes. In this case, a process can communicate with another by writing data into the shared memory so that the other one can read it. What is crucial in this situation is that the processes must be synchronized – if both of them are writing for the same shared memory space (into the same variable), the result will be undefined (think of a simple *read-modify-write* statement).

In the case of message passing, a process is allowed to send a message directly to another process or to a group of processes. This communication might either be asynchronous or synchronous. In the asynchronous communication model, the process does not wait for the recipient to accept the message; it is only

blocked till it sends the message usually through the channel. By contrast, in the synchronous communication model (often referred to as a *rendezvous*) the execution of the caller process is blocked until the recipient has acknowledged the incoming message and has sent its reply to the message. The processes are not allowed to progress further until the communication is completed in its entirety.

Several alternatives exist for the aforementioned communication models, but those are usually problem specific solutions. For instance, it might be useful to implement a delayed synchronous communication model where the client processes send asynchronous messages to the dedicated server process, usually under heavy load. Until the server is capable of processing the incoming message, the clients can continue their own work. Whenever the server is able to process the message, it can contact the client in a synchronous way to notify it about the results. If the client needs the result of the message before the server is able to construct it, it is blocked. This mechanism is known as a future object and is commonly used in most of the concurrent programming libraries.

13.7 Mutual exclusion and synchronization

Beyond communication, synchronization is another critical issue. By synchronization we mean a mechanism that allows aligning parallel processes, i.e. it can define an ordering in which the instructions of the parallel processes are executed.

As we have seen before, the ordering of the executed instructions is important in a concurrent environment. If a process is waiting for an input from another process, it has to wait until the counterpart sends its reply to the request message. If multiple processes would like to access the same resource (may it be a shared variable, file, database connection, or anything else), they have to wait until it is released by any previous processes that was using it. We have illustrated this facility in one of the introductory examples by waiting for a specific condition to be met with the *await* statement.

Restricting access to shared resources raises several unique problems, for which we must be prepared when implementing the specific solution.

13.7.1 Deadlocks

One of these unique problems is known as *deadlock*.[11] As we have seen in Section 13.2, a deadlock is encountered when a group of processes are mutually waiting for each other (or for an external signal that never appears), and none of them are able to make any progress.

It is relatively easy to encounter such a situation. Let us assume that we have two different resources A and B, and two processes need both of them in order

[11] `http://en.wikipedia.org/wiki/Deadlock`

to perform their tasks. In this case, it is easy to cause a deadlock by ”playing” with the resource acquisition order. If one of the processes locks resource A, then try to lock resource B; if, however, the other process locks resource B, try to lock resource A simultaneously, we get a deadlock.

However, an important thing to note, is that to encounter a deadlock, we need to satisfy *all* of the following conditions (also known as *Coffman conditions* [CJS71]) met simultaneously in a system:

- *Mutual exclusion*
  There is a shared resource that cannot be handled by multiple processes in a given instant of time.
- *Hold-and-wait locking*
  A process is allowed to wait for the acquisition of an additional resource that is currently used by another process while it is already owning a different shared resource.
- *No preemption*
  The runtime environment (like the operating system or a virtual machine) is not allowed to interrupt a process and force it to release a resource it is currently owning.
- *Circular dependencies*
  It is possible in a given instant of time that a set of processes are mutually waiting for each other to release a resource. For example, there is a p_1 process that is waiting for a resource that is held by p_2, while p_2 process is waiting for a resource that is held by p_3, and so on, while there is a p_n process at the end that is waiting for a resource that is held by p_1.

As a corollary, we can conclude that if we can explicitly eliminate all of these conditions, our software is theoretically deadlock-free.

Unfortunately, this is not always feasible, or an explicit deadlock handling is required for which several approaches exist:

- One of the most common approaches is to ignore them completely by assuming a deadlock will never occur in the software;
- Another approach is to focus on the detection of encountered deadlocks in mission critical systems. If a deadlock is detected, the system may try self-healing itself. This approach can be done by model checking and interrupting a set of running components, either by resource preemption or by process termination, as an example;
- Breaking any of the Coffman conditions would also be sufficient. Unfortunately, this can be hard, or even impossible, in most real-life situations. The most commonly used solution involves breaking mutual exclusion by well known algorithms like the Banker’s algorithm [Dij82].

13.7.2 Starvation

Another important issue is *resource starvation.*[12] In this case, multiple processes would like to acquire the same resource, but a set of certain processes may never get the privilege of holding he required resource.

A commonly referred example is when two group of people meet in the middle of a narrow corridor where only one man can pass. If one of the groups is too polite, it will not be able to go through the corridor until all the groups from the other side has left. If processes behave the same way, and we know that there is an endless flow of processes on the other side, the processes on hold will never get a chance to progress. Another example could be the circular road intersection: if there is a continuous flow of vehicles from any of the entrances, all the other joining roads are stopped.

13.7.3 Techniques for synchronization

Communication and synchronization issues must be solved by paying attention to the above mentioned problems.

Synchronization (if it is available in the specific language) can be solved in the form of rendezvous (synchronous message passing), but several other alternatives also exist. Before we move on to describe the different synchronization techniques, we need to provide some definitions.

In case of shared variables, the question is how the different processes can use these variables with mutual exclusion, – i.e. this is necessary to prevent processes interfere with each others' instructions involving the shared variable. The typical example is a simple read-modify-write instruction $x = x + 1$ executed by p_1 and p_2 threads (for theoretical description, see Section 13.2 and technically in Section 13.4 in detail). While the p_1 process reads the value of the x variable, the p_2 process does the same while executing, but executing the update, and storing the result simultaneously with p_1. This way the value of x is increased by 1 only, not as it was expected by 2. So if the processes do not execute the required operations without mutual exclusion, there might be a serious information loss in the background.

If we apply mutual exclusion on variable x, *only one process* can access it in the given instant of time, and thus we have guaranteed the safe usage of the shared variable.

A *critical section* is as a set of instructions (or blocks of statements) where the execution of the software is restricted to a single process. It is used to ensure safe update of a shared memory space [Pet81].

In the example above, the $x := x + 1$ instruction is a critical section in the program, and mutual exclusion ensures that only one process can execute it. In order to ensure mutual exclusion, all the processes have to perform a managed *entry protocol* before entering into the critical section, and must perform an *exit*

[12] `http://en.wikipedia.org/wiki/Resource_starvation`

protocol after leaving it. This way we can prevent entering multiple processes into the critical section.

13.7.4 Solutions for managing critical sections

There are several possible solutions for managing critical sections, of which we give a few examples now.

Busy waiting

The busy waiting is probably the most intuitive solution. Until a given condition is met, a process continuously reevaluates it in a loop, see Listing 13.15.

```
while (!p) {
  sleep 100;
}
```

Listing 13.15: Busy waiting

The disadvantage of this solution is that it wastes valuable processor time. Busy waiting is primarily used to handle critical sections, which is one of the most essential synchronization tasks. Under special circumstances, however, on specific hardware architectures busy waiting may be a preferred solution.

Semaphores

In concurrent programming, and also in the area of operating systems, the *semaphore*[13] is an essential tool to protect and manage processes which have to execute the instructions of a critical section. The concept of semaphores was introduced by the Dutch E. W. Dijkstra [GD68], and has been included in several languages since the introduction of ALGOL 68, either as a language primitive or a library component.

In its most generic form, a semaphore is a simple integer number with an associated waiting queue. An `s` semaphore basically supports two different methods, one of them is symbolized by `P(s)` while the other with `V(s)`. These canonical names come from the initials of the Dutch words *verhogen* (*increase*) and the portmanteau *prolaag*, the short form of *probeer te verlagen* (literally meaning *try to reduce*).

The instructions embedded within the `P(s)` and `V(s)` method calls are executed in an atomic, uninterruptable way. Atomicity here grants that the state

[13] `http://en.wikipedia.org/wiki/Semaphore_(programming)`

transitions within the encapsulated instructions (i.e. the inner state) are not observable to any of the other processes except the one actually executing it [And91]. For the other processes it seems as a single instruction.

The semantics of the `P(s)` and `V(s)` methods are defined as follows:

- `P(s)`: If `s > 0`, then decrease the value of `s` by one; otherwise the process that wants to execute `P(s)` is included in the end of the waiting queue of the semaphore;
- `V(s)`: If there is any processes waiting in the queue of the `s` semaphore, the first of them is activated; otherwise increases the value of `s` by one.

The initial value of the counter associated to the semaphore specifies how many processes are allowed to enter into the guarded critical section. The strict semaphore (also called binary semaphore) allows only two values, 0 and 1. This construct can be utilized to implement mutual exclusion.

The advantage of the semaphore is that compared to the busy waiting, a more efficient software can be implemented. The processes waiting for any shared resource are not wasting the performance of the computational unit, instead they are simply blocked and inserted into a waiting queue.

The disadvantage of using a semaphore is that it is against structured programming, and it is easy to cause deadlock-prone situations (i.e. situations where all the parallel processes are blocked because they are waiting mutually for each other, and thus the program cannot progress any further).

A binary semaphore can be used for mutual exclusion as the following example demonstrates:

```
s = 1;      // Initialization - done separately

// Code executed by the processes
P(s);       // Executing the entry protocol
...         // Critical section
V(s);       // Executing the exiting protocol
```

The binary semaphore is also commonly referred to as a lock for this reason. Also, its methods are often referred to as locking in the case of `P(s)`, and unlocking in the case of `V(s)`.

Monitor

The *monitor*[14] is a more structured language construct than the semaphore, that encapsulates data structures (variables and functions or methods defined over them) in a way that considering the set of all the subroutines defined at the level of the monitor, only one of them is allowed to be active in a specific instant of time. Accessing its inner state is also restricted to the interface it defines. The

[14] `http://en.wikipedia.org/wiki/Monitor_(synchronization)`

only way to access the inner representation of the monitor from the outside can be done through the defined interface of the monitor.

Monitors may be illustrated perfectly through a slightly modified illustration of Bill Venners [Ven00]. Imagine a library building with a secret room which only one person is allowed to occupy in a given instant of time. In the room there are rare and unique resources whose secret is sought by visitors. Unfortunately, the librarian keeps a close attention to who and when is granted the privileges to enter into the room to access the secret knowledge (resources). Visitors (processes), in order to get entrance to the secret room, must wait in a queue in front of the room within the library. When there are no visitors in the secret room and a new visitor arrives into the queue, it is granted access to the room, otherwise he must wait for his turn. From the time a visitor enters this room until the time it leaves, it has exclusive access to any data in the room. Entering the library is called *"entering the monitor"*, while entering the special room inside the building is called *"acquiring the monitor"*. Occupying the room is called *"owning the monitor"* and leaving the room is called *"releasing the monitor"*. Leaving the entire building is called *"exiting the monitor"*.

Monitors may also be *"Wait and Notify"* (sometimes called a *"Signal and Continue"*) monitors. In this case, visitors may suspend their work in the secret room and have a nap on one of the armchairs (*wait signal*). The armchairs are so comfortable that visitors sleeping do not wake up by themselves, but must be awaken by an external signal (i.e. by another visitor with a *notify signal*). After the notification the notifier will release, and the notified visitor will acquire the monitor. If there are only sleeping visitors in the secret room, the librarian may allow any new visitor to enter the area.

At an abstract level, the monitor utilizes a set of conditional variables. For each conditional variable, it also defines a waiting queue, and like the semaphore it supports two methods called `wait` and `signal` whose semantics are defined as follows:

- `c: condition`;
- `wait(c)`: the process executing this function call becomes blocked and is put in the waiting queue of the `c` conditional variable. The responsibility to start this process is delegated to someone else.
- `signal(c)`: If there is a process in the waiting queue of the `c` conditional variable, the first one of them is started so that it can continue its execution. Otherwise, nothing happens.

At first sight, conditional variables are very similar to semaphores, but there is a very important semantical difference between these concepts. That is, the conditional variable does not own an associated counter, i.e. it does not keep track of the incoming `signal` requests as the semaphore does. Moreover, if we call the `signal` method at a specific point in time when there is nobody waiting in the queue of the conditional variable, the signal event perish without any consequence [Hoa74].

Conditional critical section

A second alternative for the semaphore is the conditional critical section, which was introduced by Brinch Hansen at about the same time as the concept of the monitor[Han72].

This construct is a critical section where processes can enter and access shared variables only in a managed way. Entering the critical section is guarded by a conditional expression, and, if the specified conditional expression is unsatisfied, the execution of a process is suspended before the guarded block.

Outside the critical section none of the processes are allowed to access the variables of the critical section, and all the other processes have to wait till none of the processes are in the critical section *and* the conditional guard expression is satisfied.

Several other tools, concepts and language constructs have emerged to handle the issues of synchronization, which we cannot wish to cover in details here. However, we will provide a detailed summary of languages at the end of this chapter to demonstrate language specific approaches (e.g. the protected objects in Ada).

13.8 Taxonomy of languages supporting concurrency

Languages supporting concurrency can be classified from several points of view. A concurrent software is inherently non-deterministic, meaning several ways it can be executed. What is essential is an efficient and robust solution for a given problem in a concurrent environment that uses distributed/shared resources.

Several approaches have been developed to support the execution of concurrent applications. It is possible to classify the different language constructs based on the execution model [PV92]. In the next section, we will enumerate the execution models which can be interpreted for low and/or high level concurrency. In some cases, a entire class of programming languages might be associated with one of the models shown.

- *Control driven computation* The primary focus is on the monitoring of the started processes with common commands like `fork()`, `join()` and `wait()`. Most imperative languages like Java fall into this category. Operating systems which follow the POSIX standards also include such mechanisms to handle processes and to expose a C/C++ application programming interface for that.
- *Data driven computation* The execution of the software is based on the structure of the data and the associations encoded in it. The execution of the concurrent components starts when all the input data are available for the process. The Irvine Dataflow (ID), the Value Algorithmic Language (VAL) and the Intel Concurrent Collections framework is based on this execution model.

- *Demand driven computation* The processes are executed when there is an explicit signal (request) from another process, thus the execution is defined by messages. Some Prolog-based languages like Parlog or Concurrent Prolog are examples of this execution model.

Another class of concurrent programming implements the processing of arrays and matrices in parallel computer architectures (MIMD). These architectures have vector- or matrix processors, and always handle similarly structured data. The communication between the processes is synchronous. This is one of the oldest approaches for concurrent programming, and with regard to the languages which support this kind of execution model, we distinguish three different approaches to solving the issues of concurrency.

1. The different FORTRAN-variants (like FORTRAN IV, CFT and Cyber FORTRAN) extract the concurrently executable parts of the software from the `do` loops, and thus they automatically try to parallelize the program.
2. The parallel parts can also be declared explicitly (e.g. in the case of CFD and DAF FORTRAN), but this requires a deep understanding of the underlying hardware architecture that executes the software. The syntax and semantics of these languages offer great support for writing code on MIMD architectures, but the drawback is that the software written this way is not portable.
3. The third group of programming languages solves the issues of concurrency independently from the underlying hardware architecture (like the Actus language).

Although the programming languages mentioned above are some of the first languages, they contain language constructs which were later adopted, and even improved by modern languages. As an example, languages following the data driven computation approach (such as C* and Dataparallel-C) used the same array structures,. Let thus us now look at the concepts briefly.

Cyber 200 FORTRAN is a high level programming language where it is the task of its compiler to determine which parts of the sequential software can be executed concurrently. Basically, it marks the body of simple and nested `do` loops which work on arrays to be executed concurrently. This operation is called vectorization [Cyb83]. The CFT programming language works along the same concepts as Cyber FORTRAN, the difference being that it cannot handle nested loops. Software written in these languages are not portable, meaning they can be executed only on the hardware architectures they were compiled for.

In the case of CFD FORTRAN, the central processing unit decodes the next instruction, and either executes it, or sends it for execution to any of the subprocessors. The central processing unit basically executes only the arithmetic operators that are used to determine a memory address or to handle loops. This language was designed for a specific SIMD (*single instruction, multiple data*)

architecture. Programmers, however, do have control over the process: they can specify which processor should execute a specific instruction by their index. The language constructs contain a special `if` statement which can be used to compare vectors and update their values concurrently. The language also contains concurrent versions of the `and`, `or`, `not`, `any` and `all` operators. Combining them with the `wait` statement the data exchange can be synchronized.

VS FORTRAN language uses the concept of tasks. Tasks have separate memory space and behavior. The concurrent code can be specified by the programmer, but even the compiler may trace down the possible concurrent execution paths. The language operates with virtual processors that are associated with physical processors by the operating system. The result is that this language has become independent from the underlying hardware and can be ported to and make run on different architectures. The concurrent code is automatically generated by the compiler, and the different loop constructs and iterations can be run in an asynchronous way. The prerequisite of concurrent execution is that the iterations must be independent from each other and this information must be available at compilation time. Common operators are: `originate`, `terminate`, `schedule`, `wait for task`, `wait for any task` and `wait for all tasks`. To define parallel execution paths, the following operations are supported by the language: `parallel do`, `local`, `dobefore`, `doevery`, `doafter`, `exit`, `parallel sections`, `section`, `end sections`, `parallel call`, `wait for all calls`. The VS FORTRAN offers a blocking facility in order to handle critical sections.

The Actus programming language ([Per79] and [PCM83]) is a descendant of the Pascal language that supports the handling of parallel data structures. It is independent of the architecture as it supports both vector and matrix processors. Concurrency can be handled explicitly, meaning the computational resources may theoretically be exploited to the maximum. It has a built-in **array** type capable of storing both simple and complex types that can be handled concurrently. For concurrent processing of an array the index of the first and last elements are required only, and these happen automatically. These arrays also support the *rotate* and *shift* operators.

Languages are often apostrophed as are programming languages for multiprocessor machines. Their processes usually perform the same operations for similarly structured data again and again. Hence, they are sometimes called *synchronous languages*.

The motivation of modern languages to introduce new language constructs to handle concurrency could be expressed by the following questions:

- On which level would the language like to support concurrency?
- How should the language support concurrency? Should it be done through shared memory or message passing?
- Should it be possible to create side-effects?
- How should language support concurrency in terms of communication?

- What kind of elementary concurrent language constructs should be included into the language?

Programming languages may be classified into groups based on the questions above, although there are languages that can be classified into multiple groups at the same time.

- Languages that support shared variables – namely,
  Pascal Plus [Wel79], Concurrent Pascal [HP79], Modula-2 [Wir83] and [BP90];
- Languages that support message passing where the communication is handled through *send*−*receive*-like operators: CSP [Hoa78], Occam [Bar92], Scala [OSV11];
- Data driven languages using shared memory: VAL [Ack79], Dataflow [Arv78], Lucid [WA85];
- Object-oriented parallel languages: Emerald [Hut87], Pool-T [Ame87];
- Functional programming languages that contain parallel language constructs: Clean [Pla99], many Haskell- or ML-variants (D-Clean [ZHH06], Glasgow Parallel Haskell [LT01], pH [NA01], JoCaml [FFMS01], Scala [OSV11]);

13.8.1 Processes, tasks, threads: Concurrent execution units

So far in this chapter we have referred to *processes* as the elementary units of concurrent execution. The process is one of the most important concepts of parallel programming. In the next sections, we will take a brief overview of languages from the point of view how they define this building block. Our overview will also tries to demonstrate the wide variety of developed parallel language constructs, and the different interpretations of the same theoretical concepts. The overview does not attempt to be comprehensive.

As we will see, the interpretation, usage and actual implementations of processes in different languages are of a wide range. Moreover, several naming conventions exist. In the following sections we will introduce processes with regard to communication as well as it is one of their most important properties.

The Modula-2 language ([Wir83] and [BP90]) is built upon the concept of processes. The processes communicate through shared variables and *signals*. Signals are operations exported from processor modules. They are used for synchronization and cannot contain any message. A process can *send* or *wait* a signal from another process. Further operations of processes include *startprocess*, which starts a new parallel process, and *awaited*, which is a property of a given signal and shows if there is a process waiting for the given signal.

Signals are different from semaphores in a fundamental way – namely, when there is a signal that is not awaited by any of the processes, it is considered as a no-operation. The language contains language construct called *coroutines* [Con63]. It implies parallel processes capable of activating each other. When a

process is suspended, it resumes execution at the same position where it was suspended before when the right of execution was passed to another coroutine.

The application in Listing 13.16 presents a solution for a simplified variant of the producers-consumers execution model by Modula-2 processes.

```
MODULE ProcessDemo;
  FROM Terminal IMPORT
    Read, Write;
  FROM SYSTEM IMPORT
    PROCESS, NEWPROCESS, TRANSFER, ADR, SIZE;
  VAR p, c: PROCESS;
      workspace: ARRAY[1..100] OF CARDINAL;
      char: CHAR;
  PROCEDURE Consumer;
  BEGIN
    LOOP
      IF char="$" THEN EXIT END;
      Write(char);
      TRANSFER(c, p);
    END
  END Consumer;
BEGIN
  NEWPROCESS(Consumer, ADR(workspace), SIZE(workspace), c);
  LOOP
    Read(char);
    TRANSFER(p, c);
  END
END ProcessDemo.
```

Listing 13.16: A simplified solution for the producers-consumers problem in Modula-2

The *TRANSFER(p1, p2)* function interrupts the execution of the *p1* process, suspends it and passes the right of execution to the *p2* process.

The SIMULA 67 [Lam82] programming language is built on the concept of coroutines as well. The following SIMULA 67 example also demonstrates the complexity of execution. The *coroutine* in this context is an instance of an arbitrary class which is cooperating with other objects and can be temporarily stopped and resumed later. An instantiated object (*coroutine*) depends on the object which has created it. The new object can pass the right of execution back to its parent by the *detach* instruction, while the parent can activate the detached object again by the *call(X)* instruction. The execution can be passed from one coroutine (object) by the *resume(X)* instruction to another X coroutine (object).

When the execution of the coroutine is resumed, it is continued at the exact position where it was suspended previously. The lifecycle of a coroutine object starts when it is instantiated, and the object starts executing the instructions in its body. Such an object is in an *active state*, and it is considered as an *attached object* to its *generator* (the object which performed the instantiation).

The coroutine can suspend its execution by the *detach* instruction for the benefit of its generator. These objects are considered to be in *detached state* (but not in *terminated* state). An object can give up the right of execution voluntarily to an arbitrary coroutine by the *resume(X)* instruction. The object then may gain the right of execution again either by a *call(X)* instruction from its generator or a *resume(X)* call by a "sibling-process". The execution of the object is *terminated*, when the execution of its body reaches the **end** keyword. In a terminated state, the execution of the object cannot be started again either by the *call(X)* or the *resume(X)* instructions. However, the object (including its member variables and operations) may still be accessible.

In the ALGOL 68 [BW79] programming language instructions separated by comas (*collateral clauses*) and in **begin**–**end** blocks can be executed concurrently, thus becoming *parallel clauses*. The **begin**–**end** block is not finished until all its enclosed instructions are executed. The synchronization of actions is controlled by Dijkstra-semaphores through the *sema* keyword.

In Concurrent Eiffel [Eif13], concurrency is implemented by the *separate* keyword. Calling any method of a *separate* object does not block the caller, it can progress further (however, if it is a function call, the result is awaited). Any method or class can be annotated by the *separate keyword*. If an instance of such a class is instantiated, all its methods are run at a new processor.[15] An argument may also be defined as *separate* to ensure controlled access resulting in automatic mutual exclusion. These arguments cannot be used by a separate object by default, but using *exception* or *yield* can ease this restriction. The Eiffel language has several (sometimes unimplemented) variants supporting concurrent software development like Distributed Eiffel or Cameo [PP08].

The Modula-3 language [BW96], despite of it being the member of the Modula-family, defines processes as *threads*. These are communicating through shared memory space. The tools of concurrency are in the *Thread* module in the form of *Fork* and *Join*. A sample application is given in Listing 13.17.

The mutual exclusion is implemented by the *Lock* instruction and the *Mutex* objects. The example in Listing 13.18 demonstrates the usage of locks through a simple decrement function.

[15] In Concurrent Eiffel, the processor is an abstract notion to an autonomous concurrent execution unit. It sequentially executes instructions on one or more objects.

```
...
TYPE
   Closure = Thread.Closure OBJECT
                a, b, result: INTEGER;
             OVERRIDES
                apply := Start
             END;
VAR
   cl := NEW(Closure);
   thread: Thread.T;
   a: ARRAY [1..4] OF INTEGER;
   max: INTEGER;

PROCEDURE Start(cl: Closure): REFANY =
BEGIN
   cl.result := Max(cl.a, cl.b);
   RETURN NIL
END START;

BEGIN
   cl.a := a[1]; cl.b := a[2];
   thread := Thread.Fork(cl); (* evaluates the maximum *)
   max := Max(a[3], a[4]); (* of 4 numbers concurrently *)
   EVAL Thread.Join(thread);
   max := Max(max, cl.result);
END
```

Listing 13.17: Defining processes in Modula-3

With the help of the *Channel* abstract type it is possible to create channels used for message passing (synchronous communication) after initialization. These channels are unbuffered, unidirectional and synchronized.

We also encounter the concept of threads in the Java programming language [Har98], where processes are implemented by the *Thread* class. An instance of this class represents a process. The body of the process is defined by the instructions defined in the *run()* method. An arbitrary number of threads can be instantiated and started. It is also possible to set different priority levels to these threads, and to suspend, restart or stop them. Java threads can also be synchronized by the *join()* method. Monitors can also be found in the language, and they can be accessed by the **synchronized** keyword. Communication is done by shared variables. In a synchronized block, both *cooperation* (through *wait()*, *notify()* and *notifyAll()*) and *mutual exclusion* are supported through associated monitors. The language offers an extensive built-in API support for multithreaded programming (for details, see Section 13.14).

```
...
TYPE
   Counter = MUTEX OBJECT
             n: CARDINAL;
             ...
           END;
PROCEDURE Decrease(counter: Counter): BOOLEAN =
BEGIN
   LOCK counter DO
      IF counter.n > 0 THEN DEC(counter.n); RETURN TRUE
      ELSE RETURN FALSE
      END;
   END;
END Decrease;
...
```

Listing 13.18: Mutual exclusion in Modula-3

In Delphi subclasses of the *Thread* class can be executed concurrently. To implement synchronization and communication, a class must declare a method responsible for the execution, in which it can implement the communication of the concurrently executed code with a *synchronize(⟨MethodName⟩)* instruction.

In the BETA [MMPN93] programming language a new execution thread is created from an object by the `P: @|Activity` command. Such a thread is executed concurrently with other threads. The `fork` command is also a tool to execute the same set of commands concurrently. BETA has a predefined pattern representing a semaphore. An example in Listing 13.19 embraces two concurrent components, `A` and `B`:

```
(# S: @semaphore;
   A: @ | (# do st1; S.P; st2; S.V; st3 #);
   B: @ | (# do st4; S.P; st5; S.V; st6 #);
do S.V; A.fork; B.fork
#)
```

Listing 13.19: Defining semaphore in BETA

In the example above, component `A` may execute the `st1` or `st3` concurrently with the `st4` or `st5` statements of component `B`. By contrast, component `A` and `B` cannot execute the `st2` and `st5` statements at the same time (note the conventional `P` and `V` notations for the monitor operations).

Monitors are available through the `Monitor` pattern, while synchronization is done by the `Port` pattern, whose `entry` operation is used to synchronize different processes.

Another important implementation of processes is *tasks*. The PL/I [GI90] programming language supports concurrent programming on the level of instructions, and it is one of the early languages that introduces the concept of tasks. The tasks are concurrently executed program units. Method `P` may call method `Q` in a concurrent way by the `Call Q(`⟨*parameters*⟩`) Task(X)` instruction. Method `Q` notifies `P` through the `X` event variable if it finished its execution. Method `P` may test the `X` variable anytime, i.e. it can query if method `Q` is still under execution or it has been terminated already. Task `P` may also perform a task rendezvous with `Q` by the `wait(X)` statement. Task `Q` inherits all variables of `P` so they can share data.

Processes are represented by task-objects in the Ada programming language. An Ada tasks may have an *entry point* which may be called from another task. Every task has a body describing the activities of the given task. In its body, every entry point is associated with an **accept** instruction. When execution of the body reaches that point, the task is ready to accept the call of another for a rendezvous. The communication is synchronous, and the caller task is suspended until the acceptance of the rendezvous, the execution of the corresponding **accept** instruction. An entry point may also have **in**, **out** or **in out** parameters. When a task calls the entry point of another task, its execution is suspended until the end of the rendezvous. This mechanism may be avoided by using a *timed entry call*, where a maximal waiting time can be specified for a rendezvous. If the called task cannot response in the given time interval, the execution of the caller is resumed. The language offers an extensive API support for handling tasks, for the details, see Section 13.10.

Processes may also communicate through channels by message passing. The CSP/K and SP/K languages are both extensions of the PL/I programming language. The message passing is synchronous: process `P` sends the `x` message to process `Q` by the `Q ! x` instruction. This message is handled within process `Q` by the `P ? y` instruction that stores the message sent by `P` in variable `y`. A process is described in the following form: `name: procedure options (concurrent)`. Entry points can be specified by the `name: process;` declaration. The language is discussed in details later, see Section 13.11.

The Occam [Bar92] programming language is interesting because in general it has two elementary building blocks: processes (`proc`) and channels (`chan`). The most important constructions are enumerated below.

1. `SEQ` specifies instructions to execute in a sequential order (as the reader can see in Listing 13.20). It is important to note that *this is not done implicitly* as it is common in other imperative languages.

```
SEQ
  x := x + 1
  y := x + 3
```

Listing 13.20: Defining sequential instructions

2. `PAR` executes instructions concurrently (see Listing 13.21).

```
PAR
  c1 ! data1
  c2 ! data2
```

Listing 13.21: Defining concurrent instructions

3. `ALT` is used to define alternative (*guarded*) instructions bound to different conditions. The example in Listing 13.22 is waiting data either from the `input` or `observer` channel. If there is a message from `input`, it increments a counter, but if there is a message from `observer`, it returns the current counter.

```
ALT
  input ? data
    SEQ
      ctr := ctr + 1
      ...
  observer ? info
    SEQ
      stat ! ctr
      ...
```

Listing 13.22: Defining alternative instructions

Standard branching instructions like `IF`, `WHILE` and `SEQ`... `FOR` are also supported by the language. Variables declared in the form of `TYPE variablename:`, where the `:` character binds the variable to a given process. Processes are declared by `PROC process`, and channels declared by `CHAN prot channel`. Communication is implemented by message passing. The language is discussed in Section 13.12.

Channels are also used by the Par C programming language. It is an extension of the standard C language with the channel data type along with the **par** and **select** instructions. Communication between processes is done by channels or global variables. The syntax of **par** is identical to the **for** loop, but it spawns parallel processes. A channel is able to accept any type as a message, while the **select** statement is similar to the **select** statement of the Ada programming language discussed earlier.

The Super Pascal programming language, member of the Pascal family is another example using channels. The **forall** keyword is used to start the same statements in multiple instances concurrently (the number of processes is dynamically evaluated during runtime) and the **parallel** keyword is used to execute a given number of statements concurrently (as you can see it in Listing 13.23 below). Processes communicate over channels. The regarding language constructs to use are **open** to initialize a new channel, **send** to put a message on the specified channel, and the **receive** method to accept a message from the channel.

```
forall i := 1 to 100 do
    processConcurrently(i)

parallel
    producer() | consumer()
end
```

Listing 13.23: Concurrent execution of statements

The CC++ language comes with extensions for standard C and C++. The *parbegin–parend* expression is used to define concurrently executed code blocks. Synchronization is done by special variables declared as *synch* typed variables. The CC++ utilizes both parallel and sequential language constructs. The cooperation of these components are done by atomic functions. Another such an extension is Cilk++ [BJK95].

13.8.2 Monitors

Monitors are also an important element of inter-process communications. Since there are several interpretations for the same concept, there is also a wide range of specific implementations. For example, in Concurrent Pascal [HP79] the monitor is an abstract encapsulation of variables shared among several processes, and classes are abstract data types. An instance of a class may be attached either to a process or a monitor. The monitor ensures the protection of its data fields, while its methods and functions are associated with a waiting queue ensuring that only one of them may be active at a given instant of time. The execution

of a method of the monitor is initialized with *init*, may be suspended with *delay* and resumed with the **continue** instructions.

The Portal programming language is another example from the Pascal family that supports concurrent programming with the help of monitors. In Modula-2, the shared variables used to communicate between processes are encapsulated within monitors. The CSP/K and SP/K languages use monitors to implement mutual exclusion and the `wait-signal` construct for synchronization. In the Java programming language every object has an implicit monitor associated to implement cooperation (`wait-notify-notifyAll`) and mutual exclusion (through `synchronized` blocks).

13.8.3 Alternative approaches

Functional languages

These languages follow a set of different paradigms discussed in detail in Chapter 15. We include only a brief overview here. An elementary concept of functional languages is *mapping* and *reducing* of expressions. Processes considered as separate function calls in these languages (in pure functional languages there are no mutable state thanks to *referential transparency*). Thus, the evaluation of function calls and expressions is done concurrently along explicit annotations in the source code or implicit evaluation strategies. In these cases, processes are created dynamically, for instance in the VAL [Ack79], ParAlf, Concurrent Clean [PE01], Distributed Haskell and JoCaml [FFMS01] programming languages.

These principles started emerging in the functional programming world, but have become popular in mainstream programming languages too for their usability. Examples include the countless *map-reduce* framework for processing large datasets which semerged after Google's model described in [Lam08].

Logic programming

This follows an alternative approach, where the evaluation of rules is done concurrently (see Chapter 16 for details). Some Prolog variants such as Concurrent Prolog and Parlog follow this approach.

The Parlog programming language is based on the *And/Or-Parallelism.* As in Prolog, its semantic is defined by first order predicate logic, but the programmer is allowed to express concurrency explicitly in a declarative framework. A *goal* is declared by a number of literals which can be separated by a few additional operators. The *And* operator is symbolized by the `,` operator and evaluates expressions from left to right concurrently. Clauses are separated by the `.` or `;` operators. The expressions separated by the `.` operator are evaluated concurrently, and if there was no resolution for the required goal, the evaluation continues by the clauses declared after the `;` operator (*Or-Parallelism*). The concurrent components communicate through shared variables. Communication in this case

is called *incomplete messages* or *back communication*: the process that would like to communicate binds a shared variable as a *nonground nonvariable term* which is later filled by the receiving process. Synchronization is automatically managed, cooperation is asynchronous, parallel components are instantiated automatically, and recursion is also allowed.

Strand is a high-level symbolic language environment uses concurrent tasks. Its syntax is similar to Prolog. Synchronization is done through shared variables which are also used for mid-process communication. The Strand programs are always executed concurrently.

Object-oriented languages

Sequential, object-oriented languages, which satisfy the following three conditions:

1. The program starts with the instantiation of exactly one single object;
2. When an object sends a message to another object, its execution is suspended until the reply is received (i.e. the caller waits for the acceptance and processing of the sent message);
3. An object is active only when it is executing a method that is processing an incoming message.

Concurrency can be achieved in these languages by leaving any of the aforementioned conditions, allowing multiple processes to be active at the same time:

1. The program may start with the execution of multiple objects;
2. When an object sends a message to another it does not wait for the answer (i.e. it is not suspended) and immediately continues its execution (*asynchronous communication*)
3. The objects are not waiting passively for incoming messages, but have their own execution logic in their bodies. This execution then may be suspended at certain points in their bodies to answer incoming messages, which is done in the form of rendezvous (i.e. the caller and the receiver are synchronized).

The concurrent building blocks of the Pool-T [Ame87] programming language are objects. The communication is done in the form of rendezvous or message passing. The objects are created dynamically and their execution is asynchronous.

Parallel processing environments

An example for distributed parallel processing environments is when Linda, a coordination and communication model, has two kinds of tuples: active and passive. Active tuples represent processes, while passive tuples represent data. Tuples are handled in a shared environment called the *tuple space*. Data in the

tuple space is accessible to any process (or restricted to a group of processes), but it is not associated directly with any of the processes by default. All the communication is done through the tuple space: if a process wanted to send a message to another one it simply writes a tuple into the space that the other process can read and take.

There are basically four operations allowed on the tuple space that processes can execute:

eval *(processname)*: creates a new process of the specified name that evaluates tuples and writes the results back to the tuple space.

out *(P1, . . .Pn)*: The **out** instruction creates a new *(P1, . . .Pn)* tuple and writes this new data structure into the tuple space. It is also possible to specify elements by using an active tuple instead of a value. In this case, the associated elements will be evaluated by the execution of the given processes, and when all these processes are executed, the active tuple with a writing on it, "freezes" into the tuple space, thus becoming a passive tuple.

in *(P1, . . . , Pm)*: The **in** instruction is used to read and take passive tuples from the tuple space. Its arguments may be either values or variables. The instruction searches for tuples in the space where the tuple components are of the same type as *P1, . . . , Pm*, or in the case of values, they are identical. If there is no such tuple on the space at that moment, the caller process become suspended until a tuple that satisfies the supplied parameters is available. The localized tuple (or a random one if there were multiple alternatives) is removed from the tuple space, and the variables with the `?` wildcard are assigned the current tuple values.

rd *(P1, . . . , Pk)*: The **rd** (read) instruction is similar to the **in** instruction with one exception – namely, it does not remove the localized tuple from the tuple space (i.e. it is *non-destructive*).

Linda was originally implemented in C and FORTRAN, but several libraries are available for different languages like CppLinda for C++, Erlinda for Erlang, JavaSpaces and TSpaces for Java, PyLinda for Python or Rinda for Ruby. Linda has also inspired several projects like Piranha, Trellis and Linda Program Builder.

In a heterogeneous distributed network, tools such as the MPI and PVM frameworks supports developers to spawn processes, manage their communication, and message passing, and to solve synchronization issues during the development of distributed applications. These tools are described later at Section 13.13.

13.9 Common execution models

Fortunately, there are a range of problems which indicate the need for concurrent execution. In the next sections we describe some of these. These problems are used to illustrate possible synchronization issues and alternative solutions for designing concurrent algorithms. Our reason to describe them is twofold.

On one hand, while the problems themselves may seem trivial, inadequate implementations for their solution may easily result in problem of concurrency (involving deadlocks, livelocks and starvation). On the other hand, understanding these problems, and being familiar with the patterns that are needed for solving them is important for every developer preparing to write parallel application in their day-to-day work.

A selective set of solutions for these problems is described in detailed in the subsequent sections.

13.9.1 Producers-consumers problem

At the heart of the *producers-consumers problem*[16] lies the finite storage, *a buffer*, usually represented by an array, queue or similar data structure. Since this buffer is limited, the problem is also known as *bounded-buffer problem*. Two processes use this shared resource:

- a *producer*, which creates new elements during its activity and puts them into the buffer, and
- a *consumer*, whose role is to repeatedly take elements from the buffer and process them one after the other.

In its original form, there is only one of each processes but the problem can be generalized to N producers and M consumers. This is a commonly used model when the problem can be split easily into two separate tasks and the throughput must be finetuned by varying the number of producers/consumers (the rate of creating or removing elements may vary).

A good practical example for this model is a web crawler application: since the I/O communication through sockets may be slow and may get parallelized efficiently, we may have several *walker* processes that parse URLs and put the different downloaded HTML contents into the buffer as separate elements. Then a set of other processes may take an element (an HTML page) from the buffer, parse them and save the results into a database.

There are several caveats for which the implementation must be prepared for. Inadequate solutions might result in liveness hazards, performance issues and starvation. For example, when the buffer is full, producers must wait until a consumer takes an element out and frees up some space in the shared storage. Conversely, if all the elements are removed from the storage, consumers must

[16] `http://en.wikipedia.org/wiki/Producer-consumer_problem`

wait until one of the producers prepares a new data structure and puts it into the buffer for further processing. This problem can be solved through inter-process communication, typically by using semaphores.

13.9.2 Readers-writers problem

In the *readers-writers problem*,[17] there is a shared resource which may typically be represented by a database, used by multiple processes. A set of the processes is reading this resource only, while another set of the processes is modifying it too.

One might use several approaches to solve the problem. The most intuitive solution is probably to use mutual exclusion on the shared data. The problem is that this solution is suboptimal: readers should be allowed to use the shared resource concurrently because their work does not interfere. However, this extension must be done carefully since it may lead to serious starvation issues. Reader processes may monopolize the resource preventing writers to acquire it even once at least.

The key to the solution is to find a proper parametrization for the number of processes. This requires experimenting and creating benchmarks.

13.9.3 Dining philosophers problem

The *Dining philosophers problem*[18] was originally proposed in [Dij74] as a course exercise, but its current formalism was developed in [Hoa04].

The problem may be demonstrated as follows: there are five philosophers sitting around a table eating spaghetti. To eat spaghetti a philosopher needs two forks, but there is only one fork available for each two "neighbors".[19]

The philosophers spend a some of their time thinking about the universe, and when they get hungry, they pick up either of the forks on one their sides, or both. If they are successful in grabbing two forks, they can start eating the (otherwise infinite supply of) spaghetti. When they finish eating, they put down the forks and continue thinking. This process is repeated infinitely.

The original problem was used to describe how different computers represented by the philosophers may access shared peripherals like tape drives. Nowadays it can be generalized –philosophers represent processes, forks represent shared resources and spaghetti represents input, processed by the shared resources. The model could be considered as a minimized model of the interaction of operating system processes where for example hundreds of processes cooperate by hundreds of synchronization primitives like locks and semaphores.

The importance of the model is shown by its demonstrative power. For one incorrect implementation several synchronization issues may be encountered:

[17] `http://en.wikipedia.org/wiki/Readers-writers_problem`
[18] `http://en.wikipedia.org/wiki/Dining_philosophers_problem`
[19] Alternatively it may be the problem of rice and chopsticks. ;-)

- *deadlocks*: e.g. if all the philosophers may acquire the fork on their right, and the implementation does not allow the release of locked resources (*no preemption*), the system cannot progress forward.

- *starvation:* if one of the philosophers is interrupted to release its forks for the benefit of others, or the forks are assigned priorities, some of the processes might have advantage over the others. This may result in the starvation of specific processes (i.e. philosophers with lower priorities).

- *livelocks*: if any of the philosophers is interrupted to release their forks, unfortunate timing of the system may result in a livelock. For example, if all the philosophers grab the fork on their right at the same time, and then put it back on the table with the same timeout, none of the philosophers have two forks at a time. They grab the forks on their right again, and the process may repeat. The system is not "frozen" but cannot perform any useful activity.

There are several proven solutions for the problem. The original one, proposed by Dijkstra [Dij74], introduced a new component: *a waiter* whose responsibility is to give instructions to the philosophers when and how the philosophers to acquire the forks. Without his permission, the philosophers are unable to get any of the required resources – they first have to ask the waiter about their availability, i.e. the waiter acts as a semaphore.[20]

Another solution by Dijkstra [Dij71] proposes a hierarchical ordering for the forks (i.e. the resources). When philosophers cannot acquire a fork, they release their fork with the lowest priority, thus making the system deadlock-free. Unfortunately, this solution is impractical in some real-life situations (e.g. performing joins on database tables would be inefficient this way: if a table cannot be read at a certain time, previously locked tables would have to be released so they must be locked again).

A third notable solution by K. M. Chandy [CM84] suggests the use of *artificial agents* (i.e. no dirty forks), a solution, which handles the issue of starvation by assigning a dirty/clean state to the forks. This solution, however, violates one of the rules underlying the original problem – that is the philosophers must communicate with each other about the availability of the common resources. If the system starts with a "proper" initial state, this is also a deadlock-free solution.

In the next sections, we introduce the characteristics of a selection of programming languages from the the point of view of concurrency.

[20] As we have pointed out before, the concept of semaphore was also introduced by Dijkstra in 1965.

13.10 Ada

In the Ada programming language ([Nyek98], [Koz93], [Whe96] and [Bar96]) the language constructs for concurrent programming are *tasks* and *protected objects*.

13.10.1 Tasks

Tasks are program units executed concurrently by virtual processors in the same way as a simple, sequential program is executed, line by line.

Virtual processors are independent of each other and each of them is executing its own task concurrently. How the virtual processors associated to concrete processing units are implementation details handled by the language.

Tasks are working independently and are able to communicate at certain synchronization points. Tasks are cooperating in an asynchronous way by default. Synchronization of tasks are done in the form of *rendezvous* which is performed between a caller and a receiver task.

Tools for synchronization in Ada are the **entry**, **accept**, **delay**, **select** and **abort** instructions.

Tasks communicate through specified *entry points*. If a task is ready to send or receive a message from a certain task, it calls its corresponding entry point. In the receiver task each entry point is associated with at least, one **accept** instruction, and for the call the instructions embodied in the **accept** block are executed. Signature of an **entry** is similar to a method or function definition. It may also have parameters to solve the information flow between the caller and the receiver tasks.

In the following sections we concentrate on language elements used for communication and synchronization between tasks. Our goal is to illustrate the usage of these elementary building blocks through specific examples. These are the following elements: **task** is the most important language construct representing concurrency in the language; **entry** and the associated **accept** instruction is used to describe connection between different tasks; **select** whose complex semantics allows the declaration of different communication protocols; **delay** to resolve timing issues; and **abort** to stop the execution of a specified task.

Specification and body of a task

In the Ada programming language, a process is defined by a **task** unit which consists of the *specification* and the *body* of the task. The purpose of the specification and the body is to describe the activities performed by a given task. It is possible to declare *anonymous tasks* and create *unique instances* by omitting the **type** keyword. To create such a task, first it must be declared (13.24):

```
task Hello is
   entry message(s: String := "Hello world!");
end Hello;
```

Listing 13.24: Hello task in ADA

For the instructions to be executed, it must be defined as the task body (13.25):

```
task body Hello is
begin
   accept message(s: String := "Hello world!") do
      put_line(s);
   end message;
end Hello;
```

Listing 13.25: Hello task body

To define a new task type (not just a unique task instance) to implement a behavioral pattern, the specification must start with the **task type** keywords. This way it is possible to declare a new type of tasks that can be used to instantiate new objects based on this definition.

Values of a task type are tasks whose **entry** points are enumerated in the specification, and similarly to their bodies, they are defined after the **task body** keywords.

An example for the definition of a new task type:

```
task type Resource is
   entry Lock;
   entry Unlock;
end Resource;

task type Execution is
   entry Read(C: out Character);
   entry Write(C: is Character);
end Execution;

task type AnsweringMachine is
   entry IdentifyCaller(i: Integer);
   entry PickUp(s: Message);
end AnsweringMachine;
```

The body defines the execution of a task. The execution of a task body is triggered by the execution of a task, while the execution of a task is triggered after the evaluation of the declaration sections of the program, but strictly before starting the execution of the main body of the program (this scheduling is discussed in details later). A few examples for unique tasks and their bodies are shown in Listing 13.26.

```
task Producer;
task Consumer;

AnsweringMachine: array (1 .. 10) of AnsweringMachine;

task body Producer is
   C: Character;
begin
   loop
      Produce(C);
      Storage.Put(C);
   end loop
end Producer;

task body Consumer is
   C: Character;
begin
   loop
      Storage.Pop(C);
      Consume(C);
   end loop;
end Consumer;

task body AnsweringMachine is
begin
   accept IdentifyCaller(i: Integer) do
      put_line("Client " & Integer'Image(i) & ". called you.");
   end IdentifyCaller;
   accept PickUp(s: Message) do
      put_line(Message'Image(s));
   end PickUp;
end AnsweringMachine;
```

Listing 13.26: Solving the producers-consumers problem with ADA tasks

Task types, task objects

Each task is considered as a separate object, and thus they are called *task objects*. As all objects, tasks have types as well. Task objects having the same type do the same instruction sequences repeatedly, but independently of each other.

If the task object is an object (or a member of an object) that was introduced:

- in an object declaration, the actual value of the task object is defined on the basis of that object declaration.
- through the evaluation of a pointer (**new**), the concrete value of the task object is defined on the basis of the evaluation of that pointer.

The task type is considered as a **limited** type, which means that neither the assignment operator nor the comparison operators (=, /=) are applicable for them. This also means that their behavior is strictly restricted to the attributes of the task. Consequently, the value of task objects are constant, and they cannot be modified during runtime. Task types cannot be specified as **out** or **generic in** parameters of a method, but a function is allowed to return a task object by passing by reference. A task object is declared in the same way as any other object, as it is illustrated in Listing 13.27.

```
T1, T2: TaskType;

-- Create an array of tasks
type TaskArray is array(1 .. 10) of TaskType;

-- defining pointers is done by using access
-- about using access, see the section on using pointers
type TaskPointer is access TaskType;

-- Dynamically create a new task
TP: TaskPointer := new TaskType;
```

Listing 13.27: Defining the array of tasks

Listing 13.28 shows a few examples about defining different task variables.

```
type AnsweringMachinePointer is access AnsweringMachine;
AnsweringMachineArray: array (1 .. 10) of AnsweringMachine;
AnsweringMachinePointer: Pointer := new AnsweringMachine;
```

Listing 13.28: Task variables

Every task object owns separate instances of the variables defined in the task type. A new task object from a given task type is executing the instructions of its body concurrently with any other object. Besides their own variables, a task may also access any global variable defined outside of the task body which may be a shared variable.

A task may also contain a discriminant section which parametrizes the type (see Listing 13.29). The same rules apply for using a discriminant with a task as when using them for arrays or record types. If a task has a discriminant, it is evaluated at the instantiation of a new task object. During the lifetime of a task object the value of its discriminant cannot be changed.

```
task type Identifier(Discriminant: Type) is
   entry Declaration;
end Identifier;
```

Listing 13.29: Task with discriminant section

The discriminant is a parameter that has influence on the execution of a task object. While tasks of the same task type execute the same sequence of instructions repeatedly, their discriminant is allowed to contain different values. This execution of the tasks may also differ either at initialization (i.e. evaluating the initial value of the discriminant) or during communication.

As an example (shown in Listing 13.30, we consider a task type that writes data into an array. If the array is specified as a discriminant, the task objects may work on different array instances with different types:

```
task type ArrayWriter(ArrayName: access String) is
   entry Put(K: Character);
end ArrayWriter;
```

Listing 13.30: Definition of ArrayWriter

A task with a discriminant cannot be a uniquely instantiated task. Instantiation is done as the example in Listing 13.31 illustrates, where *CharArrayName* and *StringArrayName* are two *String* variables:

```
A: ArrayWriter(ArrayName => CharArrayName'Access);
B: ArrayWriter(ArrayName => StringArrayName'Access);
```

Listing 13.31: Definition of ArrayWriter

The body of a task may be replaced by the declaration shown in Listing 13.32.

```
task body Name is separate;
```

Listing 13.32: Definition of ArrayWriter

The declaration which denotes that the body of the task is in a separate source file. In this case, the source file containing the actual implementation of the task body must start with the **separate**(⟨*taskdeclaration*⟩) line.

Starting and executing tasks

Execution of a task is defined by its body. The initial step of execution is the starting of a task which includes the evaluation of its declarations.

Starting a task is done right after processing the declarations of the object that introduced it, but before executing the first statement of the defining entity.

If the declaration is in a **package** specification, starting takes place after processing the declarations. Task objects, declared either separately or as a member of an object, and assigned by pointers, are instantiated when the pointer is evaluated, after the initialization of the non-task typed components of the enclosing object is finished.

If an error occurs when a task has started, the state of that task becomes *completed*, and a *Tasking_Error* is reported whose location is pointing right before the first instruction of the program unit containing the definition of the task or at the line that contains the pointer referring to the task. A task in completed state cannot be restarted again.

Termination of tasks

A given task is connected to several other program units. All tasks are depending on at least one parent, which may be another task, a code block under execution, a subroutine or a package. This dependency is direct in the following two cases:

- The task (object), code block, subprogram or package which contains the definition of a given task is considered as the parent of the task;

- If the task was created by the evaluation of a pointer (allocator), the task (object), code block, subprogram or package declaration of the concrete **access** type is considered as the parent of the task.

If a task depends on a block or subprogram, it also depends on the task executing the block, subprogram or package indirectly. If *T1* task depends on *T2* task, then *T1* is a descendant of *T2*. This is a transitive relation: if *T1* task depends on *T2* task, and *T2* depends on *T3*, then *T1* depends on *T3* as well.

A task becomes completed when it meets any of the following conditions:

- All its instructions in its body are executed correctly;
- When there was an error but the task did not had any corresponding error handling;
- An error was raised and its handling was completed successfully and there was no additional instructions after the error handling.

The **abort** yields to abnormal interruption of a task that stops its execution. A task stopped this way cannot participate in any additional rendezvous which may result in allocated but unreleased resources. A task stopped by the **abort** statement is an *abnormal task* that becomes completed when it reaches the next synchronization point (i.e. entry call, starting of a new task, finishing its activity, any of the **select**, **abort**, **delay** statements, reaching either the beginning or end of a corresponding exception handler or an **accept** block). A task is permitted to stop itself (suicidal tasks).

A task *terminates*, if one of the following conditions exists:

- It is completed, and all of the tasks which depend on it are terminated;
- Its execution has reached a **terminate** statement, and the task depends on a program unit that is completed (i.e. its execution is ended,[21]) and all the tasks which depend on the parent are also terminated, completed or waiting at a **terminate** statement.
- It got into an abnormal state by an **abort** call, has become completed and has been removed from all waiting queues.

Task attributes

All tasks provide the following attributes:

TaskName'Callable: a boolean value which is *false* if the task has reached one of the completed, terminated or aborted state; *true* otherwise (i.e. it is able to answer to any call).

TaskName'Terminated: a boolean value which is *true* if the task is terminated; *false* otherwise.

EntryName'Count: returns the number of awaiting **accept** calls for the specified entry point.

[21] The sole exception from this rule are packages.

13.10.2 Entry, entry calls, accept statement

A task may define any number of entry points. A task communicates with another task by calling its corresponding entry point (similar to function calls).

As we have mentioned before, the synchronization and data sharing between tasks is done by **entry** calls in the first place. The declaration of an **entry** is similar to the declaration of any subprogram, but it is allowed strictly within a task definition. An entry point may have parameters like any subprograms, with the same rules applied for the formal and actual parameters. An important difference is, however, that an entry point is not allowed to declare an **access** parameter. The specification of the entry point defines the direction of communication upon entering and the possible messages (entries may have **in**, **out** or **in out** parameters).

As an example, the *Hello* task in the example in the introduction of this section has a *message* entry point that may be called by *Hello.message(str)*, where *str* is the actual parameter of the entry point.

Each **entry** is associated with at least one **accept** statement (**entry** and **accept** coexist symmetrically) defining the sequence of instructions to perform on the given **entry** call.

Using entry points tasks are able to directly communicate with each other. If a task calls the entry point of another task which acknowledges the call, a *rendezvous* is performed between the two tasks. If two tasks would like to communicate in the form of a rendezvous, the first task reaching the rendezvous point is suspended and waits for its partner.

When a task performs an entry call to another task which has not reached the corresponding **accept** statement, its execution is suspended. By contrast, if a task has reached an **accept** statement and there are no other tasks waiting at a corresponding **entry** call, it is suspended as well.

If a task calls an entry point of another task, it waits until the other task accepts its call, i.e. its execution must reach a corresponding **accept** statement for the given entry point. Thus, a task calling its own entry point results in a deadlock.

At an **entry** call when the called task reaches a corresponding **accept** statement, the caller is suspended, and the body of the **accept** statement is executed. This is how a *rendezvous* takes place between two tasks. After the processing of the body of the **accept** statement, both tasks continue their work separately as they have done before.

If there are multiple tasks calling on the same entry point while the receiver task have not reached the corresponding **accept** statement, the calls are put into a waiting queue. All **entry** is associated with a separate queue, where the tasks calling the same **entry** are waiting in the order their call has arrived (it is a first-in-first-out queue).

Each execution of the corresponding **accept** statement extracts the first element of the queue. There are two ways how a suspended task could be removed

from the waiting queue of an entry point: the receiver has reached a corresponding **accept** statement, thus it extracts the task from the queue that was waiting for the longest time for the communication (*successful rendezvous*); or the timeout was reached for the given call, or the caller was abnormally terminated, aborted (*missed rendezvous*). The number of tasks waiting at a given entry point at a given instant of time is stored in its *E'Count* attribute.

The *Storage* task (shown in Listing 13.33) is a simple example for the usage of the **accept** statement. It can store a single element that is accessed both by the *Producer* and by the *Consumer* tasks.

```
task body Storage is
   T: Character;
begin
   loop
      accept Put(C: in Character) do
         T := C;
      end Put;
      accept Pop(C: out Character) do
         C := T;
      end Pop;
   end loop;
end Storage;
```

Listing 13.33: Definition of Storage body

The **accept** statement is a statement inside the body of a task. For a single entry point multiple **accept** statements may be defined. An incoming call is then served by the first executed **accept** statement of the receiver.

The parameters of an **accept** statement are identical to the parameters of the corresponding **entry** statement. Under a rendezvous, the tasks may communicate through the actual parameters. If a parameter of an **accept** is declared as an **in** parameter, its value is copied into the variable represented by the formal parameter; if it is declared as an **in out** parameter, the result is copied back to the variable represented by the actual parameter. Until the execution of the **accept** block is finished, the execution of the caller task is suspended.

All the instructions that a task may execute outside of a rendezvous should be put outside of an **accept** to let the caller task continue its work without unnecessary waiting.

The **accept** statement is also usable to synchronize the execution of tasks. A simple **accept** statement without parameters that has an empty body consisting only from the **null** statement may enforce the caller to continue its work strictly after the other task has reached a certain execution point or state. The **accept**

statement of task T must be defined in the body of task T - it cannot be declared in any of its subprograms, packages or tasks that is encapsulated within T.

It is important to note the differences between a method/function call and a rendezvous. In the case of a method/function call, the body of the subprogram becomes part of the caller. Local variables of the method/function are represented by corresponding temporal variables within the caller. The same subprogram may be executed by multiple processes, so different instances of the same code block may be active at the same time (*reentrancy*). During a rendezvous, two active processes are joined. The interaction is done in a symmetric and synchronous way, when both of the participants are ready to perform it. Mutual exclusion is automatically managed for the objects defined in the declaration of a task.

Entry families

An **entry** declaration may also introduce a group of related entry points called an *entry family*. An entry family is an array of entry points. They are declared by a single entry point, and just like arrays, the entry points are indexed by a discrete type. Outside of the body, the **entry** names must be referred by a qualified name, where the prefix is the name of the task object. Another advantage of using entry families is that the number of entry points is easily increased with the size of the associated waiting queues. In the case of a single waiting queue the entry calls must be accepted by using a first-in-first-out scheduling, but in the case of multiple waiting queues alternative approaches might be implemented. The example in Listing 13.34 demonstrates the declaration of a simple entry family:

```
type Discrete_Type is Integer range 1 .. 5;
entry Family (Discrete_Type) (Parameters);
```

Listing 13.34: Definition of an entry family

Calling the *Family* entry point is done by the following way, where *Expression* must be a member of *Discrete_Type*, as it is illustrated in Listing 13.35.

```
Family(Expression);
```

Listing 13.35: Calling the entry point

The **select** *statement*

A rendezvous makes the communication between two tasks possible, but its usage might grow tedious. The reason is that a task must wait until someone does not accept its call, and vice versa; if a task receives a message, it must wait until someone contacts it with a message.

The `select` statement used together with the `accept`, `delay` and `terminate` statements lifts the burden. It allows a task to wait and accept multiple messages, thus makes managing rendezvous easier. It is also possible to define conditions (*guards*) for the acceptance of certain messages. The `select` statement also offers a convenient way for a task to terminate itself or to continue its own execution if there were no incoming messages for a while (*timeout*).

Consequently a whole set of different rendezvous can be implemented in the language with the combination of the **select**, **accept**, **delay** and **terminate** statements. Examples include implementing selective activation of tasks, selective waiting, conditional **entry** statements and timed entry calls.

13.10.3 Selective handling of incoming messages

Let us suppose we need to print words and numbers into the console. In order to prevent appearing the characters of the words and digits of the numbers randomly on the screen, we have to implement mutual exclusion for the task performing the printing (see Listing 13.36).

```
task Buffer is
   entry PrintNumber(I: in Integer);
   entry PrintString(K: in String);
end Buffer;
```

Listing 13.36: Selective handling of messages

Using *selective wait* is easily done by putting an additional **or** statement between the defined **accept** statements inside a **select** (13.37).

A task may contain multiple **accept** branches inside a **select** statement, and these alternatives are separated by the **or** keyword, allowing a task to listen for multiple entry calls at the same time.

In the example above, when the task reaches the **select** statement, it waits for any of the embodied entries to be called. If any other task calls one of the corresponding entry points, the corresponding **accept** alternative is executed. If

```
task body Buffer is
begin
   loop
      select
         accept PrintNumber(I: in Integer) do
            put(Integer'Image(I));
         end PrintNumber;
      or
         accept PrintString(K: in String) do
            put(K);
         end PrintString;
      end select;
      new_line;
   end loop;
end Buffer;
```

Listing 13.37: Selective handling of messages

the entry call was performed before the receiver reached the **select** statement, the caller is suspended until the receiver is ready to accept the message.

If more than one entry points has a non-empty waiting queue within the same **select** statement, the executed **accept** alternative is chosen in a non-deterministic way.

Selective waiting

By using selective waiting the execution of different **select** alternatives may be associated with preconditions called *guards*. If one of the branches of the **select** statement starts with a **when** *condition* => ... definition, its execution is guarded by the specified condition. This means that it can accept a message only if the given condition is satisfied.

A branch is considered *open* if it has no associated guard (`when`), or the specified condition is satisfied. Otherwise the branch is considered *closed*.

Guard conditions are evaluated *only once* when the execution encounters the **select** statement. A task may accept messages only for its open branches; execution of closed branches is prohibited. Re-evaluation of the guard conditions requires another execution of the corresponding **select** statement which can be achieved by, for example, a delayed execution of the **select** statement in a loop. It is important to note, however, that if there are no open alternatives, and a selective wait is executed, a *Program_Error* is raised.

The **terminate** alternative

In most cases activities of a task are defined by a selective wait within a loop. This means that if there are no `terminate` alternatives specified, the task never ends – it never passes the execution to any other program unit. Similarly, the unit containing the definition of the task object also runs forever. The `terminate` alternative of the `select` statement ensures the termination of the task execution. The `terminate` alternative may also be defined with a guard in the form of `when` ⟨*condition*⟩ `=> terminate`.

A task defined this way executes the **terminate** alternative when it has accepted all the incoming entry calls defined by its specification (i.e. all of its **entry** waiting queues are empty). Listing 13.38 shows an example.

```
select
   accept AcceptMessage(X: in Message) do
      -- ...
   end A1;
or
   accept SendMessage(Y: out Message) do
      -- ...
   end A2;
or
   -- The task terminates if there are no incoming messages to process
   terminate;
end select;
```

Listing 13.38: Termination alternative

The **delay** and the **abort** statements

The `delay t` statement suspends the execution of the task *for at least* `t` seconds. In Ada, the main program is also defined as a task. This means the `delay` statement may be executed outside a task definition which results in the suspension of the main program. To specify time, Ada provides a real `Duration` type and the `Calendar` package to handle dates. Using a `delay` statement within a selective wait makes it possible to delay the execution by the specified amount of time in the task before progressing further (or trying to execute guarded alternatives again in a `select` within a `loop`). The specified delay is evaluated from the execution of the `select` statement, and the `delay` alternative may also be associated with a guard condition.

The **abort** *T* statements puts the task into the previously described *abnormal* state. While in this state, a task may neither accept nor initiate a rendezvous.

The aborted task becomes completed when it reaches the next synchronization statement as discussed before.

*The **else** alternative*

Another possible alternative for selective wait is the `else` statement. The `select` statement executes the instructions of the `else` alternative when none of the previous entry calls could have been served at the given instant of time (i.e. there are incoming calls for the alternatives but they cannot accept the call temporarily because of their guard preconditions, and/or because there are no incoming calls for the open alternatives).

The selective wait must contain at least one branch that contains an **accept** statement, and in addition, it may have a branch that contains any of the **delay**, **terminate** or **else** alternatives.

Entering a **select** statement defines the open alternatives by evaluating the preconditions. In the next stage, one of the open alternatives (or if there are only closed ones, the **else** alternative) is executed. Then, the selective wait is considered as executed. If there are multiple open alternatives, the selection is done based on the following rules:

- If there is an open **accept** alternative where a rendezvous may be initiated, that one is chosen;
- If there is no open **accept** alternative, an open **delay** alternative is chosen;
- The **else** alternative is chosen strictly if all the other alternatives are closed;
- The **terminate** alternative cannot be chosen until a selective wait has a non-empty entry waiting queue.

Following is an example code for selective waiting:

```
task body Resource is
   Locked: Boolean := False;
begin
   loop
      select
         when not Locked => accept Lock do
            Locked := True;
         end Lock;
      or
         accept Unlock do
            Locked := False;
         end Unlock;
      or
         terminate;
      end select;
   end loop;
end Resource;
```

Distinct **accept** statements may be nested, and the body of an **accept** statement may just as well contain an entry call (*nested rendezvous*).

By contrast, additional rendezvous for the same entry point that is being executed cannot be initiated (i.e. **accept** statements for the same entry point cannot be nested into each other). The main reason is that a long delay may precede the incoming message for the nested rendezvous which implicitly delays the execution of the enclosing rendezvous (*hidden waiting*). It is a good practice to avoid nested rendezvous, and if unavoidable, make them guarded.

Conditional entry call

In the previous sections we have described how a task can answer different entry calls selectively, but it is also possible to call the entry points of a task based on certain conditions.

In order to prevent overtly long waiting for the acceptance of a submitted entry call, a *conditional* or a *timed entry call* is the right solution. However, it is best not to use conditional entry points inside loops (cf. *busy waiting*) for performance reasons. Calling multiple entry points is not supported, but polling can be applied instead.

As it is demonstrated in Listing 13.39, a conditional entry call is a **select** statement with an **entry** and an **else** alternative. The `entry` call is processed if the rendezvous may be initiated instantly, otherwise the `else` alternative is executed (this may have several reasons, e.g. the execution of the counterpart has not reached the required `accept` statement yet, the `accept` is a closed alternative, or the required entry point has a non-empty waiting queue).

```
procedure TryLocking(R: in Resource) is
begin
   loop
      select R.Lock;
      else null; -- Waiting for resource release
      end select;
   end loop;
end TryLock;
```

Listing 13.39: Example for conditional entry call

Timed entry call

A timed entry call is a `select` statement with two alternatives: the actual entry call and a `delay` statement. The entry call is processed if the rendezvous can be initiated instantly. Otherwise the delay alternative is executed postponing the required (but currently unavailable) operation that cannot be done at the given instant of time, see the example below for its usage.

```
select
   Server.SetUser(user, password);
   put_line("Login completed.")
or
   delay 4.0;
   put_line("Server was busy, trying again.");
end select;
```

13.10.4 Exception handling

Task-related exceptions reported based on the following rules:

- When there is an error during the initialization of a task, a *Tasking_Error* is reported in its parent;
- When an entry call was made on an abnormal, terminated or completed task, a *Tasking_Error* is reported in the caller;
- An exception raised within a rendezvous that was not handled is reported both in the caller and the receiver tasks.

13.10.5 Examples

Mutual exclusion

In the introductory section of this chapter we have seen that mutual exclusion is a vital tool in concurrent programming to ensure exclusive access to a shared

resource for a single process. In the Ada programming language this can be implemented in several straightforward ways.

In the first example (13.40) the shared resource is defined within the task, where *T* is a previously defined type, and *Shared* is the name of the shared object.

```
task SharedVariable is
   entry Write(X: in T);
   entry Read(X: out T);
end SharedVariable;

task body SharedVariable is
   Shared: T;
begin
   loop
      select
         accept Write(X: in T) do
            Shared := X;
         end Write;
      or accept Read(X: out T) do
            X := Shared;
         end Read;
      or terminate;
      end select;
   end loop;
end SharedVariable;
```

Listing 13.40: Sharing resources within tasks

The shared resource may well be a global variable. In this case mutual exclusion must be implemented explicitly by using a semaphore. Next we provide a simple implementation for the conventional *P* and *V* methods, where only the

first caller task may enter the protected code (*binary semaphore*).

```
task type Semaphore is
   entry P;
   entry V;
end Semaphore;

task body Semaphore is
begin
   loop
      accept P;
      accept V;
   end loop;
end Semaphore;
```

Using a shared resource properly requires locking it first (i.e. calling the P method on the associated semaphore), and releasing it after the task has finished processing it (i.e. calling the V method on the associated semaphore), as shown in Listing 13.41. This way mutual exclusion is provided for the common resource.

```
   Shared: T;
   S: Semaphore;
begin
   ...
   S.P;
   Shared := F(Shared); -- Updating the variable
   S.V;
   ...
end
```

Listing 13.41: Providing mutual exclusion by a semaphore

The semaphore used for global variables (or client-side locking in general) is not a safe synchronization method. If there is a single place in the codebase where any of the semaphore methods have been omitted by accident when accessing the shared resource, the program starts producing nondeterministic bugs.

Producers-consumers example

Let us take a look on a naive implementation of the producers-consumers example in ADA as shown in Listing 13.42 (for its description, see Section 13.9.1).

```
with Shared_Queue;
package Int_Shared_Queue is new Shared_Queue(Integer, 128);

task body Producer is
   C: Integer;
begin
   loop
      -- Producing the new C element...
      Int_Shared_Queue.Queue.Put(C);
   end loop;
end Producer;

task body Consumer is
   C: Integer;
begin
   loop
      Int_Shared_Queue.Queue.Pop(C);
      -- Consuming the extracted C element...
   end loop;
end Consumer;
```

Listing 13.42: Solving the producers-consumers problem

The produced but not processed data is stored in a queue that is accessible for both the producer and the consumer tasks with mutual exclusion. Handling such a queue is demonstrated in Listing 13.43.

Protected objects

Protected objects [Nyek98] contain data that is accessible in a task through a set of predefined *protected methods*. They offer a convenient solution without the need for explicit mutual exclusion. The protected members must be defined within the **private** section of the specification.

Three kind of subprograms can be exported from a protected module - namely, protected procedures, protected functions and protected entries. A protected entry is similar to a guarded entry point of a simple task. If the specified precondition is true, the caller task executes the body of the entry point. If it is false, the caller is put into a waiting queue for the given entry call until the specified precondition is satisfied just like in the case of tasks. The protected procedure has read and write access to the variables defined in the **private** section of the specification. When a task calls a protected procedure, no other task is allowed to access the private variables, i.e. there is no need for explicit mutual

```
generic
   type Elem is private;
   Queue_Length: Natural;
package Shared_Queue is
task Queue is
   entry Put (D: out Elem);
   entry Pop (D: in Elem);
end Queue;
end Shared_Queue;

package body Shared_Queue is
task body Queue is
   type P is new Natural range 0 .. Queue_Length;
   subtype Actual_Storage is P range 1 .. P'Last;
   Storage: array (Actual_Storage) of Elem;
   Counter: P := 0;
   Inx, Outx: Actual_Storage := 1;
begin
   loop
      select
         when Counter > 0 =>
            accept Pop (D: out Elem) do
               D := Storage(Outx);
            end Pop;
            Outx := Outx mod P'Last + 1;
            Counter := Counter - 1;
         or
         when Counter < P'Last =>
            accept Put (D: in Elem) do
               Storage(Inx) := D;
            end Put;
            Inx := Inx mod P'Last + 1;
            Counter := Counter + 1;
         or terminate;
      end select;
   end loop;
end Queue;
end Shared_Queue;
```

Listing 13.43: Solving the producers-consumers problem by using a shared queue

exclusion on the shared data. Protected functions have only read access for the private variables, therefore a protected functions may be executed by multiple tasks at the same time.

A protected entry point differs from a protected procedure in two ways: it has an additional waiting queue and it must have a guard condition.

Protected objects provide a convenient and easy way to solve several problems and to build up the required synchronization architectures in an inherently concurrent environment. The example in Listing 13.44 demonstrates how a binary semaphore could be implemented by a protected object.

```
protected type Resource is
   entry Lock;
   procedure Unlock;
private
   locked: Boolean := False;
end Resource;

protected body Resource is
   entry Lock when not locked is
      begin
         locked := True;
      end Lock;
   procedure Unlock is
      begin
         locked := False;
      end Unlock;
end Resource;
```

Listing 13.44: Example for protecting objects

The usage of semaphores is shown in Listing 13.45.

```
Printer: Resource;
begin
   Printer.Lock;
   Print;
   Printer.Unlock;
end program;
```

Listing 13.45: Example for protecting objects with semaphores

A protected unit may also define an *entry family*. The example in Listing 13.46 demonstrates how to use entry families to handle prioritized messages,

where the *Console* always prints the most important message to the standard output.

```
type Priority is (High, Medium, Low);
protected Console is
   entry Message (Priority) (Text: in String);
   function CanPrint (Actual: Priority) return Boolean;
end Console;

protected body Console is
   entry Message (for Actual in Priority) (Text: in String)
      when CanPrint(Actual) is
   begin
      Put_Line(Priority'Image(Actual) & Ascii.HT & Text);
   end Message;

   function CanPrint (Actual: Priority) return Boolean is
      HigherPrior: Priority := Priority'Last;
   begin
      while HigherPrior > Actual loop
         if Message(HigherPrior)'Count > 0 then
            return False;
         end if;
         HigherPrior := Priority'Pred(HigherPrior);
      end loop;
      return True;
   end CanPrint;
end Console;
```

Listing 13.46: Example for handle prioritized units

Tasks using the *Console* object defined above can send messages to the standard output in the following way, where each member of the entry family has a separate waiting queue:

```
Console.Message(High)("Minima maxima sunt");
```

It is possible to create *protected objects* and *protected types* in the same manner as how task objects and task types are declared.. A protected object can be instantiated statically (through a variable definition) or dynamically (through an allocator) from a protected type. The sole difference between the definition of a protected type and a protected object syntactically is the presence of **type** after the **protected** keyword.

Protected type declarations might also have discriminants for which the same rules apply as for records: the discriminant may either be a discrete type or a pointer definition. The example in Listing 13.47 shows another semaphore definition with a discriminant this time:

```
protected type Semaphore ( Max: Positive := 1 ) is
   entry P;
   procedure V;
private
   Nr: Natural := 0;
end Semaphore;

protected body Semaphore is
   entry P when Nr < Max is
   begin
      Nr := Nr + 1;
   end P;

   procedure V is
   begin
      Nr := Nr - 1;
   end V;
end Semaphore;
```

Listing 13.47: Defining semaphores with discriminants

13.11 CSP

The CSP (Communicating Sequential Processes) is a language described by C. A. R. Hoare in 1978 [Hoa78]. An updated version of the language is described in [Hoa85]. It is important to note that originally CSP was invented as a formal notation: an academic approach to describe, simulate, and most importantly, to allow reasoning (i.e. the notation permits to formally provide the deadlock-freeness) about concurrent systems, but without many practical implementation features. There is considerable literature on the concept and the features of the language, but implementations have been rare. Today, there are several CSP-like implementations, frameworks and domain specific languages, like C++CSP for C++, CSP.NET for the .NET platform, JCSP for Java, PyCSP for Python and Agent for Ruby.

In addition the language is important as it has had a great influence on numerous upcoming languages and approaches like Occam (which was based

on CSP), Ada, Google's Go and Linda-systems. Moreover, before 2006, Hoare's book was the third most cited computer reference book of all times, states the Citeseer[22] database. Thus, the language definitely deserves some attention.

CSP considers the input/output statements as elementary language constructs (like the assignment operator for instance). To handle communication between concurrent processes, input/output operations have a short and elegant notation. The input operations are done by the `?` operator. The following example:

```
p2 ? msg()
```

means that process executing an input operation is waiting the arrival of the `msg()` message from the `p2` process. The pair of the input operator is the output operator which is denoted by the `!` symbol. The usage is similar:

```
p1 ! msg()
```

which means that the process executing the output operation is sending the `msg()` message to the `p1` process through an abstract channel. Communication is done in the form of rendezvous between the `p1` and `p2` processes, where the latter process declares the former as the receiver of the message (output operation), and the former declares the latter as a possible sender of the message (input operation).

The communication is synchronous and the process executing the first operation must wait for its counterpart. Until the proper operation is executed on the other side, the process is blocked. The communication is done by *pattern matching*. The sent and received messages match and are acceptable only if the names of their constructors are identical respectively, i.e. they are from the same type. In the previous two examples the same compound variable has been present having an empty expression list, whose constructor was the `msg` identifier.

The input/output operation is unsuccessful if the specified sender or receiver process is already terminated.

The additional operators and language constructs in CSP are the following ones. To handle nondeterminism, *prefixing* is an important operation that consists of two parts, an event (*channel guards*) and a sequence of statements. The event definition may contain both a boolean condition and an input statement. The specified sequence of statements can be executed only if the given precondition is true, or if an input operator is specified, and the sender process has not terminated yet, and it is ready to send the required message. Output operations are not allowed in event definitions. In the following example:

```
p ? msg1() ⟶ p ? msg2()
```

when the `p` process receives a `msg1` message, then it receives a `msg2` message afterward. Subsequently, the acceptance of the first `msg1` message is the precondition for the acceptance of the second `msg2` message.

[22] `http://citeseerx.ist.psu.edu`

The *interleaving operator* (`| |`) is used to describe simultaneous execution of a sequence of statements. These statements are arbitrarily interleaved in time. The processes are started at the same time, and the interleaved statement ends when all of its parallel components are terminated. The processes are independent of each other and are prohibited to use shared variables (i.e. none of them is allowed to use a variable reference that appears at the left side of an assignment operator in any other process).

The **choice** statement (`[]`) consists of a finite number of event definition, like (α_1 `->` `p1` `[]` α_2 `->` `p2`). It describes an event on whose execution any one of its components whose precondition is satisfied may be executed non-deterministically. If none of the preconditions are satisfied, the operation does not have any effect. If all of the components whose precondition is true have an input operation that specifies a process that is not ready to send its input (and is not yet terminated), the execution of the choice statement is suspended temporarily.

The *iterative* statement (`*`) consists of choices. The enclosed choice statements are executed sequentially after each other until the defined guard condition is true, and the statement is terminated when it becomes false.

The statement sequences may also be addressed with labels. If the label is defined with an interval, like:

```
p(i:0..4)::P
```

then a sequence of `P` processes starts executing the same set of instructions, but in this time substituting the `i` variable with the corresponding element of the specified interval.

Example: Dining philosophers problem

As an example, let us consider a CSP solution for the dining philosophers problem. This particular implementation was made by Dan Richardson on the basis of Hoare's CSP book [Hoa04].

```
PHIL = *[ ...During n'th lifetime
  THINK;
  room!enter();
  fork(i)!pickup();
  fork((i+1) mod 5)!pickup;
  EAT;
  fork(i)!putdown();
  fork((i+1) mod 5)!putdown();
  room!exit();
]

FORK = *[ phil(i)?pickup() -> phil(i)?putdown();
          [] phil((i-1) mod 5)?pickup() -> phil((i-1)mod5)?putdown();
        ]
```

```
ROOM = occupancy:integer; occupancy = 0;
  *[ (i:0..4)phil(i)?enter() -> occupancy := occupancy + 1;
     []
     (i:0..4)phil(i)?exit() -> occupancy := occupancy - 1; ]

MAIN = [room::ROOM || fork(i:0..4)::FORK || phil(i:0..4)::PHIL].
```

The implementation details deserve a closer look. In the definition of the `ROOM` process the precondition to enter the room is the value of the `occupancy` variable that must be lower than four. This way the case when all the philosophers enter the room simultaneously is avoided. This is important because in the case when the room is full and all the philosophers get the fork on their left for example at the same time could lead to a deadlock – a case when processes are infinitely waiting for each other. Such situations are ruled out this way for this particular implementation.

13.12 Occam

The Occam [Bar92] programming language was specifically designed to support the development of concurrent applications. An application constructed by this particular language can be imagined as a collection of concurrently executed processes.

As it has been emphasized several times in this chapter, in the world of concurrent programming one of the most important issues is how communication is implemented between the processes. In Occam, communication is done through unidirectional channels (*simplex communication*) without buffering. The type of the channel (i.e. the type of message structures sent through the channel) is specified by the channel protocol at the declaration of the channel. These properties come from the strictly typed aspect of the language. Communication is implemented by the means of two basic operations. One of them is the output operation which is used to write a value on one side of the channel. The output operation waits until a proper input operation reads and removes the written value from the channel. The general form of the operation is shown in Listing 13.48.

```
channel ! expression
```

Listing 13.48: Writing a value on a channel

The other important operation is the input operation which reads a value from the channel and puts it into a variable with a proper type. Just like the

output operation, the input operation also waits until it is able to read the required message from the channel. Listing 13.49 shows the general form of the operation.

```
channel ? variable
```

Listing 13.49: Reading from a channel

Let us now examine which types are used for message passing and for the declaration of channels. The available primitive types include `BOOL`, `BYTE`, and integer and real values represented on 16, 32 or 64 bits named by `INT16/32/64` and `REAL16/32/64`, respectively. As an example, declaration of an `INT32` variable is done in Occam as the code in Listing 13.50 demonstrates.

```
INT32 x:
```

Listing 13.50: Declaring a variable in Occam

Accordingly, the declaration of a simple channel whose identifier is `inchannel` used to communicate 32 bit integer values looks as follows (see Listing 13.51).

```
CHAN OF INT32 inchannel:
```

Listing 13.51: Declaration of a channel for 32 bit integers

In the most recent version of the Occam language (Occam 2.1 was defined in 1994, while Occam-π was introduced in 2006) it is possible to define complex data structures, like literals, array and record constructs with the possibility of *channel retyping*. This new language element makes it possible to define a channel for instance that can be used to send a string whose length is defined only at the call site. The following example defines a simple channel used to transfer an integer value (the length of an array) and a series of elements:

```
CHAN OF INT::[]BYTE inchannel:
```

Listing 13.52: Defining a channel for transferring an integer

The output operation may look like the following, for instance:

```
inchannel ! 4 :: "Long Message"
```

Listing 13.53: Sending a message through the channel

while the input operation may receive the message as follows:

```
inchannel ? length::array
```

Listing 13.54: Receiving the message

In the examples above, variable `length` receives the value of 4 while the `array` variable receives the `''Long''` message (the first 4 elements of the corresponding input).

After discussing the basic concepts of communication, next we will show how simultaneously running processes may be defined. The first step is to take a look on how two processes are bound together. The simplest composition is done by the `SEQ` keyword which ensures the sequential execution of the specified instructions:

```
SEQ
   screen ! "Input character: "
   keyboard ? char
   screen ! char
   screen ! cr
   screen ! lf
```

The example above reads a character from the standard input and echoes it back to the screen followed by a carrige return and a newline character.

A more interesting building block is the composition of statements by the `PAR` keyword which results in the concurrent execution of the specified instructions:

```
WHILE next <> eof
  SEQ
    x := next
    PAR
      in ? next
      out ! x*x
```

The short example above first assigns the value of the `next` variable to `x`. Then it concurrently updates the value of the `next` variable from the `in` channel while writing the value of `x*x` into the `out` channel at the same time. These statements are repeated until the value of `next` does not contain the end of file character.

It is important to note that in parallel execution, a variable whose value is updated in any of the processes (either through an assignment operator or in an input operation) cannot be referred to in any other concurrent process. This rule makes *the following example erroneous*:

```
PAR                 -- This composition is erroneous!
   error := 42      -- The error variable is used...
   ch ? error       -- ... for multiple concurrent processes.
```

Similarly, a channel that is used during an input operation cannot be used in an input operation within another process. The same stands for channel variables used in output operations. *Another demonstrative erroneous example* is the following:

```
PAR                 -- This composition is erroneous!
   ch ! 0           -- The ch channel is used in an output operation...
   SEQ
      ch ? x
      ch ? y
   ch ! 1           -- ... within multiple concurrent processes.
```

When it comes to concurrent execution, it is common to execute the same sequence of instructions repeatedly. The `PAR` construction (along with `SEQ`, `ALT` and `IF`) has a special syntax for this common case embedding a `FOR` statement called *replicated processes*. The left and right hand side variants of the following example are identical.

```
PAR i = 1 FOR 4              PAR
    user[i] ! message            user[1] ! message
                                 user[2] ! message
                                 user[3] ! message
                                 user[4] ! message
```

The `ALT` control flow statement is another interesting construction, similar to the `choice` statement of the CSP programming language. This language element is used commonly in situations, when for example there are multiple input channels, and the result of the output channel is depends on which input channel was the source of the data.

```
ALT
  ⟨boolean condition⟩ & in1 ? next
        out ! x*x
  in2 ? next
```

```
        out ! x+5
```

As the example above illustrates, a boolean condition may be associated to the guard precondition, just like in the CSP programming language.

When the alternatives are identical, the `FOR` statement may be used with the `ALT` composition as well, similarly to the `PAR` keyword. The left and right handside variants of the following example are identical.

```
ALT i=1 FOR 3          ALT
  in[i] ? next           in[1] ? next
    out ! next             out ! next
                         in[2] ? next
                           out ! next
                         in[3] ? next
                           out ! next
```

13.13 MPI

MPI (Message Passing Interface) [MPI94] is a standard specification for defining communication based on messages between different applications, algorithms or subprograms. It is implemented by various programming languages such as C [KR89] or Fortran 77 [LV77] to support low-level parallelism. In this chapter, we deal with the implementation in C, therefore the code samples are written according to C syntax. The main purpose of MPI is to solve inter-thread point-to-point communication, to handle tasks in groups and to organize them in a graph of Cartesian topology. As most of the MPI functions operate on variables, not on defined types, we have to specify their type as another argument. All of the standard basic types such as `int` or `char` have their proper identifiers, for instance `int` type is specified by `MPI_INT`, `char` type is done by `MPI_CHAR`. Complex, or self-defined data types must be converted to MPI derived data types by `MPI_Type_create_struct` function.

As MPI 1 standard does not define methods for starting tasks, in its concept a task means a stand-alone applications which are going to be executed in parallel. Specification of MPI 2 solves many restrictions of the previous version. It allows to create tasks after starting the application, and thus it extends the concept of a task by allowing also subprograms to be executed as a different task. Another great improvement of the MPI 2 standard is that it provides an opportunity for communicating tasks which are not in any relation to the graph topology (in MPI 1 the communication was restricted to parent-child-related tasks).

MPI control methods

To be able to use MPI functions, at first we have to initialize the MPI library itself by calling the `MPI_init` function, and since MPI deals with command

line arguments, we must specify them as parameters. In the case of normal termination the allocated resources are freed by using `MPI_Finalize` method. Otherwise, in the case of an error, a task or a group of tasks can be interrupted by the `MPI_Abort` method.

Creating tasks

There are two methods for creating MPI tasks. Both allocate a given `maxprocs` number of processors and execute the applications with the specified `argv` parameters, then create communication possibilities among them, and finally return an external communication environment (`MPI_COMM_WORLD`). The only difference between these methods that is while `MPI_Comm_spawn` can be used to start one program in several instances, `MPI_Comm_spawn_multiple` starts different applications, or the same application but with different input parameters. Consider the following example of starting two applications.

```
char *array_of_commands[2] = {"prog1", "prog2"};
char **matrix_of_argv[2];
char *argv0[] = {"-gridfile", "ifile.grd", (char *)0};
char *argv1[] = {"ifile2.grd", (char *)0};
matrix_of_argv[0] = argv0;
matrix_of_argv[1] = argv1;
MPI_Comm_spawn_multiple(2, array_of_commands, matrix_of_argv, . . .);
```

Groups

To avoid code repetition and to ensure more flexible code optimization, processes can be organized into groups. By definition groups are ordered set of tasks, in where all tasks can be identified within the group by an index starting from 0. It can be done by connecting a pair of a group identifier and a sequential number to the tasks, which refer to the task unequivocally. Group ID, which can be referenced to as a variable with *MPI_GROUP* type can only be used inside of the tasks and cannot be given to another task. There are two other types in relation to the groups: *MPI_GROUP_EMPTY* denotes the empty group, while *MPI_GROUP_NULL* denotes the non-existing group. Although one pair of a group ID and a sequential number identifies a task, a task can have membership in several groups, so it can have multiple pairs associated with it. All of the set operations can be performed on groups obtaining new ones. For example we can have the intersection (*MPI_Group_intersection*), union (*MPI_Group_union*), subtraction (*MPI_Group_difference*) of two groups; we can also add (*MPI_Group_incl* method), or remove (*MPI_Group_excl*) a task to/from the group.

Groups can be compared (*MPI_Group_compare*), their size (i.e. how many processes are in the group) can be retrieved by the *MPI_Group_size* method, or we

can get the sequential number of a given task in the group (*MPI_Group_rank*). Groups can be freed by *MPI_Group_free* method.

The concept of communication environments is motivated by the need for separating the functional code of a task from its communication with the outside world. With this concept we can define different message communication levels that enable us to create and handle message channels separately from each other for more tasks. A communication environment can be set for a group only. *MPI_Comm_group* method returns a group associated with a given communication environment.

Communication methods

Basic communication methods implement unidirectional, ordered point-to-point communication. Besides that, communications can be classified in accordance with the usage of temporary storage (yes or no), synchronization (synchronous or asynchronous), by behavior (blocking or non-blocking) and the level of communication (single tasks or groups). The concept of synchronization in MPI terminology means that if task A sends a message to task B with synchronization, task A must wait until B starts a receiver method, but not until the end of it, therefore it cannot guarantee the completeness of messaging. Correct synchronized mode can be achieved if we apply blocking communication methods both for sending and for receiving a message. The most simple method called *MPI_Send* implements a blocking, asynchronous communication without using temporary storage. It is capable of sending a vector of data with a given type. Other methods which provide communication in different ways have the same signature that is, they are different in their names only. *MPI_Bsend* method implements data transfer using a temporary storage, which means the successful transfer does not follow necessarily from the end of the method since the data itself can be stored temporarily.

Synchronous communication can be done by *MPI_Ssend* method. But, as we have mentioned before, it does not mean real synchronization since the method will not be blocked until the message receiver method ends; it waits until the receiver **starts** to work.

MPI_Rsend implements the so-called expected communication, where its calling can be successful in one way: if and only if the receiver has already executed an *MPI_Recv* method, so mainly it is waiting for the message. Otherwise it returns with undefined result.

Non-blocking communication methods can be separated to two parts: one which initializes the communication while the other is for checking its end. It is a practical solution if the communication itself takes a long time, and instead of waiting, the sender and receiver tasks can process other instructions.

MPI_Isend method is responsible for sending messages in non-blocking way. It differs from its analogous blocking method in its last *MPI_Request* parameter. The method initializes the given *MPI_Request* structure to store the properties

of messaging running in the background which enables the programmers to check the current state of the transfer, to collect information about its successfulness and to check whether an error has occurred. After initialization the method returns, but does not terminate further processes (real messaging, updating the status structure) that are being executed in the background concurrently with the execution of the program's other part. Modifying the message to be sent during the transfer is inexpedient since the system reads it non-determinately.

Non-blocking messaging has additional versions, e.g. *MPI_Ibsend* method works with temporary storage, while *MPI_Issend* implements synchronized communication, and *MPI_Irsend* implements expected communication.

For non-blocking receiving, we can use *MPI_Irecv* method, which differs from *MPI_Recv* in its last *MPI_Request* parameter only.

Non-blocking messaging requires the ability to check the status of the running methods. MPI provides more opportunities for this: *MPI_Wait* method blocks the execution until the given method finishes, *MPI_Test* works without blocking the execution and checks if the messaging has been finished.

MPI contains methods for receiving data sent by any sending methods. For instance, the *MPI_Recv* method is capable of getting messages sent by using blocking communication methods. Although the type and the source of the message may be specified, it may be specified using *MPI_ANY_SOURCE*, or *MPI_ANY_TAG* parameters to accept messages from any tasks or with any type. Status information about the received messages can be retrieved from *status* parameter which has predefined fields to store the source task id (*MPI_SOURCE*), the type of the message (*MPI_TAG*), and the success-fulness of the reception (*MPI_ERROR*).

Communication in groups

In addition to point-to-point communication, MPI provides possibilities for communication across the groups (hereafter: group communication), which means that more tasks contribute to the communication and all of them must call the methods at the same time. Methods for group communication implement simple algorithms that may be achieved by using methods introduced above, but these new ones have simpler signature, are completely separated from other communication methods, and are executed faster than the composition of the original point-to-point methods.

Among others, group communication methods implement functionalities of synchronization, collect data or broadcast messages. *MPI_Barrier* can be used to synchronize multiple tasks, collect data from tasks or broadcast messages to all tasks. The method is blocked until all of the tasks using the given communication environment does not call the same barrier method.

Collecting data from a set of tasks is one of the main objects in the case of distributed applications. *MPI_Gather* provides this functionality. All tasks send

a certain amount of data collected in a vector. The place of the data item is identified by the index of the sender task.

With the *MPI_Scatter* method we can distribute a vector of data among tasks in a given communication environment by index. For example, data in ith place of the vector are going to be sent to the task with i.

MPI_Alltoall can be used to broadcast different data from all tasks to all others in the same communication environment. Technically, it is identical to the parallel execution of the *MPI_Gather* and *MPI_Scatter* methods.

In addition these basic methods described in detailed before, MPI 2 specification provides other functionalities (e.g. MPI 2 allows communication between tasks which does not have common communication environment; it allows communication by remote memory access).

A simple MPI example is shown as a next example. This application starts n tasks and all of them send its unique id and the number of the tasks to task 0.

```
#include <stdio.h>
#include <mpi.h>
main(int argc, char *argv[]) {
    const int tag = 42;
    int id, task_number, code_id, collector_id, error, i;
    MPI_Status status;
    int msg[2];
    error = MPI_Init(&argc, &argv);
    if (error != MPI_SUCCESS) {
        printf("MPI initialization error!\n");
        exit(1);
    }
    error = MPI_Comm_size(MPI_COMM_WORLD, &task_number);
    error = MPI_Comm_rank(MPI_COMM_WORLD, &id);
    if (task_number < 2) {
        printf("At least 2 processors required for this program\n");
        MPI_Finalize();
        exit(0);
    }
    if (id == 0) {
        for (i = 1; i < task_number; i++) {
            error = MPI_Recv(msg, 2, MPI_INT, MPI_ANY_SOURCE,
                             tag, MPI_COMM_WORLD, &status);
            code_id = status.MPI_SOURCE;
            printf("Received message %d %d from task %d
                    \n", msg[0], msg[1], code_id);
        }
    } else {
        msg[0] = id;
        msg[1] = task_number;
        collector_id = 0;
        error = MPI_Send(msg, 2, MPI_INT, collector_id,
                         tag, MPI_COMM_WORLD);
    }
    error = MPI_Finalize();
    if (id == 0) printf("End.\n");
    exit(0);
}
```

13.13.1 Case study: Matrix multiplication

In order to demonstrate language constructs which support concurrency, we consider a case study of matrix multiplication. We would like to multiply three matrices called A, B and C. For simplicity their dimensions are $3 * 3$. Since multiplication on matrices is an *associative operation*, theoretically, this problem can be solved in two steps. First, we count multiplication of A and B, and then we multiply the result matrix and C to get multiplication of all the three matrices.

In all the examples below we denote matrix A by $matrixA$, B by $matrixB$ and C by $matrixC$, and their values in this case are the following:

$$matrixA := \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

$$matrixB := \begin{pmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 1 \end{pmatrix}$$

$$matrixC := \begin{pmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 1 & 2 \end{pmatrix}$$

13.14 Java

There are three mayor ways to implement codes with paralellization in Java - namely, by implementing *Runnable* interface, extending *Thread* class or by implementing a *Callable* interface which is similar to *Runnable*, but can return a value and can throw exceptions.

The Runnable interface and the thread class

Since Java does not support multiple inheritance, when implementing interfaces we are still able to extend our class from another one. Another advantage of using *Runnable* interface is that they can be restarted anytime independently of each other implementation. A simple implementation of *Runnable* class is shown in the next code snippet:

```
public class SimpleRunnable implements Runnable {
    public void run() {
        System.out.println("New runnable started.");
    }
}
```

Separating and executing the example above can be done by defining the class to extend *Thread*:

```
public class SimpleThread extends Thread {
    public void run() {
        System.out.println("New thread started.");
    }
}
```

Similar to *Runnable* interface and in case of main methods, the core algorithm must be implemented in a *run()* method in classes extending *Thread* as well. After starting a thread, this method will be executed once, which means that the thread class itself must implement a way of long-time execution. For instance it can be done by starting infinite loops, and within it by creating an exit point by an interrupted condition. Among other functionalities, *Thread* class offers a possibility for controlling these issues by high-level methods detailed in the next table:

Method name	Return value	Description
currentThread()	Thread	Returns a reference to the thread object currently being executed.
getState()	Thread.State	Returns the state of this thread.
interrupt()	void	Interrupts this thread.
isAlive()	boolean	Tests if this thread is alive.
isDaemon()	boolean	Tests if this thread is a daemon thread.
isInterrupted()	boolean	Tests whether this thread has been interrupted.
join()	void	Waits for this thread to terminate.
setDaemon(boolean on)	void	Marks this thread as either a daemon thread or a user thread.
sleep(long millis)	void	Causes thread to sleep (temporarily cease execution) for the specified number of milliseconds.
start()	void	Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
yield()	void	Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Table 13.1: Most important methods of the Thread class

As it can be seen in the next code example, starting threads, or *Runnable* objects are not very different, since *Runnable* object can be wrapped within *Thread* by specifying it as a constructor parameter. However, there is huge a difference among their process, since creating a thread means creating a defined object with separated memory allocation in line with the terms of object-oriented programming theory. Importantly, if we create a thread using Runnable imple-

mentations multiple times, the runnable object will be shared among the different thread objects.

```
SimpleThread thread = new SimpleThread();
thread.start();

SimpleRunnable runnable = new SimpleRunnable();
Thread runnableThread = new Thread(runnable);
runnableThread.start();
```

It is important to note, however, that there is another possibility to create tasks by creating objects of *Callable* class. It differs from the *Runnable* class in only one respect - that is, its objects can return a value, while objects of the *Runnable* class cannot do it explicitly.

Creation of *Thread* objects is a relatively time-consuming process, therefore, however if we would like to use it for simple tasks (e.g. to exploit multiple cores without possibility of interference) implementing Runnable interface is suggested. On the other hand, if functionalities could be implemented conveniently by functionalities of Thread objects, we need to take care about their optimal management. As we can see on the next pages, Java supports these thread management concepts in many ways.

Now let us have a look at an example that implements the Matrix multiplication problem mentioned above to illustrate the usage of threads from different aspects. This example will be modified incrementally as we introduce new features of parallelization in Java.

```
public static int[][] Multiply(int matrixA[][],int matrixB[][]) {
   int matrixR[][] = new int[3][3];
   for (int i=0;i<matrixA.length;++i) {
      for (int j=0;j<matrixB.length;++j) {
         for (int k=0;k<matrixB[j].length;++k) {
            matrixR[i][k] += matrixA[i][j]*matrixB[j][k];
         }
      }
   }
   return matrixR;
} . . .
int matrixAB[][] = Multiply(matrixA,matrixB);
int matrixABC[][] = Multiply(matrixAB,matrixC);
```

The execution can be made faster on a large scale. That is, we need to create and dedicate as many threads as the number of rows matrix A, and the main program will collect all the results of the rows. When all the rows are returned from the threads, the same method will be executed, but on the result matrix and matrix C. In this sense comparing with the original application, the running time is decreased by creating a new thread. Let us now create a thread class called *RowMultiplier*:

```
public class RowMultiplier extends Thread {
    private int RowA[];
    private int matrixB[][];
    private int result[] = new int[3];

    public int[] getResult() {
        return result;
    }
    public void run() {
        for (int j=0;j<matrixB.length;++j) {
              for (int k=0;k<matrixB[j].length;++k) {
                 result[k] += RowA[j]*matrixB[j][k];
              }
         }
    }
    public RowMultiplier(int RowA[],int matrixB[][]) {
        this.RowA = RowA;
        this.matrixB = matrixB;
    }
}
```

As it is shown in the next Listing within the constructor we store the arguments, and then start executing the thread. It executes run method to operate on the stored row and the matrix, and saves the result of the multiplication in result array, which can be get by *getResult()* method. In the main method, we iterate *matrixA* by its rows, and in each iteration we create *RowMultiplier* object and read the results.

```
public static int[][] MupltiplyByThread(int matrixA[][],int matrixB[][])
                    throws InterruptedException{
   int result[][] = new int[3][3];
   for (int i = 0; i<matrixA.length;++i){
      RowMultiplier counterThread=new RowMultiplier(matrixA[i],matrixB);
      result[i] = counterThread.start().getResult();
   }
   return result;
}
...
matrixAB = MupltiplyByThread(matrixA,matrixB);
matrixABC = MupltiplyByThread(matrixAB,matrixC);
```

This solution looks good on a small scale, but once we start increasing the size of the matrices to be multiplied, the algorithm stops working. It stops, because the threads need more and more time to do the multiplication of the row and the matrix, which also means that they cannot produce the counted array right after starting them (by the *getResult()* method). As shown in the next Listing, what we can do instead is to start all the threads, then, in a next loop, use the *join()* method of the threads, which blocks the caller (the main) thread until the

called one is terminated. As the threads have already been started, this function does not block the parallel execution, and does not delay any other threads to start: it only guarantees that the results will be queried in order of the rows, and block the loop within an iteration until the current thread does not terminate.

```
    for (int i=0; i<matrixA.length;++i){
        threads[i] = new RowMultiplier(matrixA[i],matrixB);
        threads[i].start();
    }
    for (int i=0; i<matrixA.length;++i){
        try {
            threads[i].join();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
    for (int i=0; i<matrixA.length;++i){
        matrixAB[i] = threads[i].getResult();
    }
    return matrixAB;
}
...
matrixAB = MultiplyByThreadSafely(matrixA,matrixB);
matrixABC = MultiplyByThreadSafely(matrixAB,matrixC);
```

Thread groups

In this book, the focus is on applicable solutions for parallelism, and, as Brian Goetz says, *"The ThreadGroup class was originally intended to be useful in structuring collections of threads into groups. However, it turns out that ThreadGroup is not all that useful. You are better off simply using the equivalent methods in Thread. ThreadGroup does offer one useful feature not (yet) present in Thread: the uncaughtException() method. When a thread within a thread group exits because it threw an uncaught exception, the ThreadGroup.uncaughtException() method is called. This gives you an opportunity to shut down the system, write a message to a log file, or restart a failed service."* [GPB06]. Suffice it to say here that Java provides solutions named *ThreadGroup*s, but we do not go into detail in explaining its features.

The concurrent API

Java collects high-level constructs for concurrency in *java.util.concurrent* package including several tools for avoiding deadlocks, tools to support lock idiom, and means to organize threads in groups or pool structures.

The following codes are based on Java online documentation [Mic03]. Primitive types such as integers from Java 5.0 have analogous implementation named

Atomic, which ensures memory consistency by its specification. Therefore, developers need not deal with, they get this bond by default. Technically these classes implement manipulation on basic volatile variables, which can be achieved by using synchronized method technique. However, Atomic variables also protect from unnecessary synchronization.

Concurrent collections

Concurrent package extends the Java Collection Framework with analogous, but thread-safe implementations of common collections. In the earlier versions of Java 5.0, coarse-grained safety is implemented against possible troubles of concurrent usage. The collections within the concurrent package are designed specifically for multi-threaded usage. Built in mechanisms have been extended with new concepts, the *Map* interface as *ConcurrentMap*, the *Queue* interface as *BlockingQueue* and *ConcurrentMap* has a new *ConcurrentNavigableMap* subinterface. They ensure mutual exclusion on a set of collection items, and this reduce the need for synchronization and the performance overhead under heavy workload. In general, to ensure memory consistency, writer methods in concurrent collections invoked by a thread must be executed before any reader method coming from any other thread.

ConcurrentMap (which is implemented by *ConcurrentHashMap*) operates on key-value pairs, and defines atomic operations guaranteeing the consistency of the data structure. Actually, manipulation methods (*remove()* and *replace()*) are executed if and only if the key exists, and *putIfAbsent()* method applies if the key is not found in the collection.

BlockingQueue defines a FIFO queue which blocks if we retrieve from an empty queue, or add further elements to a full queue. Next we show its implementations.

- *DelayQueue* An unbounded blocking FIFO queue of elements on which a specific getting method is defined: an item can be retrieved if and only if a given time after its insertion is expired. If there is no available item after the specified delay, the retrieving method will return **null**.
- *LinkedBlockingDeque* An optionally-bounded blocking deque based on linked nodes.
- *LinkedBlockingQueue* An optionally-bounded blocking FIFO queue based on linked nodes.
- *LinkedTransferQueue* An unbounded specific FIFO queue called *TransferQueue* based on linked nodes. *TransferQueue* is a modified *BlockingQueue* in which the invoked threads'put method can wait for other threads for getting that element.
- *PriorityBlockingQueue* An unbounded blocking queue that uses the same ordering rules as class *PriorityQueue* and supplies blocking retrieval operations.

- *SynchronousQueue* It is a structure which may contain only one element, implemented as a blocking queue with specific semantics : each insert operation must wait for a corresponding remove operation by another thread and vice versa. As it can be seen, it is a really specific implementation of a queue: it cannot be iterated, because there is nothing to iterate, as it can store only one element. Insertion of a non-empty SynchronousQueue cannot be applied. If there is no head element in the queue, then no element is available for removal for the *poll()* methods, so they will return **null**.

  Synchronous queues are similar to rendezvous channels used in CSP and Ada. They are well suited for handoff designs, in which an object running in one thread must sync up with an object running in another thread in order to hand some information, an event or a task to it.

- *ConcurrentNavigableMap* is inherited from the *ConcurrentMap* interface, and modified with a useful and practical meaning. Instead of the original sematics of get method in maps (which returns the value of the given key), *ConcurrentNavigableMap* supports approximate matches. The general and scalable implementation is *ConcurrentSkipListMap*.

  The map is sorted according to the natural ordering of its keys, or by a *Comparator* provided at map creation time, depending on which constructor is used.

  This class implements a concurrent variant of *SkipLists* providing expected average $log(n)$ time cost for the *containsKey()*, *get()*, *put()* and *remove()* operations and their variants. Insertion, removal, update and access operations safely execute concurrently by multiple threads. Iterators are weakly consistent, returning elements reflecting the state of the map at some point or since the creation of the iterator. They do not throw *ConcurrentModificationException*, and may proceed concurrently with other operations. Ascending key ordered views and their iterators are faster than descending ones.

The Executor framework

To start and to manage threads at a high-level, the concurrent package contains an API called the *Executor Framework* which can be used to create threads in a more sophisticated way separated from the core application itself, and to organize threads for computation-demanding applications. It contains a feature called *Executor Interface* to create threads easier than implementing a *Runnable* class for each one. Concept of *Thread Pools* is to manage several threads with a generic solution. The Fork/Join Framework, introduced in Java 7, is to utilize multi-core or cluster systems in the application by ensuring a generic way for cloning threads, which will be executed on a different core, and for collecting threads.

Executor Interfaces

The executor interface defines a simplified interface to create and start threads, as it can be seen in the next example Listing. Instead of creating a new object of our thread that extends the *Runnable* class and then starting it, we just add it to an executor object's execute method as a parameter.

```
(new Thread(r)).start();

e.execute(r);
```

Although Java provides these possibilities to create threads, these methods are not completely equivalent. While we may be sure that *start()* method starts the thread immediately, *execute()* can make an existing worker thread to run, or drop the *Runnable* object to a queue of a worker thread.

Since the *Executor* interface provides an elementary functionality, it has been extended by additional subinterfaces: *ExecutorService* adds features for thread or executor management, and its subinterface *ScheduledExecutorService* supports future and/or periodic execution of tasks.

The ExecutorService interface

Executor interface deals with only *Runnable* objects, which can be a heavy constraint if, for instance, we use *Callable* objects, as executors cannot be used to handle them. Therefore, to resolve this issue, we may turn to ExecutorService interface (a subinterface of Executor interface) which replaces the execute method with a different submit method that allows the *Callable* object to be handled. It returns a *Future* object to be able to manage the status of the task and to get the *Callable* return value.

ExecutorService also provides methods for submitting large collections of *Callable* objects. Finally, *ExecutorService* provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks must handle interrupts correctly.

There is a need not only to handle a large set of *Callable* objects, but also to execute them periodically. This is why *ScheduledExecutorService* interface was defined and added to the collection package. This interface states that threads must be executed after a given time in line, with a given scheduling or time and time again repeatedly.

Among others, the *ThreadPoolExecutor* and *ScheduledThreadPoolExecutor* classes implement the *ExecutorService* and the *ScheduledExecutorService* interfaces, respectively.

Thread pools

Although threads are suitable to exploit multi-core systems, starting and shutting them down are time-consuming operations, and if, for instance our ap-

plication requires an increasing number of threads to process (let us imagine a web-server that creates a handler thread for each request coming from the users), it makes the program run out of memory. These are serious problems, which occur in a system that deals with large sets of data, or (in case of servers) has a constraint of acceptable response times. Therefore, thread pools have been specified to solve these issues. Thread pools consist of worker threads which exist independently from the *Runnable* and *Callable* tasks they execute and they are often used to execute multiple tasks.

One common type of thread pools is the fixed-size thread pool. This type of pools has a specified number of worker threads running; if a thread is terminated, it is automatically replaced with a new thread.

The *java.util.concurrent.Executors* class offers several factory methods to create executor objects which use fixed-size thread pools as their execution structure. One of them is the *newFixedThreadPool()* method which simply creates a thread pool capable to handle the specified number of threads. The *newCachedThreadPool()* method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks. The *newSingleThreadExecutor()* method creates an executor that executes a single task at a time. There are several alternative factory methods to create *ScheduledExecutorService* versions of the above executors.

Our next example implements the Matrix multiplication problem using thread pools as a conclusion. At first, we create a *BlockingQueue* to store *Runnable* objects (which will be used worker threads), then we create a *ThreadPoolExecutor* object called executor and set the *threadQueue* as its storage structure. Finally, we create and start a monitor thread to check the status of the worker threads (*ThreadMonitor* is a self-developed class, as it will be shown soon).

```
public static int[][] MultiplyByThreadPool(
        int matrixA[][], int matrixB[][],int matrixC[][]) {
    int matrixABC[][] = new int[3][3];
    BlockingQueue<Runnable> threadQueue =
          new ArrayBlockingQueue<Runnable>(6);
    ThreadPoolExecutor executor = new ThreadPoolExecutor(
          3, 3, 10, TimeUnit.SECONDS, threadQueue);
    ThreadMonitor monitor = new ThreadMonitor(executor);
    monitor.start();
}
```

Now we have to have a list to store the manager objects of the multiplication of A and B matrices, so a list with *Future* objects are initialized to store an array of integers. Then, we iterate over each row of matrix A and submit a new thread to the executor (return values must be stored in *listAB*).

```
ArrayList<Future<int[]>> listAB = new ArrayList<Future<int[]>>();
for (int i=0; i<matrixA.length; ++i){
   RowMultiplier rowMul = new RowMultiplier(matrixA[i], matrixB);
   Future<int[]> submit = executor.submit(rowMul);
   listAB.add(submit);
}
```

Then we create a new list of manager objects to store objects of multiplication of AB and C. We can iterate through the manager objects, and submit a new multiplication thread to the executor. Finally we add its *Future* object to the $listABC$ list.

```
ArrayList<Future<int[]>> listABC = new ArrayList<Future<int[]>>();
for (Future<int[]> result : listAB) {
   try {
      int[] resulti = result.get();
      RowMultiplier rowMul = new RowMultiplier(resulti, matrixC);
      Future<int[]> submit = executor.submit(rowMul);

      listABC.add(submit);
   } catch (Exception ex) {
      ex.printStackTrace();
   }
}
```

Finally we collect all the results to the multidimensional array $matrixABC$ and shut down the executor and its monitor.

```
for (int i=0; i<listABC.size(); ++i) {
   try {
      int[] resulti = listABC.get(i).get();
      matrixABC[i] = resulti;
   } catch (Exception ex) {
      ex.printStackTrace();
   }
   executor.shutdown();
   monitor.stopSignal();
   return matrixABC;
}
```

As a first step in our example, we have claimed that we will create a monitor object which is able to check the status of the worker threads covered by the pool. The following code block shows its implementation.

```
import java.util.concurrent.ThreadPoolExecutor;
public class ThreadMonitor extends Thread {
    private boolean stopSignal = false;
    private ThreadPoolExecutor threadPool;
    public ThreadMonitor(ThreadPoolExecutor threadPool) {
        this.threadPool = threadPool;
    }
    public void stopSignal() { stopSignal = true; }
    @Override
    public void run() {
        do {
            int active = this.threadPool.getActiveCount();
            long completed = this.threadPool.getCompletedTaskCount();
            long task = this.threadPool.getTaskCount();
            System.out.println("Thread statistics: " +
                "Active/Task/Completed: "+active+"/"+task+"/"+completed);
        } while(!stopSignal);
    }
}
```

Fork/Join Parallelism

In Java 7, the *ExecutorService* interface was implemented by a feature called Fork/Join Framework to create applications executed typically in multi-core environment. The concept follows a *split-as-many-as-you-want* idiom, which means that the main problem (e.g. operations on large, but separatable data) can be split into smaller pieces until it can be executed by one worker thread. As it implements *ExecutorService*, the Fork/Join Framework uses worker threads in a thread pool. The advantage of the Fork/Join Framework is that it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy. For purposes of illustration, consider the following example.

Expecting for matrix multiplication itself can be parallelized, let us extend the problem to show the benefits of the Fork/Join Framework. We would like to multiply n matrices $(A_1, A_2, ..., A_n)$ instead of multiplying just three of them. How should it be done? As the operation of multiplication has the property of associativity, we can multiply the matrices one by one, $A_1 * A_2$ then $A_{1,2} * A_3$ etc. It does not seem to be a good solution, though, as it can hardly be parallelized - the ith operation requires the result of the $(i-1)$th. A better choice is to create pairs from the matrices, $(A_1, A_2), (A_3, A_4), ..., (A_{n-1}, A_n)$, do the multiplication against the pairs resulting $\lceil n/2 \rceil$ multiplied matrices. Then, as in a generic case multiplication is not a commutative operation, by keeping the order of the matrices, we can create $(A_{12}, A_{34})(A_{56}, A_{78})...(A_{(n-3)(n-2)}, A_{(n-1)n}$ pairs, do the multiplication against the pairs again, and repeat the process iteratively until we have only one $A_{1,2,...,n}$ matrix, the result of the multiplication.

This process shows how the Fork/Join Framework can be used – we must create a specific class extended by *RecursiveAction* class that implements the basic operation which can be executed, in our case the multiplication of two matrices (in the code example this method is called *ComputeDirectly*). This class must implement the abstract method called *compute()* with no parameters. A commonly used solution is to implement *compute()* as a recursive method. In our case, if the given set of the matrices contains more than two matrices, we may split the matrices to two vectors and invoke a new execution against them concurrently. Otherwise we can do the multiplication. But how could we retrieve the evaluated matrix? We need smart indexing of the result vector which should be initialized with maximal number of rows and the columns in the matrices. Therefore we can be sure that it can be used to load the result matrix there.

The process is the following. Each *MMultiplier* object must be constructed with an index called *binaryIndex*. It is stored as a simple string and it must contain only 0 or 1 characters. When the two new *MMultiplier* objects are generated, their index will be generated using their parent index concatenating 0 in one case, and 1 in the other. All iterations start with index 0. By using this process, all pairs have a unique index identifier that contains only 0 or 1, which we convert to decimal numbers as normal indices. The indices will be generated according to the order of the matrices. One issue that must be emphasized that this process results an array of n matrices with $\lceil n/2 \rceil$ filled places. Therefore before the next iteration we must erase the **null** matrices, and the *MMultiplier* class must extend the*RecursiveAction* class. Furthermore, its *compute()* method must be implemented to be a recursive method.

```
public class MMultiplier extends RecursiveAction {
    private Vector<int[][]> matrices;
    private int[][][] results;
    private String index;
    public String getIndex() { return index; }
    public int[][][] getResults() { return results; }
    public void setResults(int[][][] results) { this.results = results; }
    public MMultiplier(Vector<int[][]> matrices,String binaryIndex,
            int[][][] result){
        this.matrices = matrices;
        index = binaryIndex;
        this.results = result;
    }
    protected void computeDirectly() {
        // matrix A := matrices.get(0)
        // matrix B := matrices.get(1)
        int[][] result = new int[3][3];
        if (matrices.size() == 1){
            result = matrices.get(0);
        } else{
          for (int i=0;i<matrices.get(0).length;++i) {
            for (int j=0;j<matrices.get(1).length;++j) {
              for (int k=0;k<matrices.get(1)[j].length;++k) {
                result[i][k] += matrices.get(0)[i][j]*matrices.get(1)[j][k];
              }
            }
          }
        }
        results[(int)Integer.parseInt(index,2)] = result;
    }
```

```
@Override
protected void compute() {
    if (matrices.size() < 3){
        computeDirectly();
        return;
    } else {
        int split = matrices.size() / 2;
        Vector<int[][]> matrices0 = new Vector<int[][]>();
        Vector<int[][]> matrices1 = new Vector<int[][]>();
        for (int i=0;i<split;++i){
            matrices0.add(matrices.get(i));
        }
        for (int i=split;i<matrices.size();++i){
            matrices1.add(matrices.get(i));
        }
        String currIndex = this.getIndex();
        invokeAll(new MMultiplier(matrices0,currIndex+"0",results),
              new MMultiplier(matrices1,currIndex+"1",results));
    }
}
```

We create an *MMultiplier* object with the input matrices, create a new *ForkJoinPool* object to handle the process, then call its *invoke()* method for *MMultiplier* object. As its result variable has a blocking getter method, right after it we can retrieve the vector of the evaluated matrices by the *getResult()* method. Then we set up the new vector of matrices (**null** matrices must be erased), and until the *actSize* is variable (which stores the number of valid matrices) greater than one, we execute these steps iteratively.

```
public static void MultiplyByForkJoinGeneral() {
    Vector<int[][]> matrices = new Vector<int[][]>();

    matrices.add(matrixA);
    matrices.add(matrixB);
    matrices.add(matrixC);
    matrices.add(matrixD);
    matrices.add(matrixE);
    int actSize = 5;
    int[][][] r = new int[actSize][3][3] ;

    while (actSize>1) {
        int[][][] results = new int[actSize][3][3];
        MMultiplier m = new MMultiplier(matrices,"0",results);
        ForkJoinPool forkjoin = new ForkJoinPool();
        forkjoin.invoke(m);
        r = m.getResults();
        double doubleSize = actSize;
        matrices.removeAllElements();
```

```
            for (int i=0; i<actSize; ++i){
                if (!isMNull(results[i])){
                    matrices.add(results[i]);
                }
            }
            actSize = (int) Math.ceil(doubleSize/2);
        }
        System.out.println("Final Multiplication");
        for (int i=0;i<r.length;++i) {
            System.out.println("---" + i + "-----");
            printMatrix(r[i]);
        }
    }
    private static boolean isMNull(int[][] matrix) {
        for (int i=0; i<matrix.length; ++i) {
            for (int j=0; j<matrix[i].length; ++j) {
                if(matrix[i][j] != 0) return false;
            }
        }
        return true;
    }
```

13.15 C#/.NET

Since beside Java, C# is the other main programming language used world-wide, we avoid describing its features for concurrent programming. For the same reasons we show the concurrency-related options and concepts in reference to Java too. As part of the .Net Framework, C# is a powerful object-oriented language with a wide-range of programming techniques coming with .Net, represented by code libraries, such as database manager libraries or libraries responsible for autogenerating ASP web-pages. Concurrency is supported at two levels.

Since its beginnings .Net has offered several techniques for concurrent programming (Threads or tasks), which have then been improved by, for instance, Thread Pools. However, before the release of .Net 4.0 in 2011, there was no dedicated library for supporting concurrency, which includes specific and prefabricated thread-safe data structures. Our description is based on .Net 4.0. First of all, let us compare .Net with Java.

13.15.1 Comparison of .Net with Java

Similarities

As .Net was specified and developed by taking Java concepts in count, the two languages are fundamentally very similar. Both languages provide techniques for implementing mutual exclusion; Java has **synchronized** keyword for it, C#/.Net

names it *lock*, but the semantics of the two are the same. One other similarity is in the field of signaling. In Java, the Object class has methods to block the execution process (*wait*), and to send a signal to one or all the waiting threads (*notify* and *notifyAll*, respectively); in C# we have analogous methods defined in the Monitor class, but they are named *Wait*, *Pulse*, *PulseAll* respectively..

Differences

In Java a part of the algorithm must be implemented in consideration of its multithreaded nature, namely, the classes that implement *Runnable* or *Callable* interface, or extended from the *Thread* class, may be handled with multithread techniques. In C# there are more options since, as it is shown by the following code sample, any method can be used as core code of a thread:

```
Thread t = new Thread( () => FunctionToBeExecutedParallel() );
```

The same is true for lock-free variables. Java represents lock-free version of common variable types such as Integer or Boolean by prefabricated classes in *java.util.concurrent.atomic* package. C# follows a different path - it provides a general manner (System.Threading.Interlocked) that can be used to define lock-free behavior on *any* object type. The following code snippet shows how we may increment a locked variable through its reference using interlocked:

```
System.Threading.Interlocked.Increment(ref locked);
```

Specialties

Task Parallel Library was introduced in .Net 4.0 as a set of commonly used libraries for implementing multithread applications. It offers solutions for both data and task parallelism. An interesting feature in this set is the revision of the common iteration structures in C#. Namely, developers are allowed to execute all the iterations of a loop in parallel by using *Parallel.For* or *Parallel.ForEach* keyword. As the following code snippet demonstrates, the *doSomething* method will be executed six times in parallel:

```
Parallel.For (0, 5,i =>
                        {
                        doSomething(i);
                        });
```

As a great improvement of .Net 4.0, the *System.Collections.Concurrent* class introduces several and various lock-protected thread-safe data structures: beside the thread-safe version of the common structures (*ConcurrentQueue*, *ConcurrentDictionary*), *BlockingCollection<T>* gives a simple thread-safe version for collections that implement *IProducerConsumerCollection<T>* interface. While *ConcurrentBag<T>* is a simple, but thread-safe set for unordered objects, *ConcurrentStack<T>* is an alternative version of *ConcurrentQueue* representing

a LIFO queue instead of a FIFO. The *OrderablePartitioner<TSource>* is more interesting: it can be applied on an ordered data source, and can split them into multiple parts which can then be modified in parallel. *Partitioner<TSource>* which does the same, but without the restriction of orderliness.

In the next case study of matrix multiplication we give an example of many of these functionalities and techniques from the above mentioned *ThreadPools* until the Interlocked objects.

Case study: Matrix multiplication

Creating class for thread parameters:

```
public class ThreadParameters
{
        public int threadId{ get; set; }
        public int[] rowA{ get; set; }
        public int[][] matrixB{ get; set; }
}
```

Declaration of variables:

```
class MainClass{
  // These static variables are shared among Threads!
        static int[][] matrixA = new int[3][] {
                                 new int[] {1,2,3},
                                 new int[] {4,5,6},
                                 new int[] {7,8,9}};
        static int[][] matrixB = new int[3][] {
                                 new int[] {2,3,4},
                                 new int[] {5,6,7},
                                 new int[] {8,9,1}};
        static int[][] matrixC = new int[3][] {
                                 new int[] {3,4,5},
                                 new int[] {6,7,8},
                                 new int[] {9,1,2}};
        static int[][] matrixAB = new int[3][];
        static int numberOfActiveThreads;
        static AutoResetEvent[] eventNotifiers = new AutoResetEvent[3];
```

Invoking row multiplication (one thread for each):

```
public static void MupltiplyByThread(int[][] matrixA,int[][] matrixB) {
        Parallel.For (0, matrixA.Length,rowIndex =>
    {
                 startThread(rowIndex,matrixA[rowIndex], matrixB);
                });
                WaitHandle.WaitAll(eventNotifiers);
        Console.WriteLine("All done");
}
```

Starting a thread:

```
private static void startThread(int rowIndex,int[] rowA,int[][] matrixB){
  Console.WriteLine("Index is : " + rowIndex);
        // using Interlocked method
        System.Threading.Interlocked.Increment(ref numberOfActiveThreads);
        eventNotifiers[rowIndex] = new AutoResetEvent(false);

        ThreadParameters parameters = new ThreadParameters();
        parameters.threadId = rowIndex;
        parameters.rowA = rowA;
        parameters.matrixB = matrixB;
        ThreadPool.QueueUserWorkItem(
          new WaitCallback(RowMultiplierThread),(object)parameters);
}
```

RowMultiplier thread:

```
public static void RowMultiplierThread(object parameters) {
        ThreadParameters threadParam = (ThreadParameters)parameters;
        int rowIndex = threadParam.threadId;
        int[] rowA = threadParam.rowA;
        int[][] matrixB = threadParam.matrixB;
        Console.WriteLine("Starting Multiplication on row "+rowIndex);
        int[] result = new int[3];
        for (int j=0;j<matrixB.Length;++j){
      for (int k=0;k<matrixB[j].Length;++k){
          result[k] += rowA[j]*matrixB[j][k];
      }
        }
        Console.WriteLine("Done!");
  matrixAB[rowIndex] = new int[3];
  matrixAB[rowIndex] = result;
  // set a signal, we're done!
  eventNotifiers[rowIndex].Set();
        System.Threading.Interlocked.Decrement(ref numberOfActiveThreads);
}
```

13.16 Scala

Scala [OSV11] is a language that mixes functional programming paradigms (like referential transparency, first class and higher order functions, type inference, infinite data structures and pattern matching) and object-oriented design concepts. It was created by Martin Odersky at the EPFL university in 2003, and is developed with a very impressive community support. According to the recent ZeroTurnaround Developer Productivity Reports [Zer12], 11% of developers are using it in a way or another at the moment – and how it supports development

of concurrent applications highly contributes to that. Its popularity is reflected through the words of James Strachan (author of the Groovy programming language)[23] who stated that *"I can honestly say if someone had shown me the Programming in Scala book by Martin Odersky, Lex Spoon and Bill Venners back in 2003 I'd probably have never created Groovy."*.[24]

Scala is built on top of the Java Virtual Machine, maintaining strong interoperability with Java. This means that any Java class can be used seamlessly in Scala code and vice versa. Moreover, Scala can be run either in an interpreted environment (in a Read-Evaluate-Print-Loop, REPL), as a script or as compiled code.

Odersky founded Typesafe[25] which is the company behind the language. The board advisors include people like James Gosling (*creator of the Java programming language*) and Doug Lea (*concurrency expert, author of the concurrent library support of Java* [GPB06]). They are working on creating a *full Scala Stack* that includes everything required for enterprise projects.

Scala has a succinct, lightweight syntax and a safe, performant, strong static type system. It proposes to use immutable data structures over mutable ones by default, and offers an extensive language and library support for sideeffect-less code. These are essential building blocks for creating concurrent software.

As we have seen in the previous sections, one root of the main problems is non-determinism caused by concurrent threads accessing a shared mutable state. However, if the mutable state can be avoided (by programming in a functional manner), we get a deterministic processing. Even the name Scala (meaning stairs in Italian) is a blend of the phrases *"scalable"* and *"language"*, which also reflects the design to support concurrent software development.

Shifting attention to concurrency, Scala offers an actor model in its standard library, in addition to the standard Java concurrency APIs. Typesafe have included Akka[26] in their stack, a separate open source framework providing actor based concurrency, where actors may also be distributed or combined with software transactional memory (often referred as *"transactors"*). Alternative CSP implementations for channel-based message passing are available through Communicating Scala Objects [Suf08], or simply via the native Java implementations like JCSP.

Actors in general

An alternative approach to handling concurrency is to define several, simple, independent entities concentrating on specific subtasks without any shared state at all, and by enabling communication through message passing.

[23] `http://groovy.codehaus.org/`
[24] `http://macstrac.blogspot.hu/2009/04/scala-as-long-term-replacement-for.html`
[25] `http://typesafe.com/`
[26] `http://akka.io`

Actors were originally intended for the research field of Artificial Intelligence in the mid 1970s [HBS73], [Agh87], [WP09]. Since then, the concept of actors and the actor-based model have appeared in several programming languages, so Scala's solution is not unique. Actually, it was inspired by Erlang (a language developed by Ericsson suitable for handling massive communication load, it has become an open source project). An example of it is the Facebook chat function.

13.16.1 Comparison with concurrent processes

The actor model is often compared to languages based on the concurrent process model (successors of the CSP and Occam languages for example).[27] It might seem identical as far as communication through message passing is concerned, but there are a few fundamental differences that worth mentioning.

- The processes are usually anonymous, while actors are entities known by their names. Note that languages using channels also identify them by name, but one does not know who on the other side of the channel is – to whom the component is sending the message;
- Communication in the concurrent process model is usually done in the form of rendezvous (i.e. it is a type of synchronous communication where the participants become blocked until their counterpart is ready to accept their messages);
- Communication in the actor model is asynchronous by default: execution of actors sending messages is not blocked, but the message is delivered into a mailbox instead of directly to the recipient. The recipient is continuously processing the messages in its mailbox until they are depleted. If the actor has nothing else to do then it is usually suspended.

Actors in Scala

The actor-based model is a simple way to do concurrency and distribution. Fundamentally, an actor is an independent entity which can be *active* (i.e. generate messages on its own) and *reactive* (i.e. wait for incoming messages and perform tasks only on those events). It has a message queue by default in which the incoming messages have accumulated, and they can be handled in any order. Actors can handle messages on their own or they can send messages to other actors. In addition they can create, manage, restart other actors. This results in a high level and convenient abstraction.

Scala offers several implementations for the actor-based model. It has a built-in framework in the standard library which is widely used, the widely known Scalaz library offers short and elegant definition and handling of actors, and the Akka and Lift frameworks share a common interface offering several additional features.

[27] http://en.wikipedia.org/wiki/Communicating_sequential_processes#Comparison_with_the_Actor_Model

To demonstrate the differences, an interesting property of the Akka framework is that it follows the *"Let it crash"* philosophy, which allows recovery of failed processes (e.g. from deadlocks). By utilizing *supervisor hierarchies*, it is possible to create self-healing systems and achieve an impressive "nine-nine" uptime theoretically, which is a considerable advantage for mission critical systems. It also supports different kind of messages (synchronous, asynchronous communication and even the usage of future objects) and pairs a very small fingerprint with relatively high performance (*"allows handling* 2.7 *million actors per gigabytes of heap supporting* 50 *million messages per second on a single medium-class machine"*).[28]

In the next sections, we will demonstrate the usage of the default actor API in the standard library through specific examples.

Defining a simple actor

Creating a standard Scala actor is straightforward: one must subclass from the `scala.actors.Actor` class and override its `act()` function.

```
import scala.actors.Actor

class Greeter extends Actor {
  def act() = {
    println("Greetings!")
  }
}
```

The actor then can be activated by simply calling its `start()` function. A minimal application that does this follows.

```
object Main extends App {
  val greeter = new Greeter()
  greeter.start()
}
```

Scala offers an extensive support for creating both internal and external domain specific languages. A good example is the API that the standard library offers. Let's take a look on the next code segment that demonstrates the usage of this feature:

[28] `http://akka.io/`

```
import scala.actors.Actor
import scala.actors.Actor._

object Main extends App {
  val classifierActor = actor {
    loop {
      receive {
        case i: Int => println("Received an integer: " + i)
        case s: String => println("Received a String: " + s)
        case x => println("Received: " + x)
      }
    }
  }

  classifierActor ! 256
  classifierActor ! "Actors are fun"
  classifierActor ! 0.11
}
```

The code above demonstrates how to create a simple actor with basic functionalities and how to send messages to it with the help of the built-in domain-specific language support.

First, the actor is defined by the `actor` statement which is a shortcut for creating a new class that is a subclass of `scala.actors.Actor`. Then there is a `loop` statement which is nothing more than a shortcut for an infinite loop. The loop then calls `receive` which blocks the execution of the actor and makes it wait until a new message is delivered into its message queue.

The actor's actual task uses another powerful feature of Scala (which is widely used in functional languages) called *pattern matching*. To avoid distraction of explaining new features, think of it as an improved *switch-case* statement that allows matching on types, values, regular expressions and with many advanced features. For the current example, we make a simple distinction on integer values and strings, and write a proper message on the output for each one. There is also a default case for any other input.

The last thing to note here is the use of the `!` operator (read as exclamation mark or bang operator) which sends a message to the actor. This is similar to the notation of CSP and Occam, but also identical to the syntax used for Erlang actors. Other alternative usable commands are listed in the following table:

Method name	Return value	Description
act	Unit[29]	This is the top-level method for an Actor that is called on `start()` and typically contains one of the methods below
!	Unit	Sends an asynchronous message the actor.
!!	Future	Sends an asynchronous message to the actor and immediately returns a future representing the reply value.
!?	Any	Sends a synchronous message to the actor and awaits reply.
receive	Any	Blocks until a new message is received
receiveWithin	Any	Like `receive`, but it is possible to specify a timeout for waiting
react	Nothing	Similar to `receive`, but it is a bit more efficient and cannot have a return a value
reactWithin	Nothing	Like `receiveWithin`, but it is a bit more efficient and cannot have a return a value
restart	Unit	Restarts the actor.

Table 13.2: Commonly used Actor methods. For a complete list, please refer to `http://www.scala-lang.org/api/current/#scala.actors.Actor`

Case study: A simple ping-pong architecture

The following example demonstrates how to write a simple ping-pong service with Scala actors. These services are common in distributed network applications to keep track of active clients to a server for example. The protocol is quite simple: when a server wants to check if a given client connection is still alive, it sends a *ping* message to it. If a *pong* event is received in the server, it acknowledges the message and the connection is considered as a living one. The IRC chat protocol, for instance, defines this kind of event passing.[30]

[30] For the details, please consult the Internet Relay Chat RFCs 2810, 2811, 2812 and 2813, and visit the `http://irchelp.org/irchelp/rfc/` website.

First, we define a few classes that represent different messages.[31]

```
case object Ping
case object Pong
case object Stop
```

We define the `Ponger` component first as an actor. It should handle *ping* messages with a proper reply, and terminate on any *stop* message.

```
import scala.actors.Actor
import scala.actors.Actor._

class Ponger extends Actor {
  def act() {
    loop {
      receive {
        case Ping =>
          println("Pong: ping message received")
          sender ! Pong
        case Stop =>
          println("Pong: stop")
          exit()
      }
    }
  }
}
```

The other component is slightly different. When it starts, it sends a *ping* message by default. Then, it submits additional messages sequentially at a specified number of times. If all messages have been submitted and the proper replies have been collected, it sends the *stop* message.

[31] With a rough simplification, case classes are simple classes primarily used for pattern matching. Case objects are the same, except for singletons.

```
class Pinger(count: Int, pong: Actor) extends Actor {
  def act() {
    println("Sending initial message")
    pong ! Ping
    var messagesSent = 1

    loop {
      receive {
        case Pong =>
          println("Ping: pong message received")
          if (messagesSent < count) {
            messagesSent += 1
            println("Ping: Sending ping message " + messagesSent)
            pong ! Ping
          } else {
            println("Ping: sending shutdown message and terminating")
            pong ! Stop
            exit()
          }
      }
    }
  }
}
```

13.16.2 Parallel collections

The following section supposes that the reader is familiar with the basic concepts of functional programming, such as traversing, mapping, filtering, folding and reducing of collections.

Scala also offers an alternative approach to handling concurrent algorithms within the form of *parallel collections* [PBRO11], as several languages support this kind of approach (like the Cilk++ library [Lei09] for C++).

The idea behind parallel collections is that it requires a considerable effort to write the boilerplate code required to introduce parallelism. Providing a convenient and high-level abstraction on common tasks is a natural need in order to save programmers from low-level concurrency details.

So given the most common data structures in every programming language, why not create alternatives that can handle concurrent execution in a transparent and lightweight way? Our next simple sequential example converts a list of strings into upper case variants through a functional approach by the `map` operator:

```
val list = List("Dallas", "Ripley", "Lambert", "Brett",
             "Kane", "Ash", "Parker")
list.map( _.toUpperCase )
```

Converting the list to a parallel one where all the conversion processes (here the `toUpperCase` calls) take place in several different threads automatically is done by simply calling the `par` function on the list.

```
list.par.map( _.toUpperCase )
```

Behind the scene, Scala automatically creates a new copy of the original collection that supports parallel execution. Then a set of threads is spawned instantly in an automatized way and starts processing the elements of the collection in a non-deterministic order. Scala offers various implementations including arrays, maps, sets, ranges, trees and vectors. The collection framework defines an extensible interface so it can also be extended easily. For a detailed description see [PBRO11].

As it is demonstrated by the next code snippet, parallel collections can be created on their own as simple collections, and can be converted back to standard sequential collections anytime by `seq`.

```
import scala.collection.parallel.immutable.ParVector
val parlist1 = ParVector(1,2,3)

val parlist2 = List(1,2,3).par

val simpleList = parlist1.seq
```

It is a great advantage of parallel collections that they may be used during common operations like traversing, mapping, filtering, folding and reducing. However, the semantics of the parallel collections presents an interesting case. It is elegant and trivial to use them, but they perform a lot of processes in the background simultaneously and automatically. Nevertheless in a few specific cases they can also lead to nondeterministic program execution.

How does the parallel collections work?

We have seen examples for the parallel collection framework of Scala. However, they serve demonstrative purposes only.

The creation of parallel collections (i.e. creating a copy of the original collection), creating the threads in the background, distributing the work and reassembling the results mean a considerable overhead which can be huge when creating benchmarks on small sized collections. Typically, it worth experimenting with this approach over several thousands of elements, or in situations when the processing is based on network operations for example.

An excellent use case is when there is a slow HTTP request to download specific data from the web. Using parallel processing might improve the processing considerably as the most of the time is spent on I/O operations that can be parallelized easily.

```
// A list of RSS feeds to process
val rssAddresses = List("http://stackoverflow.com/feeds", ...)

// Slow HTTP request to different hosts
def downloadRssData(rssAddr: String) = ...

// Sequential processing
users.foreach( downloadRssData( _ ) )

// Parallel processing with multiple threads
users.par.foreach( downloadRssData( _ ) )
```

For more information on its performance, one can find additional descriptions of the parallel collections[32] and detailed benchmark analyse in the official Scala documentation.[33]

A particularly interesting case is presented next. We have a list that we process concurrently. First, each element we process is printed on the output, and in the last line, we also print the whole constructed parallel collection.

```
val list = 1 to 100
val parList = list.par.map( i => { println(i); i} )
println( parList )
```

Running the code should give something like the following output:

```
88
1
51
76
52
2
89
3
...
```

This clearly indicates the nondeterministic way in which the elements are chosen. However, the last line shows that the original ordering is somehow reconstructed:

```
ParVector(1, 2, 3, 4, 5, 6, 7, 8, 9, ...)
```

To understand this behavior, we need to provide further information about the collections.

Associative operators

There are a few other things to keep in mind when using parallel collections. First of all, the executed operator should be *associative*. This is required as the

[32] http://docs.scala-lang.org/overviews/parallel-collections/overview.html
[33] http://docs.scala-lang.org/overviews/parallel-collections/performance.html

program execution is distributed, and we may never know in which order it has been executed between the split parts. Let us consider a simple operator such as subtraction or division in the Scala interpreter.

```
val pl = (1 to 100).par

// Execution might result in surprising results, like:
pl.reduce(_-_) // 4838
pl.reduce(_-_) // 4838
pl.reduce(_-_) // -1900
pl.reduce(_-_) // -2942
```

As the example above demonstrates, the order in which the subtraction has been executed is nondeterministic as we have a *non-associative operator*.

Certain operators which are *associative*, but *not commutative*, allow the same parallelization. A common example is the string concatenation function: it is associative (i.e. `("a" + "b") + "c" == "a" + ("b" + "c")`) but not commutative (i.e. `"a" + "b" != "b" + "a"`). However, when we execute the next example, the output is deterministic, and is always `"abcd"`.

```
val pv = ParVector("a", "b", "c", "d")
pv.reduce( _ + _)
```

The reason for this behavior is that while the operations executed on the parallel collections can be executed in any order, it does not mean that the result will also be recombined in the same order. So even if the collection was assembled into different parts in an arbitrary order, the result is always reassembled in the original order.

Side effects

The operator that is executed on the parallel collection may has a *side effect*. A typical example is modifying a shared variable, which – as we have seen before – rise a race condition and lead to a nondeterministic program execution.

A simple example is shown below, where we execute a *read-modify-write* statement during the mapping operation:

```
var sum = 0

val parlist = (1 to 100).par

parlist.foreach( sum += _ )
```

In the example above (when executed with a large enough collection), we might encounter *disappearing* values from the `sum` local variable. The reason is the same what have seen in introductory section of this chapter: two threads read the previous value, execute the addition, and update its value right after each other - resulting in a disappeared component from the total `sum` value.

13.17 General tips for creating concurrent software

As we have seen in this chapter, writing concurrent software that does its job correctly is far from trivial. However, many common pitfalls can be avoided if we bear in mind a few issues suggested by Robert C. Martin and Brett L. Schuchert [Mar08]. Some of their advice is useful when developing a single-process software as well, but their importance is emphasized in a concurrent environment.

13.17.1 Single responsibility principle

The *Single Responsibility Principle* (SRP, [Mar02]) in software design states that a method (or a class, or a component) is allowed to change only because of a single reason. Since the parallel approach is complex enough to change the structure of the software, it is advisable to keep the concurrent building blocks separated from the other parts of the software: *"the concurrent code should have a separate development, modification and fine tuning life cycle during the development"* [Mar08].

13.17.2 Restrict access to shared resources

As we have seen in the introduction of this chapter, multiple processes modifying a shared resource (like a data structure) at the same time might have surprising results and a program state. The solution is that to identify all the possible access to these resources and restrict access by defining *critical sections*. These sections must be handled with care because, on one hand, they decrease efficiency, but on the other hand:

- The chance of forgetting the use of synchronized access to the common resource increases with their number. Remember: if there is even a single access to that resource where the synchronization is left by accident, our software is hopelessly broken;
- It will be hard to trace down any concurrency-related errors, because the concurrent parts are scattered around the whole sourcebase.

13.17.3 Independency

It is a great advantage if the different threads owned by the same process can be separated even on the level of data access. If all these threads can live in their "own world", they do not share data, they do not interfere with each other. Thus, most of the big caveats of concurrent code execution can be avoided (including deadlocks, performance loss on synchronization points and race conditions).

This is not always possible, but there are a few workarounds that we can still do. One of them is to *use immutable* (i.e. non-modifiable) views on the common data sources or only a copy of the original data. The result is that we can eliminate all synchronized access to the shared resource, and the performance gain

may balance the loss on initialization and cloning the original data. Moreover, the software becomes easier to understand without synchronization and easier to modify.

Some languages heavily support immutable data structures, while others (like Scala) use them by default (e.g. by importing the immutable data structures into the default scope by default and by using the `val` keyword). As Steve Jenson, a Twitter engineer suggests: *"Start with immutability, then use mutability where you find appropriate."*

13.17.4 Do not reinvent the wheel!

Writing concurrent applications usually lead to the same design decisions, issues to think of, bug reports and tool support that is required efficiently solve them.

The most important thing is to utilize the knowledge we have gained through industrial and academic examples: the problems we face are always the same, relying on design patterns successfully applied in other scenarios is a great advantage. Building on bullet-proof library support helps us avoid the need to write common building blocks (which is a huge and error prone enough task on its own).

Know the potential problems

Concurrent code has its own issues, but they can be classified into some major classes. Having a considerable knowledge about these issues (i.e. what can lead to them, in which way they modify the program behavior, the best practices to avoid them and their solutions) is the key to designing robust, scalable multithreaded applications.

Creating more and more concurrent code helps to build up experience and to get a "sixth sense" feeling when looking on a codebase. In this sense spending days of work hunting down trivial locking errors is not a waste.

The importance of synchronized access to shared resources and the concepts of deadlocks, livelocks and starvation have been discussed in details within this chapter, including suggestions as how to deal with them.

Know the common patterns

Most of the concurrent tasks we face in our daily work can be classified into three major groups. These cases can be described as the *Producers-Consumers* (Section 13.9.1), *Readers-Writers* (Section 13.9.2), and *Dining Philosophers problem* (Section 13.9.3). Knowing these problems and understanding the algorithms that solve them help one to get a better understanding of the issues of concurrent programming.

These execution models are well-known and commonly used, their understanding is essential when approaching concurrent problems.

13.17.5 Know the library support

Modern programming languages offer a wide set of tools, both low-level (like native language support for handling synchronization and monitors) and high-level building blocks (like built-in semaphores, locks, barriers, thread-safe collections and execution frameworks) to support concurrent software development. Understanding the available tools like semaphores, locks, barriers, concurrent data structures and additional libraries is a great advantage. These building blocks are commonly used, implementing them would be waste of time (as we have seen, it is far from trivial to implement even a simple semaphore), and it is unnecessary since we already have bullet-proof implementations. If we just have to read the documentation, why don't do so?

13.17.6 Write thread-safe modules

Encapsulating and making multithreaded components "pluggable" is always an advantage. First of all, it becomes possible to develop, profile and examine the component that allows fine-tuning the application to find and balance the optimal running conditions.

Another great advantage of encapsulating and documenting synchronization policies is that client-side locking can be evaded. Client-side locking violates the *Don't Repeat Yourself principle* [HT99], since the same instructions (locking the resource, executing the required operation and unlocking the resource) must be repeatedly used in each and every place in the source code. It does not only become cumbersome in the long run, but also easy to forget, making their use an error-prone task.

13.17.7 Testing

It is always a good idea to perform tests for a multithreaded application in different environments, on different architectures, operating systems and hardwares. Since all of them follow different rules for scheduling, switching processes and optimizing multithreaded applications, they can help hidden issues come to the light. If a strange program behavior or a paranormal activity occur, consider it as potential thread issue! These kind of issues are commonly known as *heisenbugs*.[34]

An extensive exploration of different parameter settings of a concurrent software (e.g. using more threads than the physically available central processing units) could also help the process.

Automated tools also offer several features that can be utilized to test concurrent applications. For instance, stress-testing applications (i.e. generating a

[34] According to the Wikipedia, it is a term used for a software bug that seems to disappear or alter its behavior when one attempts to study it. It is a pun on the name of Werner Heisenberg, the physicist who first asserted the observer effect of quantum mechanics, which states that the act of observing a system inevitably alters its state.

massive number of simulated user interactions) might reveal issues which otherwise would manifest only under heavy workload in the production environment. There exist tools specifically designed for finding concurrency-related issues like CHESS [MQB07], ConTest [Ric07] or MultithreadedTC [IEE07].

Using standardized software development processes like introducing code review, automated static [APM07] and/or dynamic code analyzer tools and profilers also reveal potential deadlocks.

13.18 Summary

In this chapter we have concentrated on the most important language constructs for concurrent software development. We have tried to demonstrate the most common problems with concurrency through a set of abstract examples with excluding language-specific solutions. Through a set of small case studies, we have described a selection of the basic issues in concurrency (shared variables, race conditions, deadlocks, livelocks), and how easy it is to fall into their traps.

Then we introduced the elementary models of communication and synchronization, including busy waiting, mutual exclusion, the problems of critical sections, the semaphore and the monitor. We have also included a short summary about how languages have solved the same problems with different techniques over time.

We have also included a taxonomy of concurrent applications, programming languages and the advantages of and the main threats in concurrent software. Common execution models are important for many reasons, they offer a guiding assistance when designing concurrent applications. The most common issues have been illustrated through; other, less common execution models are included in the following *Exercises* section. Furthermore, we have introduced the available toolbox in some carefully selected languages and frameworks including Ada, CSP, Occam, Java, C#, MPI and Scala.

We have given a concise guide to parallel programming tools, and hopefully we have been able to show the common pitfalls, their solutions, and the gains of embracing concurrency. Overall, we hope we have given a helping hand to determine the right programming model for the Reader's application and its environment. It requires a different state of mind to work with concurrent code, and we hope our chapter has helped the Reader acquire the elementary knowledge.

13.19 Exercises

In the following examples, after finishing the implementation, try running the examples with different configurations, on different hardware architectures and operating systems. Ask yourself if you can reason about the correctness of the

program, and try to identify locations in the source code that may be the source of potential synchronization or liveness issues.

Exercise 13.1. Create alternative implementations for the *Producers-Consumers* in your preferred programming language.

Exercise 13.2. Create alternative implementations for the *Dining Philosophers problem* in your preferred programming language.

Exercise 13.3. Create an implementation for the *Readers-Writers problem* in your preferred programming language.

Exercise 13.4. The *Cigarette Smokers problem*[35] was proposed by Suhas Patil in 1971 to in support of Dijkstra's semaphores [Pat71], for which he used his example. His arguments were not widely accepted, since he made additional restrictions on the problem (e.g. he ruled out the usage of conditional statements or multiple semaphores for example).

The problem may be illustrated by *three chain smokers* around a table, and each with specific resource for smoking. They have an infinite supply of tobacco, cigarette paper to roll the cigarettes and matches to light them, respectively. The smokers are not allowed to directly seize the required resources in each others pocket, but there is an *arbiter* who periodically chooses two of the smokers randomly, and asks them to put their resources in the middle of the table. When the resources are available, he notifies the third smoker to roll and smoke a cigarette. While the smoker is smoking he does not participate in any other interactions. Meanwhile, the arbiter (seeing that there are no more materials left in the middle of the table) chooses two smokers again. This process is repeated infinitely.

Create an implementation for the *Cigarette Smokers problem* in a preferred programming language. Compare the implementations by their complexity, length of the source code and supported language primitives for concurrent execution.

Exercise 13.5. The *Sleeping barber problem*[36] is also a widely known classical problem for inter-process communication and synchronization, attributed to E. W. Dijkstra.

There is a barber shop where a barber is working with a single chair and a waiting room with limited space. From time to time, a new client comes into the shop and checks if the barber is available. If the barber is free, the client instantly gets a hair cut immediately. If the barber is cutting the another client's hair, the new client checks in the waiting room (if there are unoccupied chairs, left for him, he sits down, joins the waiting queue, otherwise he leaves the shop). When finishing the hair cut, the barber checks if there is anybody waiting for him. If there is a client waiting, he asks him to sit into the barber chair; if their

[35] `http://en.wikipedia.org/wiki/Cigarette_smokers_problem`
[36] `http://en.wikipedia.org/wiki/Sleeping_barber_problem`

is nobody waiting, he sits in his chair and falls asleep. He only wakes up when a new client enters the shop. This process is repeated infinitely.

Create an implementation for the *Sleeping barber problem* in a preferred programming language. Compare the implementations by their complexity, length of the source code and supported language primitives for concurrent execution.

13.20 Useful tips

The tips that follow are given in Java.

Tip 13.1. For the Producers-Consumer problem the `ExecutorService` could be used together with `LinkedBlockingQueue`.

Tip 13.2. Dining Philosophers can be easily implemented using simple `Threads` (to represent a Philosopher and the Table that stores the tableware)

Tip 13.3. The Readers-Writers problem can be solved using `Locks` as semaphores, but Java offers a specific `Lock` class called `ReentrantReadWriteLock`, which is ideal to solving this issue.

Tip 13.4. Please note that in Cigarette Smokers problem, beside the smokers there is another actor called Arbiter who picks one type of component periodically. All actors can be implemented as `Threads`

Tip 13.5. The BarberShop and the clients can be implemented as `Threads`. Please note that the waiting room must be a finite queue, so a capacity must be defined to control standard. One option is to use infinite queues such as `LinkedBlockingQueue` initialized with a given capacity. Other solution would be the use of finite arrays with static length. In this case you have to check the actual size of the array against the capacity.

13.21 Solutions

Solution 13.1. Producers-Consumers problem

1. Main class:

```
package producerconsumer;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class Main {
    public static void main(String[] args)
                        throws InterruptedException {

        Container container = new Container(1);
```

```
        ExecutorService executorService = null;
        executorService = Executors.newFixedThreadPool(7);

        executorService.execute(new Consumer(1, container));
        executorService.execute(new Consumer(2, container));

        for (int i = 0; i < 100; ++i) {
            executorService.submit(new Producer(i, container));
        }
        executorService.awaitTermination(2, TimeUnit.MINUTES);
        executorService.shutdown();
        container.setCloseContainer(true);
    }
}
```

2. Producer class:

```
package producerconsumer;

import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Producer implements Runnable {
    private int id;
    private Random randomGenerator = new Random();
    private Container container;
    public Producer(int id, Container cont) {
        this.id = id;
        this.container = cont;
    }
    public void run() {// every producer creates only one product
        String product = produce();
        boolean putFinished = false;
        while (!putFinished) {
            if (!container.putProduct(product)) {
                try {
                    System.out.println
                    ("Producer" + id + ":Could not put product");
                    Thread.sleep(100);
                } catch (InterruptedException ex) {
                  ex.printStackTrace();
                }
            } else {
                System.out.println("Producer" + id +
                " created and sent : " + product);
                putFinished = true;
            }
        }
    }
    public String produce() {
        try {
            Thread.sleep(randomGenerator.nextInt(1000));
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
        String product = id + "_" + randomGenerator.nextInt(1000);
        return product;
    }
}
```

3. Consumer class:

```
package producerconsumer;
import java.util.Random;
import java.util.logging.Level;
```

```
import java.util.logging.Logger;
public class Consumer implements Runnable {
    private int id;
    private Container container;
    private Random randomGenerator = new Random();
    public Consumer(int id, Container container) {
        this.id = id;
        this.container = container;
    }
    public void run() {
        String product = container.getProduct();
        while (!container.isCloseContainer() || product != null) {
            try {
                System.out.println
                ("Consumer" + id + " polled product " + product);
                Thread.sleep(randomGenerator.nextInt(1000));
                product = container.getProduct();
            } catch (InterruptedException ex) {
     ex.printStackTrace();
            }
        }
    }
}
```

4. Container class:

```
package producerconsumer;

import java.util.concurrent.LinkedBlockingQueue;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Container {
    private boolean closeContainer = false;
    private LinkedBlockingQueue container;
    private int capacity;

    public Container(int capacity) {
        container = new LinkedBlockingQueue();
        this.capacity = capacity;
    }

    public boolean isCloseContainer() {
        return closeContainer;
    }

    public void setCloseContainer(boolean closeContainer) {
        this.closeContainer = closeContainer;
    }

    public boolean putProduct(String product) {
        if (container.size() < capacity) {
            try {
                this.container.put(product);
            } catch (InterruptedException ex) {
ex.printStackTrace();
            }
            return true;
        } else {
            return false;
        }
    }

    public synchronized String getProduct() {
        return (String) this.container.poll();
```

```
    }
}
```

Solution 13.2. Dining philosophers problem

1. Main class:

```
package diningphilosophers;

public class Main {
    public static void main(String[] args) {
        Table table = new Table(7);
        for (int i = 0; i < 7; ++i) {
            Philosopher p = new Philosopher(i, table);
            p.start();
        }
    }
}
```

2. Philosopher class:

```
package diningphilosophers;

import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Philosopher extends Thread {
    private Table table;
    private int id;
    private Random randomGenerator = new Random();

    public Philosopher(int id, Table table) {
        this.id = id;
        this.table = table;
    }

    public void run() {
        while (true) {
            try {
                think();
                int leftFork = (id) % table.getNumberOfParticipants();
                int rightFork=(id+1 % table.getNumberOfParticipants();
                if (leftFork < 0) {
                    leftFork += table.getNumberOfParticipants();
                }
                if (table.grabFork(leftFork)) {
                    if (table.grabFork(rightFork)) {
                        try {
                            doEatMyMeal();
                        } finally {
                            table.releaseFork(leftFork);
                            table.releaseFork(rightFork);
                        }
                    } else {
                        table.releaseFork(leftFork);
                    }
                } else {
                    think();
                }
            } catch (InterruptedException ex) {
ex.printStackTrace();
            }
        }
    }
```

```
    public void think() {
        try {
            System.out.println("Philosopher " + id + " is thinking.");

            Philosopher.sleep(randomGenerator.nextInt(1000));
        } catch (InterruptedException ex) {
    ex.printStackTrace();
        }
    }

    public void doEatMyMeal() {
        try {
            System.out.println("Philosopher " + this.id + " is eating.");
            Philosopher.sleep(randomGenerator.nextInt(1000));
        } catch (InterruptedException ex) {
    ex.printStackTrace();
        }
    }
}
```

3. Table class:

```
package diningphilosophers;

import java.util.ArrayList;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class Table {
    private ArrayList<Lock> tableware;
    private int numberOfParticipants;
    public int getNumberOfParticipants() {
        return numberOfParticipants;
    }
    public void setNumberOfParticipants(int numberOfParticipants) {
        this.numberOfParticipants = numberOfParticipants;
    }
    public ArrayList<Lock> getTableware() {
        return tableware;
    }
    public void setTableware(ArrayList<Lock> tableware) {
        this.tableware = tableware;
    }
    public boolean grabFork(int forkId) throws InterruptedException {
        return (this.tableware.get(forkId).tryLock());
    }
    public void releaseFork(int forkId) {
        this.tableware.get(forkId).unlock();
    }
    public Table(int numberOfParticipants) {
        this.numberOfParticipants = numberOfParticipants;
        tableware = new ArrayList();
        for (int i = 0; i < numberOfParticipants; ++i) {
            Lock l = new ReentrantLock();
            tableware.add(l);
        }
    }
}
```

Solution 13.3. Readers-writers problem

1. Main class:

```
package readerswriters;
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Main {
    public static void main(String[] args) {
        ExecutorService executorService = null;
        executorService = Executors.newFixedThreadPool(7);

        SharedData sd = new SharedData();

        for (int i = 0; i < 10; ++i) {
            executorService.execute(new Reader(i, sd));
            executorService.execute(new Writer(i, sd));
        }

        try {
            executorService.awaitTermination(1, TimeUnit.MINUTES);
        } catch (InterruptedException ex) {
ex.printStackTrace();
        }
        executorService.shutdown();
    }
}
```

2. Reader class:

```
package readerswriters;

import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Reader implements Runnable {
    private SharedData sharedData;
    private int Id;
    private Random randomGenerator = new Random();

    public Reader(int Id, SharedData data) {
        this.sharedData = data;
        this.Id = Id;
    }

    public void run() {
        for (int i = 0; i < 10; ++i) {
            try {
                Thread.sleep(randomGenerator.nextInt(1000));
                System.out.println("Read SharedData " +
                                   Id + ":" + sharedData.getData());
            } catch (InterruptedException ex) {
ex.printStackTrace();
            }
        }
    }
}
```

3. Shared Data class:

```
package readerswriters;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class SharedData {
    private int data;
    private final ReentrantReadWriteLock readWriteLock
```

```
            = new ReentrantReadWriteLock();
    private final Lock write;
    private final Lock read;

    public SharedData() {
        write = readWriteLock.writeLock();
        read = readWriteLock.readLock();
    }

    public void setData(int newData) {
        write.lock();
        try {
            data = newData;
        } finally {
            write.unlock();
        }
    }

    public int getData() {
        read.lock();
        try {
            return data;
        } finally {
            read.unlock();
        }
    }
}
```

4. Writer class:

```
package readerswriters;

import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.Random;

public class Writer implements Runnable {
    private SharedData sharedData;
    private int Id;
    private Random randomGenerator = new Random();

    public Writer(int Id, SharedData data) {
        this.sharedData = data;
        this.Id = Id;
    }

    public void run() {
        for (int i = 0; i < 10; ++i) {
            try {
                Thread.sleep(randomGenerator.nextInt(1000));
                System.out.println("Write to SharedData by Writer" + Id);
                sharedData.setData(Id);
            } catch (InterruptedException ex) {
     ex.printStackTrace();
            }
        }
    }
}
```

Solution 13.4. Cigarette Smokers problem

1. Main class:

```
package cigarettesmokers;
```

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Boolean> table = new ArrayList();

        table.add(false); // tobacco on table
        table.add(false); // paper on table
        table.add(false); // match on table

        Arbiter arb = new Arbiter(table);

        Smoker s0 = new Smoker(table, arb, 0);
        Smoker s1 = new Smoker(table, arb, 1);
        Smoker s2 = new Smoker(table, arb, 2);

        arb.addSmoker(s0);
        arb.addSmoker(s1);
        arb.addSmoker(s2);

        s0.start();
        s1.start();
        s2.start();
        arb.start();
    }
}
```

2. Arbiter class:

```
package cigarettesmokers;

import java.util.ArrayList;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Arbiter extends Thread {
    private ArrayList<Boolean> table;
    private ArrayList<Smoker> smokers;

    public Arbiter(ArrayList<Boolean> table) {
        this.table = table;
        this.smokers = new ArrayList();

    }

    public void addSmoker(Smoker s) {
        smokers.add(s);
    }

    public void run() {
        while (true) {
            try {
                Arbiter.sleep(1000);
            } catch (InterruptedException ex) {
ex.printStackTrace();
            }

            // Picking a component
            Random componentGenerator = new Random();
            int chosenSmoker = componentGenerator.nextInt(3);
            //asking the others to place a component onto the table
            //table.add(chosenSmoker,Boolean.TRUE);
            for (int i = 0; i < 3; ++i) {
                if (i != chosenSmoker) {
```

```
                    smokers.get(i).addYourComponent();
                }
            }
            if (canINotifyToSmoke()) {
                System.out.println("I took " + chosenSmoker +
                " to the table, let's notify the Smoker");

                smokers.get(chosenSmoker).wakeUpAndSmoke();
                pause();
            } else {
                System.out.println("Nope... I need to sleep");
                try {
                    Arbiter.sleep(500);
                } catch (InterruptedException ex) {
    ex.printStackTrace();
                }
            }
        }
    }

    public boolean canINotifyToSmoke() {
        int components = 0;
        for (int i = 0; i < 3; ++i) {
            if (table.get(i)) {
                ++components;
            }
        }
        return (components > 1);
    }

    public synchronized void pause() {
        try {
            wait();
        } catch (InterruptedException ex) {
    ex.printStackTrace();
        }
    }

    public synchronized void wakeUp() {
        this.notify();
    }
}
```

3. Smoker class:

```
package cigarettesmokers;

import java.util.ArrayList;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Smoker extends Thread {
    private ArrayList<Boolean> table;
    private Arbiter arbiter;
    private int smokerId;

    public Smoker(ArrayList<Boolean> table,
                  Arbiter arbiter, int smokerId) {
        this.table = table;
        this.arbiter = arbiter;
        this.smokerId = smokerId;
    }

    public void run() {
        while (true) {
```

```
                if (canISmoke()) {
                    try {
                        System.out.println(smokerId+" is smoking");
                        table.add(0, Boolean.FALSE);
                        table.add(1, Boolean.FALSE);
                        table.add(2, Boolean.FALSE);
                        notifyArbiter();
                        Smoker.sleep(1000);

                        System.out.println("I'm finished");

                    } catch (InterruptedException ex) {
            ex.printStackTrace();
                    }
                } else {
                    System.out.println(smokerId +":I cannot smoke");
                    pause();
                }
            }
        }

        public void addYourComponent() {
            table.add(smokerId, Boolean.TRUE);
        }

        public synchronized void notifyArbiter() {
            arbiter.wakeUp();
        }

        public boolean canISmoke() {
            int components = 0;
            for (int i = 0; i < 3; ++i) {
                if (i != smokerId) {
                    if (table.get(i)) {
                        ++components;
                    }
                }
            }
            return (components > 1);

        }

        public synchronized void pause() {
            try {

                this.wait();
            } catch (InterruptedException ex) {
            ex.printStackTrace();
            }
        }

        public synchronized void wakeUpAndSmoke() {
            this.notify();
        }
    }
```

Solution 13.5. Sleeping barber's problem

1. Main class:

```
package sleepingbarber;

public class Main {

    public static void main(String[] args) throws InterruptedException {
```

```
        BarberShop b = new BarberShop();
        b.start();

        for (int i = 0; i < 5; ++i) {
            HairyClient client = new HairyClient(b, Integer.toString(i));
            client.start();
        }
    }
}
```

2. Barbershop class:

```
package sleepingbarber;

import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.logging.Level;
import java.util.logging.Logger;

public class BarberShop extends Thread {
    private BlockingQueue<HairyClient> waitingRoom
        = new LinkedBlockingQueue();
    private int capacity = 5;
    private boolean occupied = false;

    public BlockingQueue<HairyClient> getWaitingRoom() {
        return waitingRoom;
    }

    public void setWaitingRoom(BlockingQueue<HairyClient> waitingRoom) {
        this.waitingRoom = waitingRoom;
    }

    public boolean isOccupied() {
        return occupied;
    }

    public void setOccupied(boolean occupied) {
        this.occupied = occupied;
    }

    public boolean isThereAnyFreeSpace() {
        return (waitingRoom.size() < capacity);
    }

    public void run() {
        while (true) {
            //is there anyone waiting?
            if (waitingRoom.size() > 0) {
                try {
                    //wait until the next person is getting available;
                    HairyClient client = waitingRoom.take();

                    client.wakeUp();
                    setOccupied(true);
                    cutting(client);
                    // okay, she's hair has been cut, looks better..
                    setOccupied(false);
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            } else {
                try {
                    BarberShop.sleep(1000);
```

```
                } catch (InterruptedException ex) {
                    ex.printStackTrace();
                }
            }
        }
    }

    public void cutting(HairyClient client) {
        System.out.println("Working: client is" + client.getClientName());
        try {
            BarberShop.sleep(1000);
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }

    public synchronized void wakeUp() {
        this.notify();
    }

    public void wakeUpAndCut(HairyClient client) {
        setOccupied(true);
        wakeUp();
        cutting(client);
        setOccupied(false);
    }
}
```

3. HairyClient class:

```
package sleepingbarber;

import java.util.logging.Level;
import java.util.logging.Logger;

public class HairyClient extends Thread {
    private BarberShop myBarber;
    private String clientName;

    public String getClientName() {
        return clientName;
    }

    public void setClientName(String clientName) {
        this.clientName = clientName;
    }

    public HairyClient(BarberShop barber, String clientName) {
        myBarber = barber;
        this.clientName = clientName;
    }

    public void run() {

        if (myBarber.isOccupied()) {
            if (myBarber.isThereAnyFreeSpace()) {
                System.out.println("There are free place");
                try {
                    myBarber.getWaitingRoom().put(this);
                    this.pause();
                } catch (InterruptedException ex) {
                }

            }
        } else {
```

```
                myBarber.wakeUpAndCut(this);
            }
        }

        public synchronized void pause() {
            try {
                wait();
            } catch (InterruptedException ex) {
            }
        }

        public synchronized void wakeUp() {
            this.notify();
        }
    }
```

14 Program libraries

In this chapter we will be acquainted with the question, how to design and develop program libraries. The requirements and design aspects for program libraries, as well as needed knowledge for object-oriented library design will be discussed in detail. The main features of the standard program libraries in some prevailing program languages will also be mentioned.

Programming languages and environments usually include standard built in program libraries which implement the basic services. The development of bigger program systems may often require to create a special API of their own. This chapter explains the design aspects required for the above purpose.

The most important from these aspects are: correctness, efficiency, reliability, extensibility, reusability, and appropriate documentation. Design principles of class structure and hierarchy, size and grouping, security features, naming conventions and memory management will be covered in detail.

The approaches of standard libraries from Java [Nyek08], C++ [Str00], Eiffel [Mey91] and some other typical languages will be reviewed as examples. We also present the most typical representatives of programming paradigms (functional, aspect-oriented, etc.) independent of the object-oriented approach.

Hereafter, program libraries are referred to as collections of subprograms, modules, classes and data types which contain program code implementing a well defined set of program services, and offer a unified (nowadays usually object-oriented) interface for user programmers. That is why they cannot be viewed as end user programs, but rather as (software-)building blocks to facilitate the work of the programmers. The collection aspect must be emphasized: module design principles already discussed in Section 9.3. are also valid for program libraries, but all those must be applied on a higher, more generic level.

Services of program libraries can be categorized as shown on Figure 14.1.

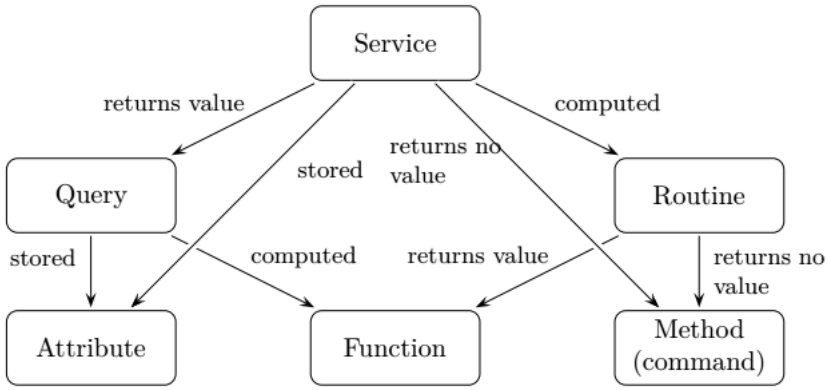


Figure 14.1: Services of program libraries

Further way for grouping services are provided by packages (Java) or clusters (Eiffel). Program libraries and packages can have a one-to-many, or a many-to-one relationship with each other. Packages are generally directly stored on physical data level, such as files or directory structure.

14.1 Requirements against program libraries

The quality of the program library must be a very important factor, independently – if it is a graphical toolset, or a b2b interface – of its goal. General software quality requirements also apply, but there are more specific demands, especially for program libraries.

Before listing these requirements, we will try to describe which personal skills should library designers have.

14.1.1 Skills of a good program library developer

Making a program library is practically the same as developing a complex software system. That is why the following requirements should be fulfilled by everyone designing and developing complex software consisting of numerous parts.

This list is not exhaustive, but contains the most important points. A good program library developer should:

- have a good abstraction skill – is capable of generalization from specific phenomena;
- have a good sense of details – in a program library every detail counts (there is no ”good enough” library, it must be complete);
- know every aspect of the language – knowing the advantages, disadvantages and what construct is best suitable for which purpose;

- be orderly – is capable of classification, can find the right place for everything (it is not practical if the elements of a program library are named or sorted inconsequently);
- have a literary sense – description remarks must have proper style and elegance;
- be an excellent programmer and designer – can understand the needs of future users of the library (where is generality necessary, and where should it be more efficient, etc.);
- be a team player – individual style must be suppressed, the goal is a uniform and consistent working style (creativity can be displayed in the variety of services provided by the library).

Having these attributes are absolutely essential to be able to start developing program libraries which fulfill the following requirements to the fullest extent.

14.1.2 Basic quality requirements

Basic requirements are the same as for general software development: correctness, efficiency, reliability, extensibility and reusability (see the Introduction). In case of program libraries these requirements apply even more.

Correctness

Correctness of a software means that a program solves exactly the problem and fits the desired specification. This is the first and most important criterion, since if a program is not working like it should, other requirements do not really count.

For program libraries, because of their complexity it is especially important to have a detailed, comprehensive, deliberate design and specification. That is why particular care must be taken – with the help of language constructs, or even by applying "administrative" tools such as CASE,[1] round-trip engineering,[2] software quality assurance methods and standards (CMM,[3] SPICE,[4] Bootstrap,[5]

[1] Computer-Aided Software Engineering. Software tools for automation of some system design and development steps by applying certain design and development methods based on e.g. code generators and reverse engineering code. Nowadays the most widespread modeling standard is the UML, the most popular CASE tools for this are: ArgoUML, Oracle Designer, Paradigm+, Rational Rose, Software Through Pictures, Together.

[2] This denotes the process, when source code of legacy systems produced by code analyzer tools is transformed to a model using specific design notations, from which after redesigning or developing source code will be generated again without loosing its original elements (code not noted in model).

[3] Capability Maturity Model. A method developed by Carnegie Mellon University which focuses on grading software developer organizations on a five-point scale for integrating quality assurance in the lifecycle of their products.

[4] Software Process Improvement and Capability dEtermination, it is an ISO standardized technique, such as CMM, but more general and more complex.

[5] A method related to SPICE and CMM, developed by a consortium of European firms.

RUP,[6] ISO 9001[7] etc.) –, to have the software product fully comply with the stated requirements specified during design in every respect.

Efficiency

The efficiency of a program is proportional to the running time and used memory size. The faster (CPU time, I/O, network load) and the less memory and storage are used, the more efficient it is.

These requirements often counteract each other. A faster run is often set off by bigger memory requirements, and vice versa.

According to Bjarne Stroustrup, creator of the C++ language, efficiency and generality are contradictory requirements. He means that in the case of creating program libraries, efficiency is more important, while if not so, the users (developers) would start to write their own more efficient code for certain purposes. This would violate standardization attempts, and lead to increased number of incompatible software.

This is contradicted by the point of view that if a service is not designed generic enough, and has for efficiency too many ”hardcodings” than developers may sometimes also start replacing library elements by their own code segments to satisfy special needs. Obviously, there is no good solution for every case to this problem. The designer of the program library must keep the main goals of the library in mind, when deciding this question.

Obviously, the C++ language is mostly used for efficiency requiring tasks, so program libraries based on it also rather focus on that. For example, in the Standard Template Library, inheritance is mostly banished for performance reasons, obviously loosing so generality. In higher level languages, generality can easily be what is more required.

Reliability

A program is called reliable, if abnormal – not described in the specification – circumstances lead to no catastrophe, but are handled in a ”reasonable” way.

This definition shows that reliability is a notion by far not as precise as correctness. One could say, of course, with a more specific specification reliability would mean correctness exactly, but in practice there are always cases which are not covered by specification explicitly.

For example, in C if an array stores n bytes, and the specification does not specify a range check to prevent overflow, when a programmer has not paid attention to reliability and let the byte on position $n + 1$ to be overwritten,

[6] Rational Unified Process. A software production process based on UML, covering the full lifecycle by using products of Rational and helping the job of designers, developers and testers.

[7] Quality assurance standard not for the development process of software, but for its endresult product.

the program will show unexplainable behavior (another variable may have been overwritten), or a protection exception from the operating system will be raised.

That is why reliability and security are very important aspects in a widely used program library. For implementing this, pre- and postconditions, and invariants can be used. This actually means a contract between the authors and users of the program library where preconditions are obligations of the user and security for the programmer, postconditions are obligations of the programmer and security for the user. Now the programmer needs no ”defensive” programming (to be prepared for every possible error and input) and the code also gets more efficient and simpler. The two possible approaches:

- The tolerant approach: program library routines have no preconditions (or only weak) and react somehow to every possible input. This, of course, has the consequence that because of the often unnecessary rigorous checking of the input, efficiency of the code strongly degrades.
- The pretentious approach: every routine has strong preconditions, the fulfillment of those is the responsibility of the user. In this case, input data checking can be safely discarded, but for error-prone input the right behavior of the program cannot be guaranteed.

As can be seen, both cases have their advantages and disadvantages. The recommended approach:

- Only preconditions required for the logically correct execution of abstract operations are checked
- Only those conditions are checked that would seriously affect efficiency, if not met.

Any violation of this contract indicates a program error. Violating the precondition indicates error from the user side, postcondition or invariant failure denotes program library error. So after developing the program library before distribution for efficiency only checking of preconditions should be switched on.

Extensibility and maintainability

Extensibility refers to how easy it is to adjust the program product to specification changes.

Users often demand further development, modification, adjustment of the program product to new external conditions. According to some surveys 70 %of program product costs are spent on maintenance, so it is understandable that this requirement significantly affects the quality of the program.

Extensibility is relevant, especially when developing big program systems and program libraries, since changes in small programs are usually never too complex. To increase extensibility, design simplicity and decentralization (to have the more independent modules and components) can be seen as the two most important basic principles (see next section about object-oriented program library design).

Reusability

In current times, when the amount of software products to develop is extremely high, it cannot be allowed to build every system from scratch. There is a growing demand for reusable components.[8]

If a software system is designed with reusability in mind, this requires a little bit more effort which will pay back at the next development by already having done a lot.

14.1.3 Special requirements for program libraries

The following requirements are specifically for those software products which are used not by endusers, but by other software developers. As program libraries are specifically made for this purpose, all of this is particularly true with regard to them.

Classes from a program library can be expected to have the following properties:

- Easy and intuitive usage;
- Immediate and wide usability in a wide range of software systems;
- High level documentation;
- Portability and compatibility.

In the following, these requirements will be discusses in detail.

Easy and intuitive usage

Services offered by program libraries should be easily understandable and after some usage easy to remember.

Easy usability can be examined in many ways. One of these is the separation of services. If a program library has many similar services, the user will have a difficult choice to select just the right one. In general, it is better to define less, nearly completely orthogonal[9] services – in every case to suit a well defined role, following the optimal implementation.

Generality (accommodation to the possible uses) and simplicity (achieving every task by combining elemental operations) contradict each other. Professional programmers need the primitive operations for efficiency; on the other hand, casual programmers need more general components and accommodation of conventions for convenience.

[8] As a component many things are considered, for example, software components with runtime connection capabilities through interfaces (e.g. CORBA, COM, EJB), or even standard program language types (such as in C++ the Standard Template Library or in Java the classes from its API). The naming is valid in both cases, but we will use it for components which can be combined on source code level.

[9] Implementing separate functional aspects or dimensions see aspect-oriented programming and the Multi-dimensional Separation of Concerns

Unified naming convention also greatly helps the usage of a program library. This means consistent naming, on the level of services based on categorization of the operations with the implemented data types, following these rules:

- The name of getter/query services for data elements should start with the *get* prefix;
- The name of setter services for data elements should start with the *set* prefix;
- The name of services to remove data elements should start with the *remove* prefix;
- The name of services returning a logical value should start with the *is* prefix.

Of course, this list can also be freely continued. Please note that these prefixes come from the English language, because in the field of programming English is dominant without a doubt.

Following these, a unified naming system can be easily adopted. Like for prefixes, unified naming should be used also for the same attributes of data types:

- For the capacity of a given structure: *capacity*;
- For the element count of a given structure: *count*;
- For the emptiness of a given structure: *empty*; etc.

Combination of these words can be used to name services, such as *setCapacity*, *getCount*, or *isEmpty*. Capitalization of each word part also makes the adoption of the naming convention easy.

Based on these examples, the following principles can be summarized:

- The name should be short (usually one word), but communicative;
- The name of a service should not refer the containing class (except when renaming at multiple inheritance);
- The name of the classes should be always a noun or noun structure;
- Names of commands (procedures) should be verbs (in imperative), maybe with supplementary nouns;
- Names of not logical getters should be nouns or nous structures;
- The name of a logical getter should be an adjective which suggests a yes/no answer, or the *is* prefix should be used;
- The name of the service should be interpretable from the perspective of the target object, since by calling a service there is always an object (the target object) which offers the service;
- Operation and query pairs should be named with the same stem (extendible – extend).

Immediate, broad usability

Reusability is the feature of the software products that they can be partly or as a whole reused in new applications and in a wide range of software systems.

The experience should be utilized that many elements of software systems follow common patterns. Reimplementing already solved problems should be avoided.

This question is particularly important, not primarily when producing individual program products, but for a global optimization of software development, as the more reusable components are available to help problem solving, the more energy remains to improve other quality characteristics (at the same costs).

High level documentation

A particularly important attribute of well usable program libraries is the accurate, well structured documentation.

This has the function to guide the user (programmer) easily through the multitude of services, and to be able to optimally utilize available possibilities.

When writing the documentation, the following should be considered:

- Sourcecode is not abstract enough. Beside the relevant information for the user it also contains the low level implementation, so reading it means browsing through too much information, or could lead the programmers to use implementation possibilities not intended for the public. It has the advantage that it is always up to date.
- Separate documentation can become inconsistent with the software. It has the advantage that it contains only relevant information for the user.

Regarding this two aspects the recommended way of ”internal documentation” can be specified: documentation should be embedded in the source code. It has the big advantage that source code and documentation are always consistent (as changing the code definitely changes also the documentation), and the generation of the documentation can be automated by using utility programs (such as javadoc).

Portability and compatibility

Portability regards how easy it is to convert the program to another machine, configuration or operating system – usually to run in different runtime environments.

Compatibility shows how easy it is to combine the software products with each other. Programs are developed not isolated, so efficiency can go up by orders of magnitude, if ready software can be simply connected to other systems, and adapts well to background infrastructure (operating system, database engine, web server, 3D API, XML schema, etc.).

Portability and compatibility often contradict each other, since, for example, if a program library is designed to efficiently use all possibilities of a particular operating system, then obviously, even only converting to another operating system may be difficult, and efficiency would certainly suffer. The same is true vice versa: if an API is designed for maximum portability, possibilities of different environments would most likely not be fully utilized.

To achieve a healthy compromise, the main goal of the particular library must be kept in mind, and the decision has to be made according to that if portability or compatibility should be more emphasized.

14.1.4 Conditions for fulfillment of the requirements

As a result of the requirements listed above, the following properties can be expected from a good program library:

- Consistency – every component (types, classes, modules and packages) of a library should be as a result of comprehensive and coherent planning, and should follow numerous systematic, explicit and unified conventions;
- Components should be homogen and of good quality;
- At design time, object-oriented approach should be applied, since supporting data abstraction is particularly suited for developing program libraries.

Simultaneous compliance to only a part of the above criteria is very hard. It can be, for example, that the most generic algorithm is not the most efficient for many special, but frequent cases.

Most of these requirement – mainly reusability, extensibility and compatibility – can be best supported, if libraries are designed according to the principles of modularity discussed in Section 9.3. As already stated there, object-oriented approach is a good way to complete modular design and to handle problems originated from there efficiently, therefore suitable for the production of high quality program libraries. In the following a short overview of the object-oriented method will be given from the aspect of the program libraries.

14.2 Object-oriented program library design

Usability, quality, extensibility and reusability of an object-oriented program library is very significantly influenced by the number of its classes (compared to the possibilities and the problem) how big these classes are, in what kind of a hierarchy they are structured, and which interconnections they have.

According to this the following few pages will describe which approach should be chosen, when designing the class hierarchy of program libraries. After that we will review what number of services should preferably a class have, and to implement these services what number of parameters should be chosen (certainly

considering properties of the given task). Finally, some typical class types will be introduced based on their place within the class hierarchy and on technical characteristics.

14.2.1 Class hierarchy

Developing a program library with object-oriented approach also means building a class hierarchy. In terms of usability the complexity or simplicity of this hierarchy is not unimportant.

Conforming to object-oriented approach the most generic classes should be introduced, needed special classes should be defined by using inheritance. Doing so makes maintenance of the class hierarchy easy.

Generalization

The proper use of generalization is shown in the following example in Objective-C. The two code snippets illustrate the definition of classes implementing a generic window and a dialog box.

```
// WRONG APPROACH!
@interface MainWindow : Object
  // ...
  - show;
  - moveToX: (int)x Y: (int)y;
  - onExit;
@end

@interface DialogBox : Object
  // ...
  - show;
  - moveToX: (int)x Y: (int)y;
  - onOK;
@end
```

```
// Proper approach
@interface Window : Object
  // ...
  - show;
  - moveToX: (int)x Y: (int)y;
@end

@interface MainWindow : Window
  // ...
  - onExit;
@end

@interface DialogBox : Window
  // ...
  - onOK;
@end
```

In the first incorrect case, two classes are defined without applying generalization, these are similar to each other, since they both are concrete specializations of the same general concept (window).

As in this case, there is no general window class, certain properties (*show* and *moveToX:Y:* methods) are defined in both places the same way. This results in the need of changing these services on two different places, if required what causes on the one hand extra work making maintainability complicated. On the

other hand if changes are applied only on one place inadvertently, inconsistency occurs which deteriorates program quality.

In the second code snippet, there is a *Window*, the other two classes inherit from this. This means on the one hand the collection of common behavior in one place which improves maintainability and program quality, on the other hand this helps reusability, since such a common window type was constructed which could serve later as a base to any other window classes.

Specialization

The following Smalltalk [GR83] code is also a good example to show how defining proper generic ancestors make concrete type specification much easier by specialization, as opposed to implementing each of those independently without applying inheritance.

```
class Animal superclass Object
  instance variables
    color
    yearOfBirth
    habitat
  class methods
    new . . .
  instance methods
    eat . . .
    drink . . .
    sleep . . .

class Dog superclass Animal
  instance variables
    DateOfRabiesVaccination
  class methods
    new . . .
  instance methods
    walk . . .

class Cow superclass Animal
  instance variables
    dailyMilkYield
  class methods
    new . . .
  instance methods
    chew . . .
```

This example shows clearly that the resulting code is much shorter, if generalization – specialization is used, since if the common part would not be separated,

it would be defined twofold. Furthermore, if the animal hamster should also be implemented, separating the common part brings even more, since that must be coded only once instead of threefold.

Simple or complex hierarchy

Introducing a new class could make the class hierarchy simpler. This is mostly of interest when applying multiple inheritance, since introducing an additional node class results in fewer inheritance relations.

The following Eiffel example will demonstrate this.

```
-- WRONG APPROACH!
class RADIOBUTTON inherit WINDOW INPUT_COMPONENT
  -- ...
end

class CHECKBOX inherit WINDOW INPUT_COMPONENT
  -- ...
end

class LISTBOX inherit WINDOW INPUT_COMPONENT
  -- ...
end

-- Proper approach
class INPUT_COMPONENT_WINDOW
  inherit WINDOW INPUT_COMPONENT
  -- ...
end

class RADIOBUTTON inherit INPUT_COMPONENT_WINDOW
  -- ...
end

class CHECKBOX inherit INPUT_COMPONENT_WINDOW
  -- ...
end

class LISTBOX inherit INPUT_COMPONENT_WINDOW
  -- ...
end
```

The example is also shown on the UML diagram 14.2. to aid the overview of class hierarchies.

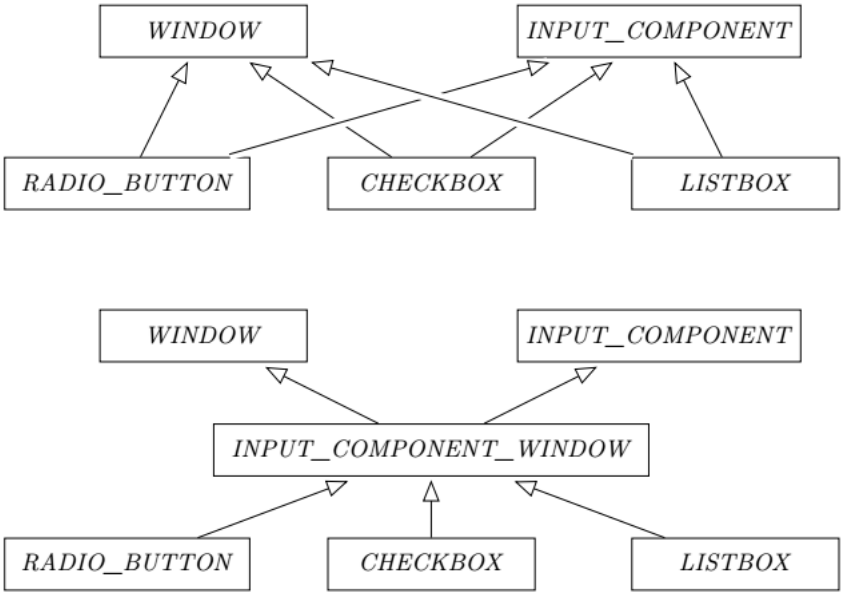


Figure 14.2: UML diagram of the two variants of the Eiffel example

The example also shows that the size of the code now is not reduced, but its transparency increased significantly by having much fewer inheritance relations, and the program also becomes logically clearer.

This is because that typegroup was aggregated into a separate class which is used on manyfold as common part, so it is highly probable that it can cope by itself as an independent notation within the logic of the program.

14.2.2 Size of the classes

In the following, some notions regarding class size will be discussed, and some statistical data will be presented to help choose the right size for classes in program libraries.

Simple, direct and incremental size

The size of a class can be determined the following way:

- Direct size = the count of the newly introduced direct (not abstract and nor inherited or redefined) services of the class;
- Incremental size = the count of the inherited and redefined services;
- Simple (full) size = the number of all services implemented by the class = direct + incremental size.

The following C# [Sch02] example will be analyzed to explain these metrics.

```
using System;

public class TelephoneSet {
  public string telephoneNumber() {
    string telephoneNumber;
    // Defining telephone number
    return telephoneNumber;
  }
  public void endOfCall() {
    telephoneReceiver.putDown();
  }
}

public class PortableTelephoneSet : TelephoneSet {
  public void endOfCall() {
    bigRedButton.push();
  }
  public int lengthOfAntenna() {
    int length;
    // Defining length value
    return length;
  }
}
```

In this example the direct size of the class *PortableTelephoneSet* is 1, because of the newly introduced *lengthOfAntenna* method. Counting the redefined *endOfCall* and the inherited *telephoneNumber* methods the incremental size is 2, together the simple size gives 3.

Outer and inner size

Another aspect can be if the metric includes exported attributes or not: according to this the outer and inner sizes are differentiated (the inner includes all, the outer only public members).

The outer and inner sizes will not differ significantly if public services are implemented not by private operations, but are delegated to other classes.

Upper limit due to handling

If rules should be given for the class size, the proper metric must be chosen first.

The full size should not be limited, since it greatly depends on the number of ancestor classes and their complexity. The direct and incremental sizes are

relevant for the class source. A natural choice could be the incremental size, since this is what really correlates to complexity.

Below a certain size the class becomes unmanageable. This problem can be approached in two ways:

- *Minimalist approach*: a class in a program library contains only the most basic services, but no redundant ones (those which could be implemented by using basic ones). This, of course, is better for the writer of the program library, since this requires less work and it also means simpler maintenance.
- *"Shopping list" approach*: every service should be included in the class which fulfills the following requirements:
  - The service fits into the implementation scheme of the abstract data type;
  - It would not break the correctness of the class, i.e. the class invariant;
  - It implements useful functionality;
  - It does not duplicate an already existing service;
  - It fulfills the requirements of easy usability.

In practice, the standard built in libraries of program languages are expected to apply the minimalist approach, so every operation should be executed efficiently only in one possible way, designing other program libraries should rather apply the shopping list approach to ease its usage.

The next example shows the shopping list approach:

```
(* Objective CAML example *)
module Table :
sig
  val cell_paint : int int int -> unit
  val row_paint : int int -> unit
  val column_paint : int int -> unit
  val paint_all : int -> unit
  (* other functions *)
end =
struct
  (* implementation *)
end
```

The *Table* type in this example implements a simple spreadsheet model. It offers methods to color the whole table at once, or just a single row, column or a specific cell. Input parameters are obviously the row and column numbers, and the color code.

Organizing principles

Too many services negatively affect the learning curve to use the program library, so the following guiding principles are recommended to avoid this:

- Every service should be specified with a well defined contract, and only the specification, not the implementation should be used;
- Class documentation should be unified and in an easy to read format, it should contain the contracts of all services, but should hide implementation details;
- For the names of services a rigorous convention should be applied;
- Services should be grouped by exactly defined categories, these categories should be the same for each class, their order in the documentation should be the same, within each category the services should be listed in alphabetical order.

Example data

Generally speaking, if a class contains more than 80 services, the class hierarchy should be considered for splitting.

Number of services	Prevalence of classes (%)
0 – 5	40 – 60
6 – 10	10 – 20
11 – 15	5 – 10
16 – 20	5 – 10
21 – 40	5 – 15
41 – 80	5 – 10
81–	0 – 5

Table 14.1: Distribution of class sizes in Eiffel program libraries

Table 14.1 shows the suggested size distribution of classes based on data from Eiffel program libraries.

14.2.3 Size of services

As using program libraries means calling their implemented services, it is vital for the user to know the parameter count, types and order of each service. So from the aspect of usage, the size of a service can be specified as the parameter count.

This size strongly depends on the task of the program library, as for example, methods of a GUI implementation, or for statistical computations numerous arguments could be required.

For better handling numerous arguments, they should be divided into the following two groups:

- Parameter: input argument to specify a value for an operation;
- Option: such an argument which could be even omitted, since it could have a default value. It only affects the handling of the parameters. During the development of a service its parameter list should not change, but new options can be safely added.

Consequently, to reduce the size of a service only parameters should be used in the argument list. Options further should be represented as separate attributes with setter and getter services. This is the so called option setting method. This, of course, raises the question of globality for the representation of the new attribute, and even also increases the size of the class.

The following PHP example illustrates the option setting method:

```
<?php
class FileManager {          // traditional class
  function open($filename ,$mode, $use_path) {
    return fopen($filename ,$mode, $use_path);
  }
}

class FileManagerWithOptions { // Option manager class

  var $m_mode;
  var $m_path;

  function setMode($mode) {
    $this->m_mode = $mode;
  }
  function setUsePath($use_path) {
    $this->m_path = $use_path;
  }
  function getMode() {
    return $this->m_mode;
  }
  function getUsePath() {
    return $this->m_path;
  }
  function open($filename) {
    return fopen($filename, $this->m_mode, $this->m_path);
  }
}
```

```
// using traditional class
$fm = new FileManager;
$fm->open("/etc/passwd", "r", 0);

// using option manager class
$fmwo = new FileManagerWithOptions;
$fmwo->setMode("r");
$fmwo->setUsePath(0);
$fmwo->open("/etc/passwd");
?>
```

In this example the *FileManager* class has only one method with three arguments, the same method in the *FileManagerWithOptions* class has even only one parameter, the two former arguments became attributes with setter and getter methods.

Example data

Average parameter count of the services is usually around 0.5. Services with more than 5 parameters should be considered for splitting. Ratio of queries–commands within services is approximately 60–40 %.

Table 14.2 shows the suggested size distribution of services based on classes from Eiffel program libraries.

Argument count	Service prevalence (%)
0	50 – 60
1	30 – 35
2	5 – 15
3	0 – 5
4	0 – 5
5–	0 – 5

Table 14.2: Distribution of argument count in class services of the Eiffel program library

14.2.4 Types of classes

In the following, some types of classes from program libraries will be characterized.

Concrete types

Every program library hierarchy contains types representing basic data structures (list, string, date, complex number, etc.). These are called concrete types.

Concrete types can be characterized as follows:

- They fit a concrete notion and mode of implementation;
- Clarity, independent usability;
- They are strongly dependent of, but have efficient implementation, so at any change all user code must be recompiled;
- They depend only minimally on other classes;
- They can be compiled and used isolated, but cannot be inherited from;
- Being concrete is the opposite of abstract, so they have no generic or abstract properties, and are usually used in low level programming.

The following example shows the possibility of the Java language which supports defining classes being so concrete that inheriting from them is explicitly forbidden. This is done with the use of the **final** keyword.

In the first codesnippet the whole class is made final.

```
// concrete class
final class ChessAlgorithm {
  // . . .
}
```

If a class specifies this class as its parent, the following error message will be shown by the compiler:

```
ChessAlgorithm.java:6: Can't subclass final classes: class ChessAlgorithm
class BetterChessAlgorithm extends ChessAlgorithm . . .
      ^
1 error
```

In the next example only one method will be made final:

```
// Ancestor class with final method
class ChessAlgorithm {
    // . . .
    final void nextStep(ChessMan toMove, TablePosition newPos) {
        // . . .
    }
}
```

In this case the class can be specialized, but the method marked as **final** cannot be redefined.

Abstract types

Those classes are called abstract which are needed to represent an important logical unit within the class hierarchy, but cannot be directly instantiated, only through their descendants. Their role is to introduce a given interface, such as a set, list, tree, etc.

Abstract types can be implemented by the help of concrete, derived types. Properties of abstract types are:

- For the same access path manifold implementations can exist;
- Efficient memory usage and short running time thanks to virtual methods.

Abstract class definition is supported in many languages by appropriate keywords. For example, the Java, Ada 95, C# languages include the **abstract** keyword, C++ denotes with the *= 0* postfix that a member function is purely virtual, i.e. abstract. Usually it is possible to specify if an operation is abstract or concrete

The following Ada 95 example shows an abstract set type with a concrete implementation (hashtable based set type).

```
package Set is
  type TSet is abstract tagged null record;
    function Size(h: in TSet) return Integer is abstract;
    function IsEmpty(h: in TSet) return Boolean;
    procedure Add(h: in out TSet, data: in Float) is abstract;
    procedure Remove(h: in out TSet, data: in Float) is abstract;
    -- Specifications of other abstract operations.

  type THashTableSet is new TSet with private;
    function Size(h: in THashTableSet) return Integer;
    procedure Add(h: in out THashTableSet, data: in Float);
    procedure Remove(h: in out THashTableSet, data: in Float);
    -- Specifications of other abstract operations.
  private
    -- Implementation of the THashTableSet type.
end Set;

package body Set is
  -- Implementation of the operations of the TSet type.
  function IsEmpty(h: in TSet) return Boolean is
    begin
      return Size(h) = 0;
    end IsEmpty;

  -- Implementation of the operations of the THashTableSet type.
  function Size(h: in THashTableSet) return Integer is
    m : Integer := 0;
    begin
      -- Computing the size.
      return m;
    end Size;
```

```
  procedure Add(h: in out THashTableSet, data: in Float) is
    begin
      -- Adding an element.
    end Add;

  procedure Remove(h: in out THashTableSet, data: in Float) is
    begin
      -- Removing an element.
    end Remove;

  -- Implementation of other operations.
end Set;
```

Node types

Node types are located as inner nodes within the inheritance hierarchy. As already shown (see Section 14.2.1), they play an important role in simplifying class hierarchy. Characteristics of node types are:

- They implement services of the ancestor classes, meanwhile extending their interfaces with virtual methods and also giving an implementation for these;
- They are dependent from their ancestors;
- One can inherit from them;
- They can be instantiated.

An example: in a geometrical application the *quadrant* type would be a descendant of *shape*, but also may be refined (*parallelogram*, *trapezoid*, etc.).

Fat interfaces

This is a kind of interface type which is independent of other classes, introduces a lot of services, and gives only for the most important services an implementation. It declares the more special services as virtual, and for the sake of instantiation also gives an empty implementation for them, where usually an error message is thrown signaling that the actual implementation is still missing.

An example: consider the general container classes which implement storage of the elements by their own, but sorting or enumerating elements for a given condition is not really their responsibility.

Application frames

These abstract classes implement actually a miniature application. Within the class only the program logic is implemented, every other methods to execute and parametrize basic operations are virtual.

An example: consider the filter class which iterates over all elements of an input data stream and executes a certain operation if a condition is met. Iteration over the input data stream, checking a given condition and calling a certain operation, and possible error handling are all implemented by the class itself.

Handler classes

The connection between an abstract interface and its concrete implementation during runtime, even the size of the implementing class is constant, but the need for handling different sized classes through a given interface can arise. This is what a handler class can be used for. Using it divides the implementation in two parts: the actual representation and a handler object for accessing this representation. This can actually be seen as the object-oriented implementation of the pointer type.

An example: with the help of file handler classes arbitrary sized data files can be represented.

14.3 New paradigms

In the nineties new theories and methods were introduced for improving program language modularity by differentiating or extending the methods of OOP. Applying these can improve program library quality characteristics.

As an example, consider aspect-oriented programming (AOP) which focuses on gathering the aspects throughout the class hierarchy into separate modular units, or the generative programming (GP) which focuses on creating such program libraries or components after modeling an application domain which can be used by knowing the concrete problem to simply and as fully as possible generate code from the plan of the system. This latter is related to intentional programming (IP) which is capable of generating full code from program libraries of adequately designed and structured data abstractions.

These theories and their language implementations will be handled in Chapter 17 about aspect-oriented programming.

14.4 Standard program libraries

As stated above, high level languages and development environments usually offer their services in the form of standard program libraries. The program language in fact is restricted to its basic statements and keywords which would be not enough to solve everyday problems on their own. This could also be expressed as the usability of the program language is strongly determined by the feature richness of its standard program library. Such a high level program

language is, for example, Java, which has a well known wide spread standard library with loads of services.

Considering low level languages – such as assembly – they also use premade program library services – for example, through interrupts. This is not so obvious, because the called program library is not stored on a data storage, but is within the ROM of the computer, so its services are useable without any linking operations.

Standard program libraries offer in/for a specific program language such services which would be needed by almost all programmers. A good example for this is the implementation of the communication between the program and its user, since without data input and result display only a very few programs could properly function. Services of a program library can be used also from other program languages. This could happen on low level, if compiled units would be called from other languages, but according to its original syntax (when for example a C program calls an external routine with Pascal signature, or consider the *native* feature of the Java language), or the calling of a service could be done by an interpreter, but the use of CORBA services could also be considered where thanks to the common description language (IDL) the definition and the implementation of a given service are completely separated.

14.4.1 Data structures

One of the basic services a standard library must provide is to offer implementations of often used complex data types and structures for the programmers to aid their work. Since the plain program language only supports tools for type constructs to make more complex types from basic types, but these composite types are fairly often needed, so this is a good reason to include these in a program library.

A good example for this is the Java *Collection Framework*, or the *STL* library in C++ which implement "collections" of objects, so called containers, such as stack, bag or vector. The main design aspect in this case was the orthogonality of the element types, the container types and of their possible operations. This means that every basic type can be grouped in containers, and the elements of the container can be accessed only in a given way, with the help of so called iterators. Sorting of container elements is not the responsibility of the container, an arbitrary external algorithm can be created using the iterators, since the container only supports the management, access and modification of its elements.

14.4.2 I/O

Another basic programmer task is to support data input and display data output. This, of course, cannot be expected from the program language itself, since usually every data output and input is a predefined task of the operating system. A standard library can also only make easier the usage of these services of

the operating system. Consider file handling, or reading/writing input/output channels, or communicating through computer network, every program language must do this the same way.

14.4.3 Memory management

One of the key questions of program library design is memory management. The two fundamental problems which require attention:

- Allocating memory for new objects;
- Freeing memory.

Since the first is the task of the operating system, for the second problem two solution approaches exist:

- Automatic garbage collection (such as in Java, Eiffel, Smalltalk). This usually implies the introduction of references and the removal of pointer types. This has the advantage that the programmer need not be concerned about memory management, but has the disadvantage that because of the garbage collection mechanism it will surely not be optimal for concrete applications.
- Freeing objects by hand. This is more efficient, but also much more dangerous, as all the responsibility lays in the hands of the programmer, and the smallest inattention can lead to memory ”leaks”, or in worse case even to unreproducible program errors.

As you can see, the nature of these two problems depends of course also on the actual program language. It must be emphasize here again how important it is that a program library should implement a safe and efficient memory management, since its code will be used overall in many places, so the effects of a possible error would be multiplied.

14.5 Lifecycle of program libraries

As program libraries are such software products which are used to make program development easier, the most important aspect to comply with during their whole lifecycle would be, is reusability. During design such an interface must be defined which stays stable in long term and also enables further development. After the initial phase, when the program library is already used in many places, the question of maintainability gains more and more importance. The problem is with that that further developments and extensions must not affect the function of already used services (except changes made for bugfixing). Hereinafter the lifecycle of program libraries will be seen through, and it will be examined how to best meet the quality criteria discussed so far.

14.5.1 Design phase

Beside the basic principles of modular and object-oriented approach discussed so far, the following must also be considered:

- The problem of data structures is following: data structures are the most sensitive for changes, mainly for extensions. Consider how troublesome it is to append a new field to a record causing an increased memory footprint for the type. That is why extensibility of the data structures must already considered during design phase, for example, in a way by defining initially empty, non-used fields which can be used later for further development. Of course, it should also be considered how to distinguish different versions of data structures. One possible solution for this is if the user specifies for every service of the program library the expected version number as a parameter (this of course decreases efficiency because of the additional parameter passing), or before using any data structures of the program library, a special service must be called to set the requested version. In the latter case it is the responsibility of the program library to keep a record of the requested versions of possible parallel inquiries which only makes the implementation of the code more complex. The simplest solution for that would be if every output data structure stored its own version number, but this would increase the memory footprint.
- The program library should only allow indirect access to (global) variables and fields. For this, rather getter and setter services should be introduced, hiding internal implementation details of data fields and structures and controlling data access.
- For naming services unique identifiers should be used by specifying for every name a prefix (for example in PVM, a program library to aid parallel programming, the name of every service has the *pvm_* prefix), or if the language supports it, a separate namespace should be introduced (in C++ with the *namespace* keyword, or in Java by using packages).
- The interface (API) of the program library should specify only the really public services, so changing internal code used for implementation will have less effect on the user code.

14.5.2 Implementation phase

In this phase the most important aspect is the efficient and error-free implementation of the services. The following principles may be of assistance for this:

- Services should be defined in the strictest way. It must be thoroughly considered if a service should be on the class or object level, and the modifiers should be specified as strict as possible. For example, services not reachable from the outside should be marked as private, and it should also be stated (in Java for example with the *final* keyword), if a method

should be finalized. The compiler can only generate optimal code in this case.
- It follows from the principle before that only those operations should be made reachable from the outside which are documented services of the program library. This can assure that nobody uses not documented functions of the internal implementation.

14.5.3 Maintenance phase

For every bigger program library this is the longest phase. Maintenance could be done as bugfixing, or in the form of further development. The most important basic principle of bugfixing is that it must be definitely done, no excuses should be allowed that maybe some user code could be based exactly on this error prone behavior. If a programmer finds a bug, it should not be exploited, but reported to the maker of the program library.

At further development, the most important aspect is the implementation of version management, and keeping the most possible level of compatibility during applying modifications. The following principles may be of assistance for this:

- If new services are introduced, programs using these new version should not even execute with older versions of the program library. For this the solutions discussed for design phase could be applied.
- If the signature of a service was changed, for compatibility reasons the old version should also be included. In this case calling the old version would usually set the new parameters to some default values, and it will call the newly introduced version. Meanwhile a warning can be issued which could lead the users to change their code. Some program languages have support for this mechanism (such as in Java the *deprecated* keyword: calling methods marked like this will cause the compiler to issue a warning message).
- Every new version of the program library should be thoroughly tested. For this, parallel to development testbeds should always be defined and managed together with the library. To test new functions the testbed must also be extended, retesting existing services could protect against possible side effects.

14.6 Summary

In this chapter we have discussed the question, how to design and develop program libraries. The requirements and design aspects for program libraries, as well as needed knowledge for object-oriented library design and also for designers have been examined in detail. The main features of the standard program libraries in some prevailing program languages have been introduced.

15 Elements of functional programming languages

In declarative programming languages, programs are a set of declarations describing a process of computation. A functional program consists of type, class, and function declarations and definitions, together with a start expression. Execution of a program is virtually the evaluation of the start expression. The mathematical computation model of functional programming languages is the λ-calculus. In this chapter, we provide an overview of the most important concepts in functional programming, review its historical and mathematical background, while we also discuss issues such as expression power, efficiency, the functional programming style, language constructs for supporting abstractions, and readability, modifiability, reliability of programs written in such languages.

When discussing elements of functional programming, there will be multiple languages used (such as SML, Miranda, Clean, Haskell) in order to give an introduction to the functional programming style and common language constructs employed in contemporary functional programming languages [Hud89] rather than an extensive study of a single specific language ([CMP95], [Har01], [Tho99] and [Pla99]).[1]

15.1 Introduction

The mathematical foundation of functional programming is a computation model described by Church in 1932–33, called λ-*calculus* [Bar84]. The semantics of functional programming languages is usually defined by the λ-calculus. Turing showed that computable functions over non-negative integers expressible in λ-calculus are exactly the same as the functions that are computable in the model of imperative languages, called the *Turing machine*. That is, every problem that can be solved in the imperative computation domain can be also solved using functional programming concepts and vice versa. The model of functional programming is as old as the Turing machine. The first functional programming language (LISP) was also created around the same time (1956–1962) as were the first high-level imperative programming languages (FORTRAN, ALGOL 60) as well. This is not a coincidence as these two approaches do not exclude but support each other. Solving problems in programming requires one to be able to think both in imperative and functional concepts.

The growth of computing power and the continuous development of compiler techniques used in functional-language implementations has finally enabled functional programs to achieve the same level of efficiency as their imperative counterparts. For example, the run time of a binary produced by the Clean, OCaml or Haskell compilers is just as effective as if it was written using C.

[1] The writing of this chapter has been partially funded the OTKA T037742 grant.

The influence of functional programming languages is on the rise, not least because their approach suits the object-oriented paradigm. Many software companies use a general-purpose or domain-specific functional language for solving certain complex problems. Areas of application are telecommunications (Erlang, Ericsson), trading, networking, cloud computing (OCaml) or natural language processing, digital signal processing, cryptography, embedded systems, (Haskell, Lolita), traffic control systems, or building user interfaces (iTask, Clean).

Code written in a functional language is usually shorter, more expressive, more readable, and easier to modify than their respective versions implemented in imperative languages. This is due to the fact that the concept of variables employed does not exist in purely functional languages, hence programs cannot have implicit side effects. Modifying sections of a program has less effects on the complete software, therefore it is easier to follow. When using functional languages for implementation, the complexity of larger programs can substantially reduced, and the amount of time spent on developing the program may be cut back significantly, the result is a more reliable and more bug-free program.

Widespread adaptation of functional programming languages is mostly hampered by the lack of educated software developers with good abstraction skills.

15.1.1 The functional programming style

In the case of functional languages, instead of considering each program execution step individually, the focus is on the effect of the program which can be described by program functions (effect relations) or behavioral relations. When correctness of programs is discussed, the program function or the behavioral relation is compared to the problem to be solved. A component of the composed program may have unwanted side effects if subproblems are incorrectly identified during the program refinement process.[2] Side effects make verification of program correctness very cumbersome, and derivation rules in program construction become harder or impossible to apply. In addition, program synthesis and verification techniques based on preconditions and postconditions ([Tho90], [HAKP99] and [MEP01]) may not be applicable at all. In a broader sense, the *functional programming style* refers to the process of systematic program construction using components free from ”invisible” side effects.

Preconditions and postconditions describing the problem specify constraints on the values of variables.[3] Such constraints often refer to values of function compositions in the form of equations. Function compositions usually correspond to the structure of the program itself. That is, a solution can be found by constructing constraints from elementary functions using standard function compositions. This also indirectly determines how to map states satisfying the precondition to states characterized by the postcondition. Put it in a different

[2] Side effects may appear if the relation solving the problem is not an extended identity or it is not projection-invariant over the subspace of the subproblem and its complementary.

[3] Variables are considered projection functions of the state space.

way, it specifies how functions given in the postcondition should be computed through elementary functions. If it is allowed to use *higher-order function*s, that is, functions that may have functions as their parameters, it leads to a uniform functional approach.[4]

15.1.2 Structure and evaluation of functional programs

A *functional program* consists of type, type class and function declarations, together with function definitions and a *start expression*. Execution of a program is technically the evaluation of the start expression. Evaluation can be seen as gradually rewriting the start expression by textually replacing the referred functions with their bodies (see Example 15.1.2). The exact meaning of the rewriting process is determined by the computation model of the language. The execution model of programs written in functional languages always results in a confluent reduction or rewriting system. Such a system is called *confluent* if the final result is unaffected by the order of the rewriting steps taken, this only depends on whether the result is finally computed or not. The λ-calculus is a confluent system, but there are other similar confluent term rewriting systems and graph rewriting systems.[5] In many languages, for example, in SML, Haskell or Clean it is possible to directly use expressions conforming to the grammar rules of λ-calculus.

In the case of simple *function definitions*, the name of the function and the associated *formal parameters* appear on the left-hand side (lhs) of the defining equality, while the expression to calculate the corresponding value, that is, the *body of the function*, is placed on the right-hand side (rhs). Together with the semantics of the functional language and the method of evaluation chosen, definition of a function determines both the way how function values are computed and the amount of associated costs. Note that certain functions considered well-defined in mathematical sense cannot be used with the same formula in a functional language for reasons of efficiency and computability.

Sample definitions for a few simple functions are presented in Figure 15.1.[6] The returned value of function `zero` does not depend on parameter `x`, it is a constant function with value of `0`. Function `id` is the identity function which returns the value of its argument. Function `inc` returns a value (of type integer)

[4] Based on Hudak [Hud89], it is possible to prove that all the *elementary programs*, (SKIP, ABORT, assignment), and program constructions (sequencing, branching, looping) can be indeed expressed in functional style. For example, assignments can be defined as *higher-order function*s over variables.

[5] [PvE93] shows an example of translating Miranda programs to λ-calculus. Semantics of Haskell programs is given in two steps. First, the semantics of the language core is specified in λ-calculus, then it is given for all the other language constructs based on the core definitions [PH99]. Semantics of the Clean language is given by a graph rewriting system.

[6] For each example there will be always noted which functional language compilers should be able to accept them. E.g. SML: Moscow ML 2.0; Clean: Clean 2.4; Haskell: any Haskell 98 compiler, unless noted otherwise.

```
zero x = 0
id x = x
inc x = x + 1
square x = x * x
squareinc x = square (inc x)
fact n = product [1..n]
```

Figure 15.1: *(Miranda, Clean, and Haskell):* Simple function definitions.

greater by one of its argument. Function `square` computes the square of its argument, while `squareinc` is the composition of functions `inc` and `square`. Function `fact` is the factorial function.

```
fun square (x) = x * x;
val area = fn r => pi * r * r
fun plus a b = a + b
```

Figure 15.2: *(SML):* Simple function definitions.

In Figure 15.2, the same function definitions are given in SML. Arguments are enclosed by the `fn` keyword[7] and the `=>` symbol in the definition of `area`. Function definitions can be made simpler using the `fun` keyword as illustrated by the definitions of `square` and `plus`. The `area` function calculates the area of a circle of radius `r`, and the `plus` function adds up its arguments together.

Execution of functional programs is a sequence of *reduction* or *rewriting* steps commencing from a start expression. A reducible subexpression, called *redex*, is selected by the applied *reduction strategy* in each reduction step where the function application in the chosen *redex* is replaced with its *body* while actual values are assigned to the contained parameters. An expression reaches its *normal form* when there are no more reduction steps possible – which then becomes the final result of the performed reductions.

```
Start = sqrt 5.0        // Clean
main  = product [1..10] -- Haskell

Math.sqrt 5.0;          (* SML *)
```

Let us take a look at the start expressions presented above. The normal forms are as follows: `2.236068` (`Start`) and `3628800` (`main`). In the case of SML, the normal form is the real number `2.2360679775`.[8]

[7] The `fn` keyword corresponds to the lambda operator of λ-calculus.

[8] Square root of the real number `5.0` can only be calculated after the `Math` module has been imported in the SML interpreter. Lines must be terminated by the `;` symbol, which is the *symbol of evaluation* and instructs the interpreter to start evaluating.

There may be multiple redexes in an expression, which is often the case, and then the reduction strategy is to determine in what order they are going to be rewritten. In certain cases, multiple subexpressions may be reduced at the same time or their reduction may overlap, possibly making the execution faster.

In confluent systems, normal form of expressions do not depend on the actual reduction order, that is, the normal form is unambiguous[Bar84], though, not every expression has a normal form. It is determined by the method of evaluation if it is possible to reach the normal form. *Lazy evaluation* employed in Miranda, Clean and Haskell reduces the leftmost outermost redex[9] first in the equivalent λ-expression (that is, it applies the function definition first when the expression starts with a function symbol), and reduction of the arguments is performed only on demand. The lazy evaluation is a *normalizing reduction strategy* (Curry and Feys, 1958), that is, it always reaches the normal form if it exists. In contrast, *strict evaluation* employed in ML or LISP starts reducing with the leftmost innermost redex,[10] that is, it reduces the arguments first. Strict evaluation is often more efficient but it may not terminate even when a normal form exists. Thus, strictly evaluated (or just *strict* for short) languages often contain syntactical constructs to explicitly add lazy evaluation for certain expressions (such as the SML, where lazy lists may be used) and lazy languages often develop methods for the programmers to annotate if it is allowed to evaluate an expression using the strict strategy (such as Clean, whose compiler implements strictness analysis and the programmer may introduce strictness declarations).

The two most typical reduction strategies, lazy and strict, are illustrated below. Definitions of functions given in Figure 15.1will be used at each reduction step.

Reduction – strict

```
squareinc 7
-> square (inc 7)
-> square (7 + 1)
-> square 8
-> 8 * 8
-> 64
```

Reduction – lazy

```
squareinc 7
-> square (inc 7)
-> (inc 7) * (inc 7)
-> (7 + 1) * (7 + 1)
-> 8 * 8
-> 64
```

15.1.3 Features of modern functional languages

A programming language is considered *purely functional* if it is guaranteed that constructs of the language do not cause side effects, and there is no way to destruct the previous state of variables – called destructive updates – or to make them resemble those of the imperative languages.

Unlike languages such as LISP and SML, Hope, Miranda, Haskell and Clean are purely functional languages.

[9] The leftmost outermost redex is the first redex from the left which is not enclosed by any other redex.

[10] The innermost redex is a redex which does not contain any further redexes inside and it is the first one from the left.

Most important features of purely functional languages are summarized below.

Referential transparency

Values of expressions are independent of their locations in the source code, that is, the same expression refers to the same value everywhere in the program text. Function applications do not have side effects, that is, evaluating a function does not change the value of an expression. Therefore, *variables*[11] of a purely functional program are, as a matter of fact, constants. Values of variables – like in mathematics – may not be known in advance, but they are unambiguous and cannot change during the program execution. This property plays an important role in verifying correctness of functional programs using equational reasoning, for example in the code snippet below all *free occurrences* of `x` can be textually replaced by the `f a` expression in the scope of `where` and the value of `f a` remains always the same.

```
... x + x ...
  where x = f a
```

Strong static typing

Although it is not mandatory to use type declarations, it is required for every expression to have a type determined by the *type inference rules* of the Hindley-Milner restricted polymorphic type system. This means that *the most generic type* of a given expression can be inferred by the compiler using the types of the contained subexpressions. This is also possible even if the author of the program did not declare it. There are also program constructs provided for describing abstract algebraic data types.

Higher-order functions

Functions are treated as values like elements of sets of primitive types. Functions are considered to be higher-order when any of their arguments or their return value is a function. Higher-order functions heavily contribute to *modularization* of programs and to deepening the *functional abstraction*. For example: `twice f x = f ( f x )`

Application of higher-order functions also influences the process of computation. For example, evaluation of a function may happen earlier in the case of strict evaluation if it appears as an argument to a higher-order function. The process of computation can be clearly divided into sections by using higher-order functions.

[11] For example formal parameters used in a function definition.

Currying, partial function application

There is no need for *multiparameter functions* when functions can return functions. Haskell B. Curry proposed to consider all functions only with a single parameter. If a function has multiple parameters, Curry's proposal suggests applying it to the first argument only. This results in another function which can be then applied to the next argument and so on. Thus application of a multiparameter function is implemented as subsequent applications of single-parameter functions. For example, the operation of addition can be taken as a single-parameter function where the first operand of addition is considered the only argument of the `+` operation and the result is a further function. That is, the `(+) 1` function is the same as the `inc` function. A multiparameter function is *partially applied* if it returns a function after some of its arguments have been specified from the left to the right.

Recursive function application

In order to express loops in functional languages, one must use recursion. This is usually the only way to describe an iterative computation in a functional language because loops assume the presence of destructive updates. Basically recursion means that the body of the function refers to the function itself so that the body becomes the body of a loop. Each time the function may be invoked with a different set of parameters, which in this way represent the loop variables. The recursion halts when the function stops calling itself. This can be implemented by branching on the variables to create a primitive case where the result does not rely on recursive computations any more. Hence it is possible to define *recursive* and *mutually recursive* functions. Some languages also employ a so-called *tail-call elimination* (or *tail-call optimization*) which allows compilers to abandon using a stack for storing previous values of variables. As a result, recursions may be computed in constant space. If the recursive call is placed at the tail of the function body, it is sufficient to return only the finally computed value, while the same value would be passed back between the consequent recursive calls.

Lazy evaluation – Eager, strict evaluation

Semantics of expressions is determined by the lazy evaluation strategy in modern, purely functional languages. Arguments of functions are evaluated only if their values are indeed required reaching the normal form. The method of lazy reduction is applied in Miranda, Clean, and Haskell. It is called strict evaluation when arguments are always evaluated before applying the associated function. Such method of evaluation is used in LISP, SML, and Hope. Note that by adding annotations, it is possible (and sometimes recommended) to use strictly evaluated elements in lazy languages and vice versa.

List comprehensions

List comprehensions are used to specify elements of iterative data structures and their ordering. They correspond to the notation used in mathematics describing elements of a set, called the Zermelo-Fraenkel set expressions. Infinite data structures (lists, sets, sequences, vectors, etc.) are lazily evaluated in such cases.

```
[ x * x | x <- [1..], odd x ]
```

The expression above defines an infinite list where squares of odd natural numbers (`[1,3,5 ..]`) are enumerated. Termination of the whole program depends on the demands of the function receiving the given list as an argument (modularity), and the list itself is evaluated lazily, that is, it is gradually unfolded until more elements are needed.

Pattern matching on arguments

There can be patterns used for the formal parameters of functions in definitions.[12] If the actual parameter matches the given pattern, the value of the function is computed by the associated function body alternative (for the exact rule, see Section 15.2.3). Consider the following example:

```
fac 0         = 1
fac n | n > 0 = n * fac (n - 1)
```

The first pattern in the definition of `fac` is `0`. If the actual parameter matches the `0` pattern (it equals zero), the value of the function is `1`.

Off-side rule

A group of related expressions can be identified and the *scope* of declarations can be limited by changing the level of indentation. The so-called off-side rule is a language construct introduced to build up a block-like structure for programs, as proposed by Peter J. Landin. The scope of declarations are local to the preceding ones if they are indented by one level further. Every language reference specifies how the off-side rule is to be applied.[13] Scopes may be nested by indenting declarations further to the right. In the following example, the `add4` function contains a local `succ` function which is different from the `add` function used in `succ`:

```
add4 = twice succ
  where
    succ x = x + 2
```

[12] Apart from certain exceptions, only constructors specified in the algebraic type definition of the formal parameter in question can be used, cf. Section 15.4.2.

[13] It is hard to see the scope of declarations when characters are displayed with variable width. Scopes of declarations may also change when length of identifiers is changed as a result of editing the program.

```
add = ... succ
```

Modelling I/O

Due to their purity and the lack of destructive updates, interaction with the real world may be challenging for functional-language programs. In imperative languages, input/output operations, for example, writing to files or reading from the standard input, are typically implemented as subprograms with implicit side effects: their types do not imply if they will do anything apart from the described relation, nor the type system can enforce that they will not indeed do. This deficiency clearly works against any effort required to reason about the correctness of programs. Hence modern functional languages are equipped with some I/O model. Some examples for this model are the *IO monad*, the *single-referenced, unique environment*, the *stream of request, stream of response*, or its semantic equivalent, the *continuation* model (see Section 15.7).

15.1.4 Brief overview of functional languages

Between 1956 and 1962, John McCarthy created the first language that used λ-calculus as the model of computation, called LISP (as LISt Processing) at MIT. Many LISP variants have come to life since then, including Common LISP (DARPA, 1981) which mixes procedural and object-oriented elements (CLOS, Common Lisp Object System), or Scheme (Steele, Sussman, 1975) which is mostly used at universities and in CAD systems. The first typed functional language is ML (Meta Language) which was originally designed as a meta language for the LCF (Logic for Computable Functions) automated theorem prover developed in Edinburgh. It was designed by Robin Milner in the middle of 1970s. After the creation of Hope (Burstall, 1980), Milner, Tofte, and Harper defined SML (Standard ML) between 1983 and 1990. The latest revised standard of SML was published in 1997 ([MTHM97] and [Har01]). Further ML variants are Caml (INRIA, 1984–1990, language of the Coq automated theorem prover) and Objective Caml (OCaml for short, successor to Caml Light, INRIA, 1990–). These languages are not purely functional, they contain imperative language constructs (for example, variables with destructive updates). The first lazy pure functional language was ISWIM ("If You See What I Mean", Landin, 1966) which also introduced the off-side rule. David Turner designed many languages influenced by ISWIM, such as the similarly lazily evaluated SASL (Single Assignment Language, Turner, 1981), KRC (Kent Recursive Calculator, Turner, 181), and Miranda ([Tur90], [Tur86] and [CMP95]) in 1985–1985. Miranda is a commercial product, copyrighted by Research Software Ltd. Later, Haskell (1990) ([PH99], [HFP99] and [Tho99]) utilized many elements of Miranda.

Haskell was born in 1987 at a functional programming conference (FPCA'87), and it was named after Haskell Brooks Curry. Its latest standard is Haskell 98

which is about to be superseded by Haskell Prime but the latter is not finished yet. The design team of Haskell incorporates many researchers from many universities all around the world (John Hughes, Simon Peyton Jones, Paul Hudak, Kevin Hammond, Eric Meijer, John Peterson, Philip Wadler, Simon Marlow et al).

The following requirements were set for Haskell:

- It must be equally suitable for teaching, research and development of large-scale applications;
- It must have a formal syntax and semantics;
- It must be freely available;
- It must comply with the widely adopted basic principles;
- It must provide a standardized line of development for modern functional languages with minor and major differences.

Since then, Haskell has become a de-facto standard in contemporary functional programming and unifies many research efforts, especially in the areas of type systems and advanced compilation techniques. The bleeding edge of its features is demonstrated by the Glasgow Haskell Compiler. As of version 7.6, GHC has been abundantly supplementing the original Haskell 98 standard with many worthy extensions, for example Generalized Algebraic Data Types, multi-parameter type classes, functional dependencies, type families, templating, type operators, generalized (SQL-like) and monadic comprehensions, Unicode syntax, view patterns, safe module imports, and it has been ported to many operating systems and computer architectures. Functional programs can be easily built with Haskell by exploiting its package distribution framework, called Haskell Cabal and the associated public database, HackageDB. There is also a careful selection of basic tools and libraries recommended for development, the Haskell Platform.

Concurrent Clean (Plasmeijer, Nijmegen, 1987, [PE01] and [Pla99]) evolved from an experimental graph rewriting system (called LEAN). It is a purely functional language with lazy evaluation. The syntax of the latest version (Clean 2.4) is close to Haskell, although it contains many language constructs that are not featured in Haskell 98. However, recently there has been experimental support for the Clean compiler to accept Haskell 98 sources[Gro10], together with an extended version, Haskell* that adds features of Clean 2.1 to the Haskell standard.

Function composition is an associative operation, therefore it can be evaluated in parallel. Normal form of expressions, if it exists at all, is independent of the evaluation strategy used (at least in confluent rewriting systems), meaning programs written in a functional language are easy to parallelize. Most of the languages feature a variant where further language constructs are added to help the programmer to annotate which subexpressions should be evaluated in parallel or in a distributed manner, for example Concurrent Clean [Kes96], Eden (Haskell), Concurrent ML, JoCaml [FFMS01].

15.2 Simple functional programs

15.2.1 Definition of simple functions

A function definition consists of one or more equations with the name of the function and its *formal parameters* on the left-hand side and the corresponding expression, or *function body* determining the function value and the method of computation on the right-hand side. This *expression* may be a single value, a formal parameter or a *function application* on its actual parameter (which may be a formal parameter to the defined function). Function application has the highest *precedence* in evaluation of expressions. It is also possible to set *guards* and *patterns* on the arguments that specify how actual and formal parameters should be *matched*. Function declarations may include the domain and the range of the given function, which however may be omitted in simple cases.

Variable names are introduced for describing formal parameters on definition of functions. *Scope* of the variable is the equation introducing the given variable. Function declarations are visible within the containing module by default, but it may be visible within the whole program in certain cases. Scope of declarations may be restricted by application of the *off-side rule*, or keywords introducing local declarations (`where`, `local`, `let`, `#` etc.), or by exporting and importing from one module to the other.

Recursive functions may be given too. For efficiency, it is advantageous to have only a single self-reference at the end of the body of the function (a "tail call"). *Case distinction* must be used to define base cases, otherwise the evaluation will not end (Figure 15.3).

```
fact n = if n == 0 then 1 else n * fact (n - 1)        -- Haskell
fact n = cond (n = 0) 1 (n * fact (n - 1))             || Miranda
fact n = if (n == 0) 1 (n * fact (n - 1))              // Clean
fun fact n = if n = 0 then 1 else n * fact (n - 1)     (* SML *)
```

Figure 15.3: Recursive function defined with case distinction.

15.2.2 Guards

Case distinction may also be implemented in a more mathematical way, using *guards*. Guards are checked by the run-time system in the order they were given when the containing function is applied, and the equation associated with the first guard evaluating to true will be processed. If no such equation exists,[14] the evaluation will stop with some error message or an exception handler will be activated (see Section 15.8). Hence the order of guards determines the semantics:

[14] This is called a partial function.

the meaning of the function may change as a result of changing the order of guards. For example in Clean and Haskell see Figure 15.4.

```
fact n | n == 0 = 1
       | n  > 0 = n * fact (n - 1)
```

Figure 15.4: *(Clean and Haskell):* Recursive function with guards.

For example in SML see Figure 15.5.

15.2.3 Pattern matching

Pattern matching is used to implement distinction of cases. For example in Miranda, Clean and Haskell see Figure 15.6.

For example in SML see Figure 15.7.

For pattern matching, the run-time system checks that the value or structure of the actual parameter matches the pattern specified on the left-hand side of the corresponding equation. The result of pattern matching is determined by the patterns and the order of equations containing them. Changing this order may change the semantics of the definition as the run-time system selects the first matching body for evaluation. If no such body exists, it will stop with an error message.[15] Patterns can be replaced by a *series of guards* but not the other way around. That is, pattern matching is not required to create a complete functional language, but it greatly contributes to the simplification of function definitions and improves the readability of the source code. However, care should be taken

[15] Here evaluation of pattern matching is simpler and more efficient than one could find in logic programming languages because only the constructors mentioned in the algebraic type definition can be used in patterns (with some minor exceptions), see Section 15.4.2.

```
gcd a b = gcd (a - b) b, if a > b
        = gcd a (b - a), if a < b
        = a            , otherwise
```

Figure 15.5: *(SML):* Recursive function with guards.

```
fact 0 = 1
fact n = n * fact (n - 1)
```

Figure 15.6: *(Miranda, Clean, and Haskell):* Pattern matching.

```
fun fact 0 = 1
      |  n = n * fact (n - 1)
```

Figure 15.7: *(SML):* Pattern matching.

because modifying a pattern affects the meaning of the succeeding ones as well. Note that pattern matching and guards can be combined (see Figure 15.8).

```
fact 0         = 1
fact n | n > 0 = n * fact (n - 1)
```

Figure 15.8: *(Clean and Haskell):* Combination of pattern matching and guards.

15.3 Function types, higher-order functions

Function types are defined by their domain and range. Domain and range are separated by an `->` (arrow) symbol, or in SML, domains can be assigned to variables in parenthesis separated by a colon, followed by range after a colon, as shown in Figure 15.9.

```
twice :: num -> num             || Miranda
twice :: Num a => a -> a        -- Haskell
fact  :: !Int -> Int            // Clean
val fact : int -> int           (* SML *)
fun fact (x : int) : int        (* SML *)
```

Figure 15.9: Function types.

The `num` is the common identifier of numeric types in Miranda. Haskell differentiates between numeric types, but it also provides a way to define a single function for all `a` types in `Num` *class* of numeric types. Type classes will be discussed in detail later, see Section 15.4.1.

Every function has at most a single argument in functional languages and λ-calculus. *Multiparameter functions* can be defined by using *higher-order functions*, following the method named after Curry (Schönfinkel, Curry, Feys). This is presented in Figure 15.10. *First-order* functions do not have functions either as parameters or as values.

```
plus :: num -> num -> num       || Miranda
plus :: Num a => a -> a -> a    -- Haskell
plus :: Int Int -> Int          // Clean
plus a b = a + b

val plus : int -> int -> int =  (* SML *)
     fn a => fn b => a + b      (* SML *)
```

Figure 15.10: Type of a multiparameter function.

The only argument of `plus` is `a` and the result is a function which adds the actual parameter of `a` to the number which is the only argument of the new function, identified by `b`. Thus `plus :: Int -> Int -> Int` has the same meaning as `plus :: Int -> (Int -> Int)`. `plus 5` is a valid function application where the result is a function of type `Int -> Int`. `plus 5 6` equals to `(plus 5) 6`. So type definition of multiparameter functions is right-*associative* while their application is left-associative: `f :: a -> b -> c`. That is, `f` is a function whose argument is of type `a` and whose result is a function with a domain of type `b` and a range of type `c`. According to this, the function application of `f x y` equals to `(f x) y`.

With Curry's method, it is easy to derive variants from multiparameter functions where only a few of the arguments are bound (partial application). Such a function is `inc` in Figure 15.11.

```
inc :: num -> num                    || Miranda
inc :: Num a => a -> a               -- Haskell
inc :: Int -> Int                    // Clean
inc x = (+) 1 x                      // Miranda, Clean, Haskell
inc = (+) 1                          // Clean, Haskell

fun inc (x : int) : int = plus 1 x   (* SML *)
val inc : int -> int = secl 1 op+    (* SML *)
```

Figure 15.11: Partially applied function.

The symbol of addition (`+`) is usually applied as an infix operation, that is, it is placed between the two operands. The definition of `plus` – as function definitions usually – was given for prefix application. There is the possibility to apply infix functions as prefix ones and vice versa in most of the languages, for example: `(+) 2 3` (Clean, Haskell, Miranda), `2 ‘plus‘ 3` (Haskell), `2 $plus 3` (Miranda), `op+ (2,3)` (SML). Definitions of `inc` like `(+) 1` or `secl 1 op+`[16] rely on this particular feature, where the first argument of the infix binary `+` operation is bound by using its prefix form.

Note that not only the result of a function application may be a function, but any of its arguments as well. This is demonstrated in Figure 15.12. Parentheses cannot be omitted in the function type definitions in this example. Omitting parentheses would imply the application of the associativity rule, which would result in a completely different definition. Parentheses must be used to denote that the given argument is not a number but a function.

[16] `fun secl x f y = f (x,y)` is used for binding the leftmost argument of `f` in SML.

```
atZero :: (num -> num) -> num          || Miranda
atZero :: Num a => (a -> a) -> a       -- Haskell
atZero :: (Int -> Int) -> Int          // Clean
atZero f = f 0

fun atZero (f : int -> int) : int
           = f 0                       (* SML *)

atZero inc
```

Figure 15.12: Function as argument.

15.3.1 Simple type constructions

Cartesian product and iterated *type construction* are available in functional languages as well. *Tuples*, finite and infinite *sequences* can be created.[17] Tuples are enclosed by brackets `()`, and sequences are enclosed by square brackets `[]`.

Tuples

Elements of tuples can be accessed by pattern matching or by predefined selector functions (such as `fst`, `snd`, `#i`). This is shown in Figures 15.13 and 15.14, respectively.

```
gcd :: (a,a) -> a | - , < , == a      // Clean
gcd :: (Num a, Ord a) => (a,a) -> a   -- Haskell
gcd (a,b) | a > b  = gcd (a - b, b)
          | b > a  = gcd (a, b - a)
          | a == b = a
```

Figure 15.13: Greatest common divisor of a tuple.

Zero-tuple has a special meaning among tuples that does not have any element. The `()` value is the only value of the `unit` type in SML.

```
gcd : int * int -> int
fun gcd (0,b) = b
  | gcd (a,b) = gcd (a mod b, b);
```

Figure 15.14: *(SML):* Greatest common divisor of a tuple.

[17] With regard to their implementation, sequences are discussed as *lists* in the literature of functional programming languages. It is possible to have a list with elements of different types in languages that are not statically typed. However, this is not allowed by either Miranda, Clean, Haskell or SML.

Sequences

The `:` keyword and the `::` *constructor* can be used to compose a sequence from an element and an existing sequence where the element becomes the leftmost. Constructors can be used in patterns so it can be easily decided by pattern matching whether the sequence is empty (`[]`) or not (`[x:xs]`), so as to extract the first element (`x`) and the remainder (`xs`). For an application of this technique, see Figure 15.15.

```
fun sum (x::xs) = x + sum xs
  | sum []      = 0;                (* SML *)

sum []     = 0
sum (x:xs) = x + sum xs             -- Haskell
sum [x:xs] = x + sum xs             // Clean

sum [1,2,3,4,5,6,7,8,9,10]
sum [1..10]                         -- Haskell, Clean
```

Figure 15.15: Sum of a sequence.

Figure 15.16 shows the representation of a sequence as a linked list. Elements of the list are stored independently, the list itself containing only references to those elements in its spine.

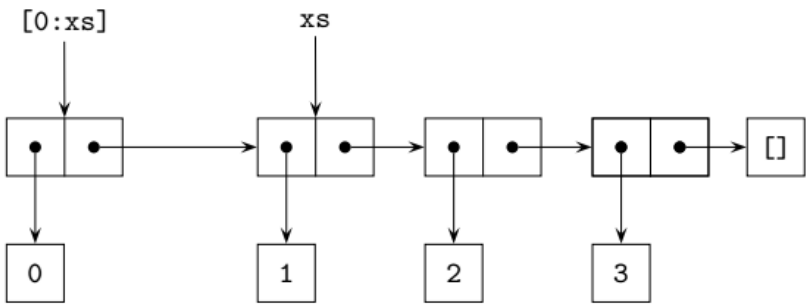


Figure 15.16: Internal representation of lists.

Some of the common list functions are defined in Clean in Figure 15.17. The `hd` returns the first element, `last` returns the last element of the sequence, and `tl` returns the subsequence of the original sequence without the first element. Functions do not have side effects in purely functional languages, so the subsequence of a sequence without the last element (`init`) is computed by building a new spine, while preserving the old one. The spine of the original list cannot be

modified. Thus, the reference to the last element cannot be simply detached[18] as it would violate the referential transparency.

```
hd [a:x]  = a                    tl [a:x] = x
hd []     = abort "hd of []"     tl []    = abort "tl of []"

last [a]    = a                  init []     = []
last [a:tl] = last tl            init [x]    = []
last []     = abort "last of []" init [x:xs] = [x: init xs]
```

Figure 15.17: *(Clean):* Implementation of basic list functions.

The `map` function can be used to process elements of a list. Argument of `map` is the function to be applied element-wise. In Figure 15.18, the `modseq` function is defined in terms of `map`: it computes a new list of numbers from an old list using a number and a binary operation. Note that it is allowed to omit the last argument, the list, from both sides of the definition (due to the so-called η-conversion rule from the λ-calculus).

```
modseq :: (num -> num -> num) -> num -> [num] -> [num]    || Miranda
modseq :: Num a => (a -> a -> a) -> a -> [a] -> [a]       -- Haskell
modseq :: (Int Int -> Int) Int -> ([Int] -> [Int])          // Clean
modseq f c = map (f c)

modseq : (int -> int -> int) -> int -> int list -> int list;(* SML *)
fun modseq f c = map (f c);                                  (* SML *)

modseq plus 5 [1,2,3]  // = [6,7,8]
modseq plus 5 []       // = []
```

Figure 15.18: Function as argument.

Further sequences can be *generated* by combining sequences and logical expressions, called *filters*. In Figure 15.19, divisors of number `n` are calculated by generating all the numbers from `1` to `n` using the `i <- [1..n]` generator, followed by the application of the `n mod i == 0` logical expression to filter out the elements which give `0` remainder when `n` is divided by them.

```
divisors n = [ i | i <- [1..n]; n mod i = 0 ]        || Miranda
divisors n = [ i | i <- [1..n], n `mod` i == 0 ]     -- Haskell
divisors n = [ i \\ i <- [1..n] | n mod i == 0 ]     // Clean
```

Figure 15.19: Divisors of a number.

[18] It is not possible to access the pointers used for the list's internal representation anyway.

Finite and infinite arithmetic sequences can be generated by the `..` symbol (*dot-dot notation*), where the difference is set by the values of the first two elements. If no second element is specified, the difference becomes `1`, and if the upper bound is omitted, the sequence becomes unbound, that is, infinite.

Therefore lazy evaluation offers *replacing recursive calls* with infinite sequences. An example of this is demonstrated in Figure 15.20. The `map` function applies its first argument as function to each of the elements of its second argument as a sequence. Thus `flist` contains all the Fibonacci numbers assigned to natural numbers starting from `0`. So the `n`$^{\text{th}}$ Fibonacci number can be easily calculated as a sum of the `n - 1`$^{\text{th}}$ and `n - 2`$^{\text{th}}$ elements of the `flist` sequence. Of course, as a consequence of lazy evaluation, not every element of `flist` is computed, only the ones that are needed.

```
fib :: Int -> Int   -- Miranda: num->num
fib 0     = 1       -- optional
fib 1     = 1
fib 2     = 2       -- Miranda: ! instead of !!
fib n = flist !! (n - 2) + flist !! (n - 1)
  where flist = map fib [0..]
```

Figure 15.20: *(Miranda, Clean, and Haskell):* Fibonacci numbers.

Finally, there are also multiple generators for generating elements in a nested or parallel fashion, as presented in Figure 15.21.

```
[ (x,y) \\ x <- [1..4], y <- [1..x] | x + y > 3 ]   // nested
[ (x,y) \\ x <- [1..4] & y <- [6..8] ]              // parallel
```

Figure 15.21: *(Clean):* Generators.

Records

Besides tuples in Clean and SML, records with fields may also be used. It becomes cumbersome to access elements of tuples with pattern matching when the number of the contained elements is large. Hence, the source code is more readable when the elements of the Cartesian product are identified by dedicated selector functions. Record types are very similar – both in term syntactical forms and patterns – to those of Clean and SML. So examples written in Clean will be shown only (Figure 15.22).

In the record pattern (Figure 15.23), it is sufficient to mention only the fields utilized in the function body.

The `'.'` (dot) symbol is used to denote selector functions in Clean (Figure 15.24), similarly to other languages, while record elements are accessed by

```
:: Point = { x       :: Real
           , y       :: Real
           , visible :: Bool
           }

:: Vector = { dx :: Real
            , dy :: Real
            }

Origo :: Point
Origo = { x = 0.0
        , y = 0.0
        , visible = True
        }
```

Figure 15.22: *(Clean):* Record.

```
IsVisible :: Point -> Bool
IsVisible { visible = True } = True
IsVisible _                  = False
```

Figure 15.23: *(Clean):* Pattern matching, record pattern.

functions of form `#<field name>` in SML. Other records can be referenced in definition of record-type values in Clean, as shown in Figure 15.25 for example. The `hide p` is a record whose fields with all their values are the same as of `p` *but* the `visible` field following the & symbol. The `Move` function will *not* displace `p` in the two-dimensional space by the translation vector `v` but derive a *new* point from `p` where all the fields have the same values *except* the `x` and `y` coordinates.

```
xcoordinate :: Point -> Real
xcoordinate p = p.x
```

Figure 15.24: *(Clean):* Accessing field of a record.

```
hide :: Point -> Point
hide p = { p & visible = False }

Move :: Point Vector -> Vector
Move p v = { p & x = p.x + v.dx, y = p.y + v.dy }
```

Figure 15.25: *(Clean):* Record update.

Arrays

Sequence elements in a linked list can be accessed only by traversing the list until the given element is reached. On the contrary, as address arithmetics is employed for arrays by the run-time system, all elements can be accessed in constant time. Clean and SML – as well as certain extensions of Haskell – feature this language construct.

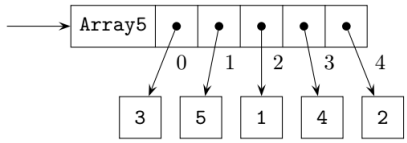


Figure 15.26: A boxed array

Arrays are considered imperative constructs in SML, values of elements may be updated. In Clean, only elements of *uniquely referenced* arrays may be updated (for uniqueness, see Section 15.6.1). The referential transparency is not violated in such cases. Uniqueness is denoted by the `*` (star) symbol in the array's type. Arrays with destructive updates are discussed in Section 15.6.

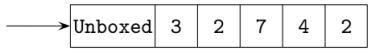


Figure 15.27: An unboxed array

Clean differentiates between *unboxed* (Figure 15.29) and *boxed* arrays (Figure 15.28) depending on whether the elements are stored in the spine or, similar to lists, the spine only maintains references to the elements.

```
Array5 :: *{Int}
Array5 = { 3, 5, 1, 4, 2 }
```

Figure 15.28: *(Clean):* Boxed array.

There is a `#` (hashmark) symbol in the type definition of unboxed arrays. Array elements can be accessed in imperative fashion.

```
Unboxed :: {#Int}
Unboxed = { 3, 2, 7, 4, 2 }
```

Figure 15.29: *(Clean):* Unboxed array.

The first element of the array is assigned to the `0` index value. This is summarized in Figure 15.30.

Arrays may also be defined by generators in Clean as it is shown in Figure 15.31.

```
Array5.[1] + Unboxed.[0]
```

Figure 15.30: *(Clean):* Accessing elements of an array.

```
narray = { e \\ e <- [1,2,3] }
nlist  = [ e \\ e <-: Array5 ]
```

Figure 15.31: *(Clean):* Array comprehensions.

In the first example, elements of the `[1,2,3]` list are read via a `<-` list comprehension and placed to the `narrray` array. In the second example, elements of the `Array5` array are read via a `<-:` array comprehension and placed to the `nlist` list.

15.3.2 Local declarations

In Miranda (Figure 15.32), Clean (Figure 15.34) and Haskell (Figure 15.33), *local* functions can be introduced by the `where` keyword. *Scope* of `where` is determined by the *off-side rule*. In SML (Figure 15.35), local declarations can be assigned to declarations using the `local ... in ... end` construct, and local declarations can be assigned to expressions using the `let ... in .. end` construct. The `let` expressions can be used in both Haskell and Clean. There are many different syntactical forms for `let` expressions in Clean with different meanings. For example, the `#` (hashmark) symbol introduces a static local declaration for a function body. Formal parameters of the primary function can be directly referenced in the local declarations introduced by the `where` keyword.

```
quadratic :: num -> num -> num -> [num]
quadratic a b c = error "not quadratic"           , if a = 0
                = error "complex roots"           , if delta < 0
                = [ -b / (2 * a) ]                , if delta = 0
                = [ -b / 2 * a + radix / (2 * a)
                  , -b / 2 * a + radix / (2 * a) ], otherwise
                where
                delta = b * b - 4 * a * c
                radix = sqrt delta
```

Figure 15.32: *(Miranda):* Local declarations.

15.3.3 An interesting example: queens on the chessboard

Place `n` chess queens on an `n` by `n` chessboard so that no two queens attack each other. Two queens attack each other if they share the same column, row,

```
quadratic :: Float -> Float -> Float -> (String,[Float])
quadratic a b c
  | a == 0      = ("non quadratic", [])
  | delta <  0  = ("complex roots", [])
  | delta == 0  = ("one root", [ -b / (2 * a) ])
  | delta >  0  = ("two roots",
                   [ -b / (2 * a) + radix / (2 * a)
                   , -b / (2 * a) - radix / (2 * a) ])
  where
    delta = b * b - 4.0 * a * c
    radix = sqrt delta
```

Figure 15.33: *(Haskell):* Local declarations.

```
quadratic :: Real Real Real -> (String,[Real])
quadratic a b c
  | a == 0.0     = ("non quadratic", [])
  | delta < 0.0  = ("complex roots", [])
  | delta == 0.0 = ("one root",[ ~b / (2.0 * a) ])
  | delta > 0.0  = ("two roots", [ (~b + radix) / (2.0 * a)
                                 , (~b - radix) / (2.0 * a) ])
  where
    delta = b * b - 4.0 * a * c
    radix = sqrt delta
```

Figure 15.34: *(Clean):* Local declarations.

```
fun quadratic a b c =
   let
      val delta  = b * b - 4.0 * a * c;
      val radix  = Math.sqrt delta;
   in
      if  a = 0.0           then ("non quadratic", [])
      else if  delta  < 0.0 then ("complex roots", [])
      else if  delta  = 0.0 then ("one root", [ ~b / (2.0 * a)])
      else  ("two roots", [ (~b + radix) / (2.0 * a)
                          , (~b - radix) / (2.0 * a)])
   end
```

Figure 15.35: *(SML):* Local declarations.

or diagonal. Find all the solutions (see one in Figure 15.36).

Solutions are given as sequences of sequences with `n` elements. Each solution describes in which row the queens should be placed in the columns from the left to the right. No column should contain two queens.

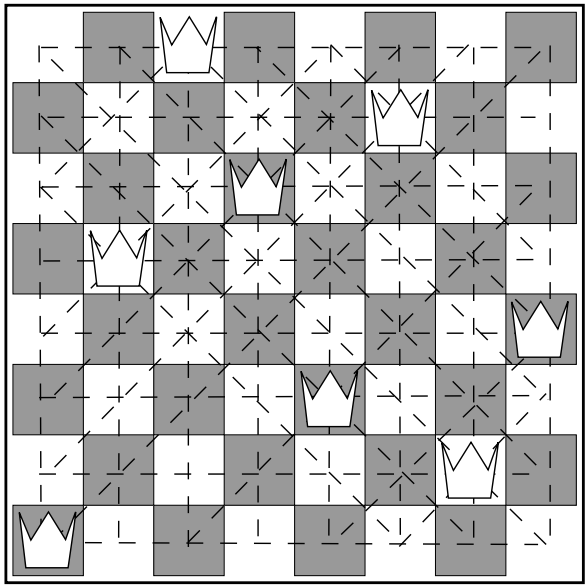


Figure 15.36: A possible solution of the eight queens puzzle.

An implementation in SML is given in Figure 15.37, where solutions are searched in permutations of the `List.tabulate(n, fn x => x + 1)`[19] sequence, that is, in sequences of form `[1,2,...,n]`, meaning no row can contain two queens. Thus, the only constraint to be checked for those permutations is whether two queens share the same diagonal (`diag`). Valid permutations are collected by the `accuqueens` function, and the next permutation is generated by `cycle`. The functions `accuqueens` and `cycle` are mutually recursive so they are connected by the `and` keyword. The `@` symbol concatenates two sequences, while the `(right as r::rr)` *as-pattern* enables referencing to the whole sequence as `right` and its first element as `r`, and the remainder as `rr`.

The program in Figure 15.38 gradually extends the solution from left the right, moving from one column to the other so that the newly placed queen does not attack any of the previously placed ones. Note that this approach results in a succinct and very readable source code when generators are used.

The Haskell implementation is listed in Figure 15.38.

The `queens` function searches for safe places in the succeeding column. `queens n m` denotes all the safe spots in the last `m` columns where `b` is a valid placement. The logical expression `safe q b` is satisfied when the queen placed to row `b` does not attack any of those that are in `b`.

```
safe :: Int -> [Int] -> Bool
safe q b = and [ not (checks q b i) | i <- [0 .. length b - 1] ]
```

[19] The `tabulate` function creates a list of n elements, where the values are calculated by individually applying the given function on elements of the [0 .. n - 1] sequence.

```
local
 fun diag' d1 d2 []       = true
   |  diag' d1 d2 (q1::qr) =
      d1 <> q1 andalso d2 <> q1 andalso
     diag' (d1 + 1) (d2 - 1) qr

 fun diag mid right = diag' (mid + 1) (mid - 1) right

 fun accuqueens []      tail res = tail :: res
   | accuqueens (x::xr) tail res = cycle [] x xr tail res
 and cycle left mid [] tail res =
          if diag mid tail then
             accuqueens left (mid :: tail) res else res
   | cycle left mid (right as r::rr) tail res =
          cycle (mid::left) r rr tail
               (if diag mid tail then
             accuqueens (left@right) (mid :: tail) res
               else res)
in
    fun queens n = accuqueens (List.tabulate(n, fn x => x + 1)) [] []
end
```

Figure 15.37: *(SML):* Solver for the eight queens problem.

```
doQueens :: Int -> [[Int]]
doQueens n = queens n n  -- n queens on an n by n board

queens :: Int -> Int -> [[Int]]
queens n 0       = [[]]
queens n (m + 1) = [ q:b | b <- queens n m, q <- [0..n-1], safe q b ]
```

Figure 15.38: *(Haskell):* Solver for the eight queens problem.

The above `safe` function checks whether the queen placed in row `q` attacks any of `b`. The `i` iterates over the column indices in `b`. The `checks q b i` is satisfied if a fresh queen of row `q` in the actual column attacks the queen of row `b` in column `i`. The `and` function is an element-wise conjuction, and it evaluates to true if all the sequence elements in its argument are true. The implementation of the `checks` function is the following:

```
checks ::  Int -> [Int] -> Int -> Bool
checks q b i = (q == (b !! i)) || (abs (q - (b !! i)) == (i + 1))
```

The expression of `b !! i` refers to the $\texttt{i}^{\text{th}}$ element of the sequence `b`, that is, the place of the $\texttt{i}^{\text{th}}$ queen. The `||` symbol is the binary logical disjunction

operation, `abs` is the absolute value function. Finally, the top-level invocation of the solver follows:

```
doQueens 8
```

Implementations in Miranda and Clean are presented in Figures 15.39 and 15.40, respectively.

```
queens n 0       = [ [] ]
queens n (m + 1) = [ q:b | b <- queens n m; q <- [0..n-1]; safe q b ]

safe q b = and [ ~checks q b i | i <- [0 .. #b - 1 ] ]

checks q b i = (q = b ! i) \/ abs(q - b ! i) = (i + 1)

queens 8 8
```

Figure 15.39: *(Miranda):* Solver for the eight queens problem.

```
queens_ n 0 = [ [] ]
queens_ n m = [ [q:b] \\ b<-queens_ n (m-1), q<-[0..n-1] | safe q b ]

safe q b = and [not (checks q b i) \\ i <- [0 .. (length b) - 1] ]
checks q b i = (q == b !! i) || (abs(q - b !! i) == (i + 1))

queens n = queens_ n n

Start = (length (queens 8), queens 8)
```

Figure 15.40: *(Clean):* Solver for the eight queens problem.

15.4 Types and classes

In this section, functional language constructs for data abstraction are presented, that is, we look at how abstract algebraic types and type classes may be defined, or how higher-order types may be applied.

15.4.1 Polymorphism, type classes

There have been many functions presented earlier that could be applied on values of many different types. These functions are called *polymorphic*. An example is the `hd` function which returns the first element of a sequence and it may be applied to an arbitrary (non-empty) list, independently of the type of the list elements. Implementation of the function does not depend on the type either,

so it can be defined in a type-independent format as well. A *type variable* is introduced in the function type definition to express that the function in question is polymorphic, that is, it may be applied to any concrete type.

```
hd :: [*] -> *   || Miranda
hd :: [a] -> a   // Clean, Haskell
hd (x:xs) = x
```

Figure 15.41: Example of a polymorphic function.

The `+` operation can be applied on values of many concrete types too, for example on integers, reals, or even on Booleans (Figure 15.42). The addition computes a new value from two values of the same type, and returns a value of the same type, and this operation is always infix. By contrast, the simple polymorphism that can be observed in the definition of `hd` in Figure 15.41, the implementation varies from type to type. For example, integers and reals have different binary representations. Hence, polymorphism works differently in this case – a common name is used for many different functions, that is, the operation symbol is *overloaded* (this is called "ad hoc" polymorphism, or overloading). Only predefined functions may be overloaded in SML. Overloaded names can be defined in both Clean and Haskell, although they are required to share the same properties (number of arguments, fixity, associativity, etc.). In such cases, there exists a common, type-independent, abstract signature for those operations. See Figure 15.43 for an example.

Description of types of function arguments and result is called function *signature*. Abstract description of the `+` operation can be composed by introducing type variables , for example as `(+) :: a a -> a` (in a simplified version). Types for each of the overloaded addition operations can be then derived by substituting the `a` type variable with a concrete type, which is called *instantiation*. These operations may have different implementation so the addition operation must be defined for each type too. In both Clean and Haskell, abstract signatures start with the `class` keyword, and the abstract type instantiations start with the `instance` keyword. Language constructs representing abstract signatures and instantiations are not classes in SML but signatures of the module language of SML, that is, structures and functors (see Section 15.5).

A `class` declaration may join many different but related abstract signatures. A set of instances defined for abstract signatures of a given `class` declaration is called a *type class*.[20] In Figure 15.42, the abstract signature defines that the operation is infix and left-associative, identified by the last letter of the `infixl`

[20] Elements of type classes are functions, different concrete versions of abstract functions. Perhaps *function class* would be a more expressive name. Anyway, it is a type class in the sense that a set of concrete types may also be defined where an instance for each abstract functions given in the `class` declaration exists.

keyword (as in ”left”). The operator precendence is also set, it is `6` for the addition.

```
class (+) infixl 6 a : !a !a -> a // abstract (+)

double :: a -> a | + a
double n :== n + n

instance + Bool where
  (+) :: !Bool !Bool -> Bool      // instance
  (+) True b = True
  (+) a    b = b
```

Figure 15.42: *(Clean):* Type class and instantiation.

The `double` is polymorphic but it does not have to be instantiated. Definition of `double` depends on the definition of `+`, and this is noted in the signature. The `double` is a *derived function* and its interpretation depends on whether an instance was defined and how it was defined for the `+` *type class*.

```
class Num a where
  (+), (-), (*)  :: a -> a -> a
  negate         :: a -> a
  abs            :: a -> a
  signum         :: a -> a
  fromInteger    :: Integer -> a
```

Figure 15.43: *(Haskell):* The numeric type class.

The `Num` type class in Haskell is given in Figure 15.43 as an example for a class with multiple signatures. The type system of Haskell and Clean distinguishes type constants from type variables: type variables start with lowercase letters, so the `a` is a type variable in this example. Type constants are for example the `Char`, `Int`, `Integer`, `Float`, `Double` and `Bool` types.

```
double :: Num a => a -> a          -- Haskell
double :: a -> a | + a             // Clean
double x = x + x
```

Figure 15.44: Class context.

Identifiers of classes usually start with uppercase letters. A *class context* may be set for type or function declarations to restrict the values of the include type variables by classes. As Figure 15.44 demonstrates, then the `double` function may be applied to values of the `Int` type as member of the `Num` type class, but not to the values of the `Char` type.

The c_1 class context restricts the a type variable to a t type which is the member of the c_1 class. The concept of *type application* corresponds to the function application. If type t_1 has form of $k_1 \rightarrow k_2$ and type t_2 matches k_1 then value of the t_1 t_2 type expression is the k_2 type (type inference). Type variables are usually implicitly universally quantified.

Type of an expression must be always inferable by the Hindley–Milner type inference algorithm. The result depends on the type context that determines the types of free variables, and the class context that restricts the type variables. Type of the polymorphic `double` function is more generic than the `Int -> Int` type, the most generic type description is of form `a a -> a | + a`. The most generic type of the expression is the principal type of the expression that could be inferred even if function names appearing in the expression are overloaded (that is, have multiple types). However, in some cases, since the compiler is not capable of inferring the most generic type, a signature must be provided. Overloading identifiers is often the cause of an unsuccessful inference. Thus, SML restricts this feature directly to avoid this problem.

Definition: *Most generic type*
Let u, v be sets of type variables, c_1, c_2 be class contexts for these variables, and t_1, t_2 be type expressions that contain such type variables. Type (expression) of $\forall u.c_1 \Rightarrow t_1$ is more generic than $\forall w.c_2 \Rightarrow t_2$ if there exists an S substitution where type variables in u are substituted in such a way that t_2 equals to $S(t_1)$ and if c_2 holds then $S(c_1)$ holds as well.

Types can be defined as algebraic data types, derived types, or as synonyms for other types. Type constructor classes will be discussed after introducing the concept of algebraic data types.

15.4.2 Algebraic data types

A new type (type construction) and its data constructors are defined at the same time as when declaring algebraic data types. All values of the given type are exclusively created by one of the data constructor functions specified at the declaration. Algebraic type definition of the enumeration type is given in Figure 15.45 below. The `Day` is a constant (nullary) type constructor, and `Mon`, `Tue`, etc. are constant data constructor functions whose values are of type `Day`.

```
:: Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Figure 15.45: *(Clean):* Enumeration type.

The `Tree` type construction presented in Figure 15.46 makes a binary tree type out of an arbitrary type. The `Tree` type constructor has a single argument, namely, the `a` type variable. Type constructors may be considered higher-order types that derive a type from another type. The `Node` data constructor function

has three arguments, where the first argument is of type `a`, and the second and third ones are of type `Tree a`. This declaration illustrates that type constructors may be just as well recursive. Constructors are visible within the whole module where they have been defined.

```
:: Tree a
   = Node a (Tree a) (Tree a)
   | Leaf

aTree = Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf)
```

Figure 15.46: *(Clean):* Recursive parametric type.

Figure 15.47 illustrates the value of `aTree`.

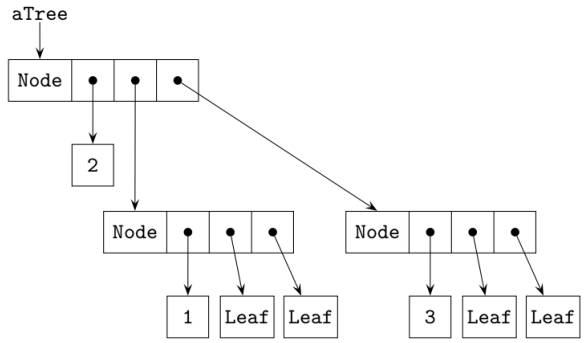


Figure 15.47: Value of type `Tree`: `aTree`.

In Miranda, algebraic data types are defined by the `::=` symbol and type variables are denoted by the `*` symbol. This is briefly demonstrated below:

```
bool         ::= True | False
nat          ::= Zero | Succ nat
list *       ::= Nil | Cons * (list *)
color       ::= Red | Orange | Yellow | Green | Blue | Indigo | Violet
either * ** ::= Left * | Right **
tree *       ::= Nilt | Node * (tree *) (tree *)
```

Figure 15.48: *(Miranda):* Algebraic data types.

In Haskell, algebraic data types are defined by the `data` keyword, in SML they are defined by the `datatype` keyword. Algebraic definition of the set type is given in Haskell together with the types of the associated data constructors below:

There can be names assigned to each of the fields in the constructors (Figure 15.49). Therefore, it is possible to define Cartesian products of data types

```
data Set a = NilSet | ConsSet a (Set a)

NilSet  :: Set a
ConsSet :: a -> Set a -> Set a

data T a b = K1 { f1 :: a, f2 :: b } ...
```

Figure 15.49: *(Haskell):* Field name assignments in the constructor.

which resemble records and allow using field names in the associated patterns. This is actually a variant record (Figure 15.50).

```
data Person
  = Male   { name :: String, age :: Int }
  | Female { name :: String }
```

Figure 15.50: *(Haskell):* Variant record definition.

In this case, even without knowing whether the actual record `x` of type `Person` is a male or female, there may be a value defined with the name `"Sam"` by a single `case` expression, as it is presented in Figure 15.51.

```
case x of
  Male _ age -> Male "Sam" age
  Female _   -> Female "Sam"
```

Figure 15.51: *(Haskell):* Matching patterns on a variant record.

Note that this syntax also comes with automatic generation of functions derived from field names. That is, for the type `Person`, the functions shown in Figure 15.52 are created. As a consequence, such ”field names” must be unique per module, otherwise they will cause a compilation error. This feature is commonly exploited in `newtype` definitions of various types (see Section 15.4.4), for example functors, monads, and so on.

```
name :: Person -> String
name (Male s _) = s
name (Female s) = s

age  :: Persont -> Int
age (Male _ x) = x
age _          = error "No match in record selector age"
```

Figure 15.52: *(Haskell):* Record selector functions.

The next example in Figure 15.53 is a possible algebraic definition of the list type in SML. Type variable identifiers are always preceded by the ' symbol and type variables are in front of the type constructor. Data constructors and their arguments are delimited by the `of` keyword.

```
datatype 'a List = Nil | Cons of 'a * 'a List
```

Figure 15.53: *(SML):* Algebraic data type.

In Miranda, abstract data types can be defined by the `abstype` keyword (Figure 15.54), while in SML, Clean and Haskell, data abstraction is implemented by the module system (see Section 15.5).

```
abstype tree *
with mirror :: (tree *) -> (tree *)
     empty  :: (tree *)

numtree == tree num
```

Figure 15.54: *(SML):* Abstract, algebraic data type.

In Figure 15.55, there are altogether two operations for the `tree` type: `empty` which is the empty tree, and `mirror` which mirrors an arbitrary tree. Algebraic type definition of the `tree` type construction is hidden, scope of the data constructors is restricted, making them unavailable at other parts of the program.

```
tree * ::= Nilt | Node * (tree *) (tree *)

mirror Nilt          = Nilt
mirror (Node a x y) = Node a (mirror y) (mirror x)
```

Figure 15.55: *(SML):* Working with recursive algebraic data types.

15.4.3 Type synonyms

Type synonym declarations (Figure 15.56, Figure 15.57) do not introduce new types – synonyms equal the original type expressions –, they are to be considered abbreviations only. Hence, recursion and instantiation are not allowed for type synonyms.

15.4.4 Derived types

In Haskell, an identically represented but different type can be obtained by deriving with the `newtype` keyword (for an example, see Figure 15.58).

```
string == [char]
matrix == [[num]]
```

Figure 15.56: *(Miranda):* Type synonyms.

```
type List          = []
type Rec a         = [Circ a]
type Circ a        = Tag [Rec a]
type PredicateOn a = a -> Bool
```

Figure 15.57: *(Haskell):* Type synonyms.

```
newtype Name   = Name String
newtype Square = S (Char,Int)
newtype F a b  = F (a -> b)
```

Figure 15.58: *(Haskell):* Derived types.

Instantiation of type classes is also allowed for derived types. However, this creates a compile-time distinction as the compiler will drop the types defined this way and will use them at the static type checking. Therefore, `newtype` definitions are restricted to a single data constructor with a single parameter, they behave like special ”tags” for simple type synonyms. So derived types to exploit the representation of an already defined type but with different interpretation. Since such `newtype` aliases are not kept at run-time, they are more efficient than introducing a whole new algebraic type.

Types derived by `newtype` are often applied to resolve problems related to types, such as the question of how to work around overlapping instances of type classes, or how to restrict operations to certain types only. This technique is also commonly used for building the so-called *smart constructors* that allows the creation of well-formed elements only or else gives a run-time error.

```
square :: Char -> Int -> Square
square c n
  | (c,n) `elem` [ (x,y) | x <- ['A'...'H'], y <- [1..8] ] = S (c,n)
  | otherwise = error "Invalid square"
```

Figure 15.59: *(Haskell):* Smart constructor.

In the example in Figure 15.59, `square` will only build a new element of type `Square` if it is a valid square of a chess board. That is, value of the first coordinate is between ’A’ and ’H’, and the value of the second coordinate is between 1 and 8. Note that this is equivalent to using a `Square = (Char,Int)` type synonym in terms of performance.

15.4.5 Type constructor classes

In Clean, it is allowed to create classes whose instances are not defined for types but for type constructions. Two examples are the `length` and the `map` functions, where the first counts the number of elements, and the second applies a function to all elements of an arbitrary data type. In the example in Figure 15.60, the class declaration contains a `t` type constructor variable which may be either a list or a tree.[21]

```
class map t :: (a -> b) (t a) -> (t b)

instance map [] where
  map f l = [ f e \\ e <- l ]

:: MTree a = NNode a [MTree a]

instance map MTree where
  map f (NNode el ls) = NNode (f el) (map (map f) ls)
```

Figure 15.60: *(Clean):* Type constructor class.

In Haskell, the `Functor` class generalizes the `map` function to type constructors, called `fmap`. The definition of this class can be seen in Figure 15.61.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Figure 15.61: *(Haskell):* The functor class.

15.5 Modules

While there is only one module type in Haskell, in Clean there may be definition and implementation modules. Definition modules are usually paired up with implementation modules so that they together form a whole. Definition module provides an interface for users of the implementation module – only those elements are visible from the module to the outside which are specified there. The pair of definition and implementation modules is the vehicle of data abstraction in Clean. Modules refer to each other using the `import` keyword, the import of modules is transitive. In Haskell, definition modules are replaced by export lists which determine the functions or data types to be available from outside the module. Export lists are placed in the module header. Every module that imports `New` of Figure 15.62 will not see `fn2` but only `fn1`. Without an export

[21] In this example, a new, non-binary tree is used.

list, everything defined in the module will be visible to the outside world. Neither of the languages allow importing anything but the elements needed from other modules.

```
module New (fn1) where

fn1 x = x
fn2 x = 3 + x
```

Figure 15.62: *(Haskell):* A module, export the function `fn1`.

Haskell and SML support using qualified names , which are useful when identifiers from other modules are used as well. It may happen that the same identifiers are defined in multiple imported modules, but with different implementation. The syntactical form of qualified names is *qualifier.identifier* where *qualifier* is usually the name of the containing module, however modules can have aliases when imported.

15.5.1 Abstract algebraic data types

In the example shown in Figure 15.63, it will be shown how an abstract data type using the module system of Clean may be implemented. Representation of the stack type is defined in the implementation module as a type synonym. This representation is not part of the definition module, that is, the list representing the stack can be only accessed from the implementation module: other modules can only see the exported abstract operations.

The module system of SML is richer, the language constructs for working with modules are described by a standalone module language [Har01]. Description of the whole module language is beyond the scope of this chapter, thus only the most important elements are discussed.

Signatures correspond to definition modules of Clean,[22] and *structures* correspond to implementation modules. However, an important difference is that SML features signature expressions and structure expressions whose values can be passed as arguments.

Signatures can be considered type description of structures. All structures have a primary, inferable signature which can be matched with a signature implementing the specification by an (preorder) equivalence relation [Har01].

There are two kinds of inheritance relations for signatures in the SML module language: a signature can extend (contain) another one, or it can be a specialization of another one (see the next example). Extensions are suitable for adding new operations, and specializations are capable of setting the represenation for

[22] Here, the word "signature" refers to the SML language construct. Signatures in SML the generalizations of function signatures. They describe interfaces of whole structures.

```
definition module Stack

:: Stack a

Push  :: a (Stack a) -> Stack a
Pop   :: (Stack a) -> Stack a
Top   :: (Stack a) ->  a
Empty :: Stack a

implementation module Stack

:: Stack a :== [a]

Push :: a (Stack a) -> Stack a
Push e s = [e:s]

Pop :: (Stack a) -> Stack a
Pop [e:s] = s

Top :: (Stack a) -> a
Top [e:s] = e

Empty :: Stack a
Empty = []

module Test
import StdEnv, Stack

Start = Top (Push 1 Empty)
```

Figure 15.63: *(Clean):* Modules and abstract data types.

certain abstract types defined as part of the signature. Signature imports are represented as inheritance between signatures, or contents of structures declared `open` automatically become visible in the module where it was imported. Structures correspond to signatures when the primary signature contains everything what is prescribed by the signature.

```
signature QUEUE =
  sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
  end
```

```
signature QUEUE_WITH_EMPTY =
  sig
    include QUEUE
    val is_empty : 'a queue -> bool
  end

signature QUEUE_AS_LISTS =
  QUEUE where type 'a queue = 'a list * 'a list
```

The ORDERED SML signature below corresponds to the Ord type class (see Section 15.4), whose possible implementation is the LessInt structure. The equivalence relation is expressed by the : notation, which means that the LessInt structure is of type ORDERED.

```
signature ORDERED =
  sig
    type t
    val lt : t * t -> bool
    val eq : t * t -> bool
  end
structure LessInt : ORDERED =
  struct
    type t = int
    val eq = (op =)
    val lt = (op <)
  end
```

Like classes can have multiple instances, signatures can also have multiple implementations at the same time. Some of them can be even nested in other structures. This is illustrated in Figure 15.64. The :> symbol prescribes in signature implementations that only those entities should be visible – when using the structure – which are present in the signature. Structures constructed this way are the vehicle of *data abstraction*. The example below demonstrates the implementation of an abstract queue type [Har01]. The queue is represented by an ordered pair of queues. The QUEUE_WITH_EMPTY signature (see the example on page 891)contained the type specification which is now assigned to the Queue structure by making its internal representation *opaque* using the :> symbol.

SML also supports parameterization of modules with structures. Parametric modules are called functors in the language which mostly resembles to templates in Ada. In the example[Har01] in Figure 15.65, it is shown how the functor DictFun equivalent to the DICT signature is parameterized by the K structure. The dict type is a type construction where key-value pairs can be searched by their key component. The DictFun structure defines the type in the signature by an algebraic data type where the search operation can be implemented efficiently. Thus it is set for the K structure parameter – describing the key type – that it must be equivalent to the ORDERED signature (see its definition on page 892).Both

```
structure Queue :> QUEUE_WITH_EMPTY =
  struct
    type 'a queue = 'a list * 'a list
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    exception Empty
    fun remove (nil, nil) = raise Empty
      | remove (bs, f::fs) = (f, (bs, fs))
      | remove (bs, nil) = remove (nil, rev bs)
  end
```

Figure 15.64: *(SML):* Modules and abstract data types.

the representation and implementation are opaque because of the `:>` symbol, so further details are not given here.

For example, the actual value of `K` can be the `LessInt` structure (see page 892)which implements the standard ordering for integers. Note that `LessInt` equals the `ORDERED` signature by keeping the inner representation visible (via the `:` symbol). Thus, it is salient that the type of the key (`K.t`) is implemented by integers.

The `LtIntDict` structure becomes an instance of the `DictFun` functor, where the type of the key is integer, while the type of the value component in the key-value pair (parameter of the `LtIntDict.dict` type constructor) remains free.

15.6 Uniqueness, monads, side effects

Input and output operations necessarily have side effects, for example, they change the program environment, the contents of the screen, the files stored in the file system, and so on. Purely functional languages also feature constructs with side effects, otherwise it would not be possible to write programs that interact with the outside world. However, side effects in such languages must be restricted to certain parts of the program and referential transparency must be maintained as well.

15.6.1 Unique variables

In certain cases, in Clean destructive updates are allowed. That is, *unique* objects may be overwritten if and only if they are referenced only once. This allows changing the value of the object in-place, that is, without copying it, because it is guaranteed that its previous value will not be accessed, and that the referential transparency will not be violated. The old instance disappears so its place can immediately be reassigned to the new one, without invoking the garbage collector. This is often effective, for example when working with larger

```
signature DICT =
  sig
    structure Key : ORDERED
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * Key.t * 'a -> 'a dict
    val lookup : 'a dict * Key.t -> 'a option
  end

functor DictFun (structure K : ORDERED) :> DICT
  where type Key.t = K.t =
struct
  structure Key : ORDERED = K
  datatype 'a dict = ...  (* representation *)
  val empty = Empty
  fun insert  ... =  ...  (* implementation *)
  fun lookup  ... =  ...
end

structure LtIntDict = DictFun (structure K = LessInt)
```

Figure 15.65: *(SML):* Functor.

data structures. If there is a need for a new version that only slightly alters from the original unique data structure, it is allowed to overwrite the corresponding element, which may therefore be implemented with a minimal overhead. The same is true for files, when the modified records are changed rather than an entirely new file is created.

In Clean, contents of arrays with unique spine can be redefined. In Figure 15.66, `mArray5` is an array whose spine is placed to the same memory address as of the unique array `Array5` and its elements are the same as of `Array5` except for the elements with indices `3` and `4`, whose new values are given after the & symbol. Every element in `mArray` will be different from the element with the same index in `mArray5`.[23]

```
Array5 :: *{Int}
mArray5 = { Array5 & [3] = 3, [4] = 4 }
mArray  = { Array5 & [i] = k \\ i <- [0..4] & k <-[80,70..] }
```

Figure 15.66: *(Clean):* Arrays.

The uniqueness of objects is checked by the compiler. A new name must be given to each fresh instance at each "assignment" but since there are no

[23] Note that after the second & symbol, `//` is followed by two parallel list comprehensions so that the generated elements are `[80, 70, 60, 50, 40]`.

more references to the old instances, the same name may be reused in the subsequent local definitions. Then the new name will refer to the new instance. This enables a style similar to imperative programming (when required) without sacrificing the purity of the language. To introduce destructive updates, it is important to determine the ordering of object assignments, otherwise it cannot be decided properly how they are referenced via the reused names. This will be demonstrated in the *Dialogue* program on page 900.

15.6.2 Monads

Monad[24] is a concept from category theory which describes algebraic properties of three operations [BW90]. This algebraic concept is implemented in Haskell to model computation with side effects. Haskell defines three monadic classes: `Functor` (15.4.5), `Monad` and `MonadPlus`. They can be instantiated by type constructions like `IO` (a higher-order type to represent I/O operations), for example (see 15.7).

There are some axioms for the abstract operations of monadic classes that should be considered for instantiations. For example, `Functor` class' `fmap` operation must satisfy the well-known laws for element-wise functions.

```
fmap id      = id
fmap (f . g) = fmap f . fmap g
```

Basic operations of the `Monad` and `MonadPlus` monadic classes implement binding monadic values or *monadic actions*.[25] When modelling I/O operations, monadic actions always have to wait for the previous one to complete. Monadic actions may act as objects. Both the arguments and the return value of the associated actions may contain an internal state which is then (implicitly) changed during the execution of the action. Thus, I/O actions pass this unique implicit internal state to each other step by step, where the state contains a collection of elements for the outside world.

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```

Figure 15.67: *(Haskell):* The `Monad` class.

The `Monad` class, presented in Figure 15.67, has two main operations. The `return` operation maps its argument to a monadic value by wrapping it, and

[24] Sometimes monads are called triples in the literature.

[25] Note that there are differences between the three basic monadic operations, monadic values and monadic *actions*.

the `>>=` operation binds two monadic values, that is, two monadic actions – of the same type – together. The `>>` is a variant of `>>=` which binds monadic actions by ignoring the result of the first argument. The `fail` operation is suitable for error handling, for example when a pattern matching fails within a monadic computation.

As for functors, there are some axioms here as well, called *monad laws* to satisfy in order to get a truly monadic construct.

```
return a >>= k           ==  k a
m >>= return             ==  m
m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
```

In category theory, monads are also functors, but this is not reflected in the Haskell implementation. That is, one has to manually define both the `Monad` and the `Functor` instances for a type. In this case, the instances should additionally satisfy the following law:

```
fmap f xs  ==  xs >>= return . f
```

The `do` syntax helps to concatenate monadic actions in a simpler, more intuitive form. However, it is often criticized that it makes programmers believe that monads are just embedded imperative blocks. The scope of `do` is determined by the off-side rule:

```
do  e1 ; e2          =  e1 >>     e2
do  p <- e1; e2      =  e1 >>= (\v -> case v of p -> e2;
                                                _ -> fail "s")
do  let p = e1 ; e2  =  let p = e1 in do e2
```

In Haskell, monads are used to represent side effects, therefore program parts with side effects can be easily identified by the type system. They are often called "programmable semicolons", because the behavior of `>>` and `>>=` operations differ in each monad. The expressive power of monads is great, meaning many language constructs can be expressed by them [HFP99]. For example, it is possible to define semantics for little imperative domain-specific languages and interpret programs written in them.

Common monads appearing in Haskell programs are the `[]` (list operations), `Maybe` (optional result), `Error` (track location and causes of errors), `Reader` (read-only variables), `State` (mutable variables), `Write` (log to a stateful buffer), `Cont` (continuations which can be interrupted and resumed), `ST` (locally-encapsulated mutable variables) and `STM` (software transactional memory). Note that they are different from the `IO` monad in that they may be "run", that is, their result can be safely obtained within the program itself.

An example of this is the different *run functions* for the `State` monad, shown in Figure 15.68. The `State` type has two type parameters, the type of the state (`s`) and the type of the result (`a`). So one can simply get the value of the monadic action by passing an initial state (of type `s`). The `runState` function computes

```
runState  :: State s a -> s -> (a, s)
evalState :: State s a -> s -> a
execState :: State s a -> s -> s
```

Figure 15.68: *(Haskell):* Run functions of the `State` monad.

both the resulting state and the result itself, `evalState` computes the result only, while `execState` computes only the final state.

Monadic effects can be combined by *monad transformer*s. A monad transformer is a type constructor which takes a monad as an argument and returns a monad as a result. Every frequently used monad type has a transformer variant with a `T` suffix, for example `StateT`:

```
type State s = StateT s Identity
```

Therefore monads are often defined as a combination of the `Identity` monad[26] and the corresponding monad transformer.

In addition, monads are characterized by dedicated type classes, for example `MonadState` which allows combining them in a single monadic block in a layered fashion:

```
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()
```

Finally, let us demonstrate the use of monads by solving a classical problem [Tho99]: *"Given an arbitrary tree, transform it to a tree of integers in which the original elements are replaced by natural numbers, starting from* 0*. The same element has to be replaced by the same number at every occurrence, and when we meet an as-yet-unvisited element we have to find a 'new' number to match it with."* An implementation with the `State` monad (and the adapted definition of the `Tree` type from Section 15.4.2) is shown in Figure 15.69.

For further demonstration of use of monads and their combinations, see `IO` in Section 15.7, `Maybe` and `Either` in Section 15.8, or `Eval`, `Par`, `STM` in Section 15.10.2.

15.6.3 Mutable variables

The `ref` keyword denotes pointers to mutable variables in SML. After evaluating the `val r = ref 0`, `val r = !r + 1` SML functions, the `r : int ref` pointer points to the `0 : int` value first. This value can then be read by the `!r` explicit reference resolution. Hence, the new value of the memory cell pointed by `r` becomes `1`.

[26] The `Identity` is a trivial monad that does not have any side effects or does not contain any elements.

```
numTree :: (Eq a) => Tree a -> Tree Int
numTree t = evalState (numberTree t) []

numberTree :: (Eq a) => Tree a -> State [a] (Tree Int)
numberTree Leaf           = return Leaf
numberTree (Node x lt rt) = do
  num <- numberNode x
  nlt <- numberTree lt
  nrt <- numberTree rt
  return (Node num nlt nrt)

numberNode :: (Eq a) => a -> State [a] Int
numberNode x = do
  table <- get
  let (newTable,pos) = nNode x table
  put newTable
  return pos

nNode :: (Eq a) => a -> [a] -> ([a],Int)
nNode x t =
  case (findIndex (== x) t) of
    Nothing -> (t ++ [x], length t)
    Just i  -> (t,i)
```

Figure 15.69: *(Haskell):* Labeling elements of a binary tree via using the `State` monad.

As it is demonstrated in Figure 15.70, arrays containing mutable elements can be handled by the `Array` module. The `array` function creates an array with fixed size and sets its initial value, the `sub` function accesses the $\texttt{i}^{\text{th}}$ element, and the `update` updates its value.

```
val m : int array = Array.array (size, initv)
   Array.sub (memopad, i)
Array.update (memopad, i, value);
```

Figure 15.70: *(SML):* Mutable array.

15.7 Interactive functional programs

Haskell hides its external state in `IO` monads. This state cannot be accessed directly by the programmer when working with I/O operations.

The `IO` monad is defined by instantiating the `Monad` class for the `IO a` type construction. The `getChar :: IO Char` and `putChar :: Char -> IO ()`

operations are both associated with the IO monad. Actually, the base type of the latter is the zero-tuple type containing the unit element. Although the base types of these two I/O opertions are different, they can be bound because both of them are of type IO. The getLine and putLine operations can be expressed by getChar and putChar. In summary, the following example illustrates how imperative-style I/O program should be written in Haskell:

```
helloWorld :: IO ()
helloWorld = putStr "Hello world!"
readTwoLines :: IO ()
readTwoLines = do
  linea <- getLine
  lineb <- getLine
  putStrLn $ reverse lineb
getInt :: IO Int
getInt = do
  line <- getLine
  return (read line :: Int)
toScreen :: IO ()
toScreen = print 5
```

The program's external state is hidden by the *World abstract type in Clean, whose values are unique. In contrast to the IO monad employed in Haskell, the programmer must explicitly refer to the external state in Clean, respecting the uniqueness property (otherwise the compiler issues an error). The type description below shows that the Start creates a new "world" from the "world" received as argument. The new external environment is generated as the result of user events, step by step. A new w world is obtained by using the stdio function,[27] where the console is an open state. In the following steps, further character messages are printed to the console, while the succeeding values of the console,[28] and the name local constant are defined by the user input at the same time. This is illustrated in Figure 15.71.

SML provides dataflow-based I/O via the modules PRIM_IO, STREAM_IO and IMPERATIVE_IO. The STREAM_IO interface buffers both input and output. The IMPERATIVE_IO module allows to dynamically redirect dataflows that are already opened. The TEXT_IO module is an instance of STREAM_IO for handling character-based I/O (Figure 15.72).

SML and Haskell do not need a standard graphical I/O library, but the Object IO library of Clean [AW00] allows to create menus, dialogue windows and windows. In the next code snippet, notice that user interfaces are described by values of algebraic types (for example Dialog). For certain user actions, for example closing a window (WindowClose), functions can be assigned (for example CloseProcess) which are then evaluated by the event handler of Object IO.

[27] The w variable did *not* get a new value, but w on the left-hand side is a fresh identifier which shadows the original value of w.

[28] The console will not get a new value, but rather a new console is created every time.

```
module helloconsole
import StdEnv
Start :: *World -> *World
Start w
  # (console,w)    = stdio w
    console        = fwrites "Enter your name:\n" console
    (name,console) = freadline console
    console        = fwrites ("Greetings " +++ name +++ "!") console
    (_,console)    = freadline console
    (ok, nw)       = fclose console w
  | not ok    = abort "error"
  | otherwise = nw
```

Figure 15.71: *(Clean):* Dialogue.

```
let
  ins = TextIO.stdIn
  outs = "output.txt"
in
  TextIO.openIn(ins);
  TextIO.openOut(outs);
  let
    line = TextIO.inputLine()
  in
    TextIO.output(outs,line);
    TextIO.output1(outs,#"\n");
    TextIO.flushOut(outs);
  end
end
```

Figure 15.72: *(SML):* Dataflow-based I/O

The interactive process assigned to the user interface is launched by creating a dialogue window and by starting to evaluate the `openDialog` function given as argument for `startIO`.

```
module HelloOIO
import StdEnv, StdIO
Start :: *World -> *World
Start world
 = startIO NDI Void (snd o openDialog undef hello) [] world
 where
   hello = Dialog "" (TextControl "Hello world!" [])
             [WindowClose (noLS closeProcess)]
```

15.8 Error handling

Error handling of Miranda and Clean is essentially based on the `error` and `abort` functions which can be used to display programmed error messages to the user before interrupting the evaluation of the program.

The `Monad` and `MonadPlus` classes of Haskell (see Section 15.6.2) contain a `fail` basic operation for implementing error handling in monads. The `IO` monad (see Section 15.7) also contains operations for handling errors. In this case, errors are derived from the built-in `IOError` abstract type. Each I/O error type can be handled via the `catch` function, and the `ioError` function can be used to pass the unhandled errors [HFP99]. Due to the lazy evaluation, exceptions in Haskell can be thrown anywhere, but only caught in the `IO` monad.

Through the features of rich type systems, it is possible to do "pure" error handling where algorithms do not require the `IO` monad[OGS08]. In many cases, instead of just throwing an exception on failure, one can employ the `Maybe` monad to return `Nothing` for errors, or else the results wrapped into a `Just` data constructor. For example, consider a "safe" version of the integer division operator (`div`) where a result is provided unless the divisor is zero:

```
safeDiv :: Integral a => a -> a -> Maybe a
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)
```

That is, the possibility of error becomes visible in the signature of the function. This can be thought of as a monadic effect, which can be conveniently applied in the construction of monadic blocks thanks to the properties of `Maybe`'s bind operator, as illustrated bellow:

```
averageDeviation :: (Integral a) => [a] -> Maybe a
averageDeviation l = do
  let average xs = (sum xs) `safeDiv` (genericLength xs)
  mean <- average l
  average [ abs (x - mean) | x <- l ]
```

The `Either` type is similar to the `Maybe` type but it can carry attached data both for an error (`Left` values) and for success (`Right` values). It can be also used as a monad, or as part of a monad stack built by monad transformers, through `ErrorT`. An example of using `Either` follows:

```
type Error a = Either String a
safeDiv :: Integral a => a -> a -> Error a
safeDiv _ 0 = Left "Division by zero"
safeDiv x y = Right (x `div` y)
```

Error handling in SML is more generic. In fact, its very similar to the implementation of Ada. There can be exceptions defined for the `exn` type[29] which can

[29] The `exn` type is extendable which means that its data constructors can be defined separately from the type constructors later.

be constant functions or constructor functions.[30] Declared errors can be thrown and the caller can answer the thrown messages by evaluating exception handler functions. In [Har01], an exception is declared for the (partial) factorial function which is thrown when the argument is negative. The `factorial_driver` function handles this exception together with other possible ones.

```
exception Factorial
local
    fun fact 0 = 1
      | fact n = n * fact (n-1)
in
    fun checked_factorial n =
        if n >= 0 then fact n
        else raise Factorial
end
fun factorial_driver () =
    let
        val input = read_integer ()
        val result = makestring (checked_factorial input)
        val _ = print result
    in
        factorial_driver ()
    end
    handle EndOfFile => print "Done.\n"
      | SyntaxError =>
        let val _ = print "Syntax error.\n" in factorial_driver ()end
      | Factorial =>
        let val _ = print "Out of range.\n" in factorial_driver ()end
```

15.9 Dynamic types

The strong static type system of Clean is extended by the `Dynamic` type ([Pil98] and [PE01]). Using the `dynamic` function, an arbitrary type[31] can be converted to a `Dynamic` type, which can be then recovered by *matching a type pattern* on the `Dynamic` value. The `sendDynamic` and `receiveDynamic` functions support sending and receiving constants and mobile code snippets ([PE01] and [HK02]). The `writeDynamic` and `readDynamic` functions are capable of writing and reading values of the `Dynamic` type.

Figure 15.73 shows how a Clean program builds a tree to store in a file; another program creates a function that counts the leaves of a tree and writes it to a file; finally, a third program reads the tree from a file and applies a function on it which is similary read from a file, and then displays the result.

[30] On throwing an exception, the data constructor of type `exn` wraps a value of a given type.
[31] Any type from the `TC` type class has a type code, and may be converted to a `Dynamic` type.

The first program builds the `tree2` tree, wraps it to a value of type `Dynamic` by the `dynamic` function, which is then written to a file using `writeDynamic`.

```
module v
import StdDynamic, StdEnv
:: Tree a = Node a (Tree a) (Tree a) | Leaf
Start world
  #! (ok, world) =
    writeDynamic (p +++ "value") DynamicDefaultOptions dt world
  | not ok       = abort "could not write dynamic"
  = (dt, world)
    where    dt = dynamic (Node 99 tree2 tree2)
           tree2 = (Node 2 (Node 1 Leaf Leaf) Leaf)
               p = "C:\\tmp\\"

module f
import StdDynamic, StdEnv
:: Tree b = Node b (Tree b) (Tree b) | Leaf
Start world
  #! (ok,world) =
    writeDynamic (p +++ "function") DynamicDefaultOptions dt world
  | not ok      = abort "could not write dynamic"
  = (dt,world)
    where    dt = dynamic count_leafs
              p = "C:\\tmp\\"
```

Figure 15.73: *(Clean):* Use of dynamic types.

In Figure 15.74, the `apply` applies the function read from a file on the tree read from a file, and displays the result. The expected types of the function and value read are checked by *matching patterns on the corresponding types.*

15.10 Concurrent, parallel and distributed programs

In this section, language constructs used for developing functional-style distributed, parallel programs in Clean, Haskell, and JoCaml will be presented and illustrated through specific examples. Languages differ in their expressive power, the abstraction level of the featured constructs may be different. This section will look at the following issues: annotations at the lowest level, evaluation strategies based on them, ways of explicit message passing, channels, high-level coordination, explicit processes, functional implementation of mobile programs, and agents. There are very different in their efficiency, applicability and implementation, and thus offering a plethora of ways for constructing concurrent, parallel, and distributed applications, all in line with the requirements of the problem to be solved.

```
module apply
import StdDynamic, StdEnv
:: Tree a = Node a (Tree a) (Tree a) | Leaf
Start world
  # (ok,f,world)  = readDynamic (p +++ "function") world
  | not ok        = abort " could not read"
  # (ok,v,world)  = readDynamic (p +++ "value") world
  | not ok        = abort " could not read"
  # result        = apply f v;
  = (result, world);
  where
    apply (f :: (Tree Int) -> Real) (v :: (Tree Int)) = f v
    apply _ _ = abort "unmatched"
    p = "C:\\tmp\\"
```

Figure 15.74: *(Clean):* Pattern matching on dynamic types.

Composition of functions is associative, therefore evaluation of functional programs can be parallelized well. The most important problem is to decide which subexpressions should be evaluated in parallel or in a distributed fashion.

There are many trends in the world of parallel and distributed functional programs: these include introducing new abstractions on the language level, parallel and distributed evaluation of functions, modifying existing evaluation strategies, linking mobile programs dynamically together, and using the TCP/IP communication protocol. Next, these language features will be illustrated with examples in Concurrent Clean ([Kes96], [PE01], [HZSP03], [SH99] and [AW00]), Haskell [Mar12], Eden [Loo12] and JoCaml[32] [FFMS01].

15.10.1 Parallel and distributed programming in Concurrent Clean

Annotations [Kes96] are one of the oldest functional language constructs used for expressing parallelism. This will be discussed first in Concurrent Clean. Annotations can be used to determine which parts of the function or the expression should be evaluated in parallel. In lazy languages, expressions are usually evaluated on demand. Parallel evaluation makes an exception to this rule. Annotations offer a *speculative way* to tell the evaluator which subexpressions ought to be started in the background, meaning their normal form is already available at the time when their values are needed. There are three types of annotations:

- `I`: Merging that enables parallel evaluation of an expression on the same processor where evaluation of the expression marked with `I` is already

[32] A variant of OCaml, which implements the join-calculus to flexible and type-checked concurrent and distributed programming.

in progress. The two evaluation process will work in parallel by time sharing;[33]

- `P`: Parallel annotation that marks parallel evaluation of the same expression on the same processor or possibly delegated to another processor;
- `P at procid`: Parallel evaluation on the given processor.

```
Nfib :: Int -> Int
Nfib n
  | n < 2  = 1
  =  {| P |} Nfib (n - 1) +  Nfib (n - 2) + 1
```

Figure 15.75: *(Clean):* Annotations.

The function in Figure 15.75 calculates Fibonacci numbers using annotations. Due to the `P` annotation, a new thread of evaluation is launched, that is, evaluation of `Nfib (n - 1)` is done in parallel with `Nfib (n - 2)`.

With the help of annotations, evaluation strategies (or just strategies for short) can be expressed by higher-order functions [THLP98], [HZSP03]. Based on two elementary evaluation methods, `'seq'` (sequential evaluation) and `'par'` (parallel evaluation), more complex evaluation methods can be constructed. For example, `parlist` might be used for writing a parallel `map` function that calculates elements of the resulting list in parallel. The individual elements of the list are then evaluated by following the strategy passed to `parlist` as `s` (Figure 15.76).

```
parmap :: (Strategy b) (a -> b) [a] -> [b]
parmap s f x = map f x 'using' parlist s
```

Figure 15.76: *(Clean):* A composed method of evaluation.

Another way of implementing parallel programming in Concurrent Clean is using channels. There are two types of sending messages: communication between two standalone programs and exchanging messages between concurrent threads in the same program [SH99].

In the first approach, two abstract channel types are applied: `(SChannel Int)` and `(RChannel Int)` which denote channels suitable for sending and receiving values of the base type `Int`. Messages are stored in a list in chronological order at the receiver side. Channels are created by the `createRChannel`[34] and `findSChannel` functions. The `lookupSChannel` function searches for an already created channel by name on the other side. Programs communicate by evaluating the `send`, `receive`, and `available` library functions. The program in

[33] The result of the evaluation will be determined by the semantics of merging.
[34] For local use only – channels are created by the `newChannel` function.

Figure 15.77 implements a producer-consumer process by using channels. The `produce` function recursively produces 10 data, and the `consume` function consumes 10 data as well.

```
// The producer application.
Start :: *World -> *World
Start w
  # (sc, w) = findSChannel "Consumer" w
  = produce sc 10 1 w

produce :: (SChannel Int) Int Int *World -> *World
produce sc i n w
  | i == 10   = w
  | otherwise = produce sc (i - 1) (n + 1) (send sc n w)

// The consumer application.
Start :: *World -> (Int, *World)
Start w
  # (maybe_rc, w) = createRChannel "Consumer" w
  = consumer maybe_rc w
where
   consumer Nothing w   = abort "already exists"
   consumer (Just rc) w = consume rc 10 0 w

consume :: (RChannel Int) Int Int *World -> (Int, *World)
consume rc i r w
  # (n, rc, w) = receive rc w
  | i == 0    = (r + n, w)
  | otherwise = consume rc (i - 1) (r + n) w
```

Figure 15.77: *(Clean):* A producer-consumer example.

In the second approach, threads with lazy evaluation can be created for the same processor by the `newThread` function. Message passing between the threads is also implemented by the `send`, `receive` and `available` functions. To illustrate this, the same producer-consumer program is presented again below, but this time the producer and consumer processes are represented by separate threads. Both thread bodies belong to the same program, but a thread may be launched on a different processor. This can be done by adding `newThreadAt pid`.

```
module ProdCons
import StdEnv, StdParallel, StdThread, StdChannel

Start w = startProcessorsW' myStart w

// Create a channel for the processor with the ‘pid‘ identifier.
// The channel connects the ‘produce‘ and ‘consume‘ functions,
```

```
// running on different threads.
myStart :: (Set ProcId) *World -> (Int, *World)
myStart pids w
  # (sc, rc, w) = newChannel w
    w           = newThreadAt pid (produce sc 10 1) w
  = newThread' (consume rc 10 0) w
where
   pid = pickFromSet pids 0

// The producer function.
produce :: (SChannel Int) Int Int *env -> *env | ThreadEnv env
produce sc i n env
  | i == 0    = env
  | otherwise = produce sc (i - 1) (n + 2) (send' n sc env)

// The consumer function.
consume :: (RChannel Int) Int Int *env -> (Int, *env) | ThreadEnv env
consume rc i r env
  # (n, rc, env) = receive' rc env
  | i == 1    = (r + n, env)
  | otherwise = consume rc (i - 1) (r + n) env
```

Channels are used in parallel functional programming based on TCP/IP ([AW00] and [THH99]) too. Any computer on the network can be identified by IP address, and they can communicate with each other by sending messages over the network. After the connection is established between the client and server, messages can be sent in the way presented in the next example:

```
// The client sends the "hello server" message through `TCP_SChannel`
clientSend :: TCP_SChannel *World -> (TCP_SChannel, *World)
clientSend sChannel world =
  send (toByteSeq "hello server") sChannel world

// The server receives the message through `TCP_SChannel`.
serverReceive :: String TCP_RChannel *World -> (TCP_RChannel, *World)
serverReceive expectedMessage rChannel world
  # (message, rChannel, world)          = receive rChannel world
  | toString message <> expectedMessage = abort "wrong message"
  | otherwise                           = (rChannel, world)
```

15.10.2 Distributed, parallel and concurrent programming in Haskell

Haskell features many different abstractions for implementing parallel and concurrent programs. Note that this language makes a clear distinction between these two approaches, though programmers are free to combine them. It is possible to use the `Eval` monad (which represents evaluation strategies, see Section 15.10.1), or the `Par` monad to develop parallel programs, while there

are also threads, `MVar`s, asynchronous exceptions and Software Transactional Memory to work with concurrent ones.

Parallelism

Parallel programs aim to exploit the multiplicity of computational hardware (for example, processor cores) in order to achieve faster execution. It focuses on performance but retains the deterministic nature of the original algorithm. In this scenario, different parts of the computation are delegated to different computational units to be executed at the same time, in parallel, while testing, debugging and reasoning can be performed on the sequential version.

Parallel Haskell programs do not explicitly deal with synchronization or communication, these are handled automatically by the runtime system and the parallelism libraries. Communication is implicit since all parallel tasks share the same heap, and can share objects without restriction, this may cause problems at hardware level leading to contention for the main memory bus.

Thus Parallel Haskell requires the programmer to think about how to divide a problem into tasks to be executed in parallel. There are two important considerations regarding this: the size of tasks (granularity) and sequentialization due to data dependencies between the tasks.

Parallel coordination is performed in a monad, this is the `Eval` monad. It is because parallel programming fundamentally involves ordering evaluation of expressions, for example to start evaluating `a` in parallel, and then `b`. Monads (see Section 15.6.2) are excellent tools for expressing such ordering relationships in a compositional way.

The `Eval` monad comes with two basic operations, the `rpar` combinator for creating parallelism, and the `rseq` for forcing sequential evaluation, which evaluates its argument to a *weak-head normal form*. Weak-head normal form means that the expression is evaluated until the first constructor is found. For example, in case of lists, it is only determined whether the list is empty or non-empty, but the head and tail are left untouched.

```
parMap :: (NFData b) => (a -> b) -> [a] -> Eval [b]
parMap f xs = do
  let (as,bs) = splitAt (length xs `div` 2) xs
  a <- rpar (deep (map f as))
  b <- rpar (deep (map f bs))
  rseq a
  rseq b
  return (a ++ b)
```

Figure 15.78: *(Haskell):* Use of the `Eval` monad.

A simple example of using `Eval` is featured in Figure 15.78. Here we add some parallelism to make use of two processors for processing a list (`xs`) with a function

(`f`). The initial list is divided into two nearly-equal sub-lists (`as` and `bs`). We use the `evaluate` function to evaluate the result of `runEval`. However, without using `deepseq` this will evaluate the expression only to weak-head normal form and will not compute any of the results, since it will only evaluate as far as the first cell of the list. However, `deep` evaluates the entire structure of its argument ("deeply"), reducing it to normal form, before returning the argument itself. `deep` is expressed by the `deepseq` function detailed in Figure 15.79.

```
deep :: (NFData a) => a -> a
deep x = deepseq x x
```

Figure 15.79: *(Haskell):* Implementation of the `deep` function.

The `parMap` function above can be run by the combination of the `runEval` and the `evaluate` functions, for example:

```
evaluate $ runEval $ parMap (+1) [1..10^6]
```

The GHC runtime system supports automatic distribution of parallel tasks, called dynamic partitioning. The argument to `rpar` is called a *spark*. The runtime collects sparks in a pool and uses them to share the work between the available processor by the technique of *work stealing*. Sparks may be evaluated or not, depending on the capacity available. Sparks are very cheap to create.

```
parMapDynamic :: (a -> b) -> [a] -> Eval [b]
parMapDynamic f [] = return []
parMapDynamic f (x:xs) = do
  y  <- rpar (f x)
  ys <- parMapDynamic f xs
  return (y:ys)
```

Figure 15.80: *(Haskell):* Dynamic partitioning with the `Eval` monad.

In the version presented in Figure 15.80, `parMap` runs down the whole list, eagerly creates sparks for the application of `f` to each element, and finally returns the new list. Note that the `deep` function is not added here, because it will be applied after the monad is evaluated using `runEval`, for example:

```
evaluate $ deep $ runEval $ parMapDynamic (+1) [1..10^6]
```

Evaluation strategies are an abstraction layer built on top of the `Eval` monad in a fashion similar to Section 15.10.1 (Figure 15.81).

In addition to partitioning, sometimes there is a need for being more explicit about dependencies and task boundaries than `Eval` can provide. This is where concurrency could be inserted by forking threads and explicitly assigning them tasks – however, this approach leads to losing the deterministic behavior. The

```
type Strategy a = a -> Eval a

using :: a -> Strategy a -> a
x `using` s = runEval (s x)
```

Figure 15.81: *(Haskell):* Expression of evaluation strategies atop the `Eval` monad.

```
runPar :: Par a -> a
```

Figure 15.82: *(Haskell):* Run function of the `Par` monad.

`Par` monad aims to fill this gap and makes dependencies and boundaries for tasks explicit without sacrificing determinism. Similarly to the `Eval` monad, the `Par` monad returns a pure result. However, `runPar` (Figure 15.82) is much more expensive to use than `runEval`, because it creates a new worker thread per processor. Hence, it is more recommended for scheduling large-scale parallel tasks.

We can create parallel tasks by the `fork` operation, and there is a way to communicate the results between the child of `fork` and its parent, or in general between two parallel `Par` computations. This is provided by the `IVar` type and its operations. A value of type `IVar` is like a *future* or *promise*. The `new` operation creates a new `IVar`, which is initially empty; `put` fills an `IVar` with a value, and `get` retrieves the value of an `IVar`, waiting until a value is available if necessary.

The `fork` and `IVar` operations together enable the construction of dataflow networks. The nodes of the network are created by `fork`, and edges connect a `put` with each `get` on that `IVar`.

For example, consider the following four functions (Figure 15.83):

```
f :: In -> A
g :: A -> B
h :: A -> C
j :: (B,C) -> Out
```

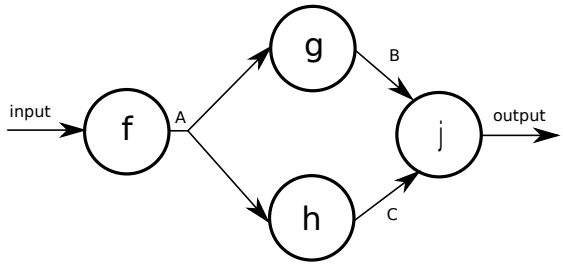


Figure 15.83: A simple dataflow network.

There are no sequential dependencies between `g` and `h`; therefore they may be run in parallel. This can be expressed as a graph in the `Par` monad, as shown below:

```
do
  [ia,ib,ic] <- replicateM 3 new
  fork (do
    x <- get input
    put ia (f x))
  fork (do
    a <- get ia
    put ib (g a))
  fork (do
    a <- get ia
    put ic (h a))
  fork (do
    b <- get ib
    c <- get ic
    put output (j b c))
```

The `Par` monad can also express other common patterns. For example, it is possible to build a `parMap` combinator, similar to the one above. This is implemented through the `spawn` function of Figure 15.84, which forks a computation in parallel and returns an `IVar` that can be used to wait for its result. The parallel map involves of calling `spawn` to apply the function to each element of the list, and then waiting for all the results, as shown in Figure 15.85. This version is different from the original version based on `Eval`. The function argument returns a monadic `Par` value that allows the computation on each element to produce more parallel tasks or to manipulate the graph in other ways, while `parMapM` waits for all the results.

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
  i <- new
  fork (do x <- p; put i x)
  return i
```

Figure 15.84: *(Haskell):* Implementation of the `spawn` function.

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f xs = do
  iys <- mapM (spawn . f) xs
  mapM get iys
```

Figure 15.85: *(Haskell):* Parallel map using the `Par` monad.

Concurrency

Concurrency is a technique for structuring programs in which there are multiple threads of control and allows introducing modularity. Such threads are executed simultaneously but their effects are interleaved, that is, they can be executed on a single processing unit through interleaving or on multiple physical ones. The "threads of control" does not make sense for purely functional programs, because there are no effects to observe, and the evaluation order is not relevant. That is, concurrency is for structuring effective code, code in the `IO` monad. Concurrent programs are necessarily nondeterministic, so it makes programs harder to test and reason about.

The basic requirement of concurrency is to be able to fork a new thread. This is implemented by the `forkIO` operation in Concurrent Haskell. It takes a computation of type `IO ()`, which is then executed in a new thread that runs concurrently with the other threads in the system. Threads are extremely lightweight in GHC, so the runtime system technically supports millions of them, limited only by the memory available. Memory used by the threads is movable so they can be packed together tightly to eliminate fragmentation; threads may also expand or shrink on demand. When using multiple physical processors, the runtime system automatically migrates threads between cores to balance the load. A trivial example of a concurrent program written in Haskell is presented in Figure 15.86.

```
import Control.Concurrent
import Control.Monad
import System.IO

main = do
  hSetBuffering stdout NoBuffering
  forkIO (forever (putChar 'A'))
  forkIO (forever (putChar 'B'))
  threadDelay (10^6)
```

Figure 15.86: *(Haskell):* A trivial concurrent program.

Threads can communicate with each other using `MVar`s as the lowest-level abstraction. An `MVar` can be thought of as a box that is either empty or full. The `newEmptyMVar` operation creates such a new empty box, and `newMVar` creates a new full box filled up with the value passed as argument. The `putMVar` operation puts a value into the box, but it waits if the box is already filled. For other direction, the `takeMVar` operation removes the value from a full box but waits if the box is empty.

`MVar`s in Haskell are quite versatile: they can be used to protect shared mutable states or critical sections (as a *lock*), establish asynchronous communication between threads, or to share a mutable state. For example, we can use them for

```
parRead :: [FilePath] -> IO [String]
parRead ps = do
  ms <- replicateM (length ps) newEmptyMVar
  mapM (\(p,m) -> forkIO (readFile p >>= putMVar m)) (ps `zip` ms)
  mapM takeMVar ms
```

Figure 15.87: *(Haskell):* Reading files concurrently.

reading multiple files concurrently, as it can be seen in Figure 15.87. `MVar`s are also useful for building larger abstractions, for example an unbounded buffered channel. The current contents of a channel can be represented as a `Stream`, like this:

```
type Stream a = MVar (Item a)
data Item a   = Item a (Stream a)
```

The end of the stream is represented by an empty `MVar`, which is called a "hole", because it will be filled with a new element. The channel itself is a pair of `MVar`s, one pointing to the first element of the `Stream` (read position) and the other pointing to the hole at the end (write position), as illustrated in Figure 15.88.

```
data Chan a = Chan (MVar (Stream a)) (MVar (Stream a))
```

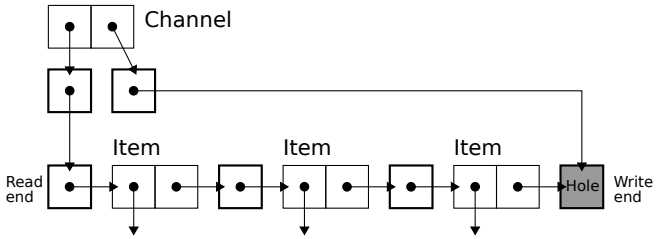


Figure 15.88: Structure of a buffered channel.

This implementation allows generalization to multicast channels without changing the structure. The associated operations can be found in the module `Control.Concurrent.Chan`.

Building larger structures with `MVar`s is not always trivial, but they have valuable properties: no thread can be blocked indefinitely on an `MVar` unless another thread holds that `MVar` indefinitely. That is, if a thread T is blocked in `takeMVar`, and there are regular `putMVar` operations on the same `MVar`, then it is guaranteed that T's `takeMVar` will return. In the implementation, this is achieved by atomically waking up the blocked thread and performing the blocked operation. A consequence of such fairness is that when multiple threads are blocked, only a single one is needed to be woken up. This "single wakeup"

property greatly contributes to efficiency when a large number of threads are contending for a single `MVar`.

For interactive applications, it is often important for one thread to be able to interrupt the execution of another thread on some particular condition, for example, in a web browser, the thread pulling data from the web server and thread rendering the page need to be interrupted when the user requests to stop. As most of the code is purely functional, it can be safely aborted or suspended, and later resumed, without affecting correctness. Therefore, Haskell employs fully-asynchronous exceptions for threads to implement cancellation.

To initiate an asynchronous exception, there is the `throwTo` primitive provided which throws an exception from one thread to another. By using `throwTo`, we can derive simple `cancel` and `wait` operations for interruptible threads. A possible implementation for them can be seen in Figure 15.89.

The `throwTo` and `cancel` functions could be supplemented with the `async` operation (Figure 15.90) to catch exceptions in the thread body and store them in the `MVar`.

```
data Async a = Async ThreadId (MVar (Either a SomeException))

cancel :: Async a -> IO ()
cancel (Async t _) = throwTo t ThreadKilled

wait :: Async a -> IO a
wait (Async _ m) = readMVar m
```

Figure 15.89: *(Haskell):* Implementation of the `cancel` and `wait` operations.

```
async :: IO a -> IO (Async a)
async io = do
  m <- newEmptyMVar
  t <- forkIO ((do r <- io; putMVar m (Right r))
                 `catch` \e -> putMVar m (Left e))
  return (Async t m)
```

Figure 15.90: *(Haskell):* Implementation of the `async` operation.

Asynchronous exceptions can be masked in order to protect critical sections using the `mask` combinator. It defers the delivery of exceptions for the duration of its argument. Inside `mask`, exceptions are no longer asynchronous, but they can still be raised by certain operations, that is, they become synchronous. So a `bracket` wrapper may be defined with `mask` to make operations safe from asynchronous exceptions. This is demonstrated in Figure 15.91.

Many exceptional conditions map naturally onto asynchronous exceptions, for example stack overflow or user interrupts. Threads never just stop and

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
bracket before after during =
  mask (\restore -> do
    a <- before
    r <- restore (during a) `catch` \e -> after a; throw e
    _ <- after a
    return r)
```

Figure 15.91: *(Haskell):* Implementation of the `bracket` wrapper function.

disappear, it is guaranteed that a thread always gets a chance to clean up and run its exception handlers.

Software Transactional Memory (STM) is a technique for making concurrent programming simpler by grouping multiple operations over state and performing them as a single atomic operation, a transaction. STM provides a way to avoid deadlocks without imposing a requirement for ordering on concurrent accesses. It is just enough to replace `MVar`s with `TVar`s, and wrap the sequence of grouped operations in `atomically`. `TVar` stands for "transactional variable", and it is a mutable variable that can only be read or written within a transaction.

```
atomically :: STM a -> IO a
```

The `atomically` function is the associated run function of the `STM` monad, which is to execute the contents of the monadic block as an atomic operation. This happens invisibly as far as the rest of the program is concerned. No other thread can observe an intermediate state, the operation has either completed, or it has not started yet. This approach scales to any number of `TVar`s. In addition to that, STM operations are *composable*: any operation of type `STM a` can be composed with other operations to form a larger atomic transaction. As a result of this, `STM` operations are usually provided without the `atomically` wrapper. Thus, one can compose them as necessary, before finally wrapping the entire operation in `atomically`.

An important part of concurrent programming is dealing with blocking when one needs to wait for some condition to be true, or to acquire a particular resource. STM also provides an elegant way to achieve this, with the `retry` operation. This means "running the current transaction again". This is so because contents of certain `TVar`s involved in the transaction may have been changed by another thread, and thus re-running the transaction may yield different results. The runtime system knows if something has changed in the block, so `retry` waits until a `TVar` that was read in the current transaction has been written to, and then triggers a re-run of the current transaction. Until that happens, the current thread is blocked. Without `retry`, a complex logic should have been implemented, including the signals between threads. This would be against modularity, since operations modifying the state have to know about the potential observers that need to act on changes. This also rises the most

frequent problem: lost wakeups. As woken threads were supposed to complete some operation on which other threads are waiting, such events often lead to deadlocks.

`STM` allows composition of blocking operations by the `orElse` operation. It takes two `STM` blocks: `a` and `b`. First `a` is executed. If `a` returns a result, the result is immediately returned. If `a` calls `retry`, `a`'s effects are discarded, and `b` is executed instead.

The `STM` monad supports exceptions similarly to the `IO` monad, with the `throwSTM` and `catchSTM` operations. The main difference is that `catchSTM` discards all the effects of the guarded block, and calls the associated handler. If there is no enclosing `catchSTM` operation, all of the effects of the entire transaction are discarded and the exception is propagated out of `atomically`. This is particularly useful in the case of asynchronous exceptions, because there are no locks to replace, no need for exception handlers, and no need to worry about which critical sections to protect with `mask`.

Distributed execution

Eden [Loo12] is a parallel functional programming language which enhances Haskell with constructs for definition and instantiation of parallel processes. The processes evaluate function applications remotely in parallel. While the programmer has control over process granularity, data distribution, communication topology and site of evaluation, there is no need to care about synchronization and data exchange between processes as it is performed by the runtime system. In addition to this, common patterns of communication and topologies are provided in the form of "algorithmic skeletons" – higher-order functions that could be extended by the user. Eden is aimed toward distributed computing as processes do not share any data, but it is also suitable for programming multi-core systems as well.

The underlying idea of Eden is to specify process networks in a declarative way, where processes evaluate functions. The function parameters are the process inputs and the function result is the process output. Input and output are automatically transferred via unidirectional one-to-one channels between parent and child processes, meaning that they are always fully evaluated before sending, and due to lack of sharing, the same expression may be redundantly evaluated by several processes.

Since only hierarchical communication topologies are supported automatically, Eden provides explicit channel management. A receiver process can create a new input channel and pass its name to another process. Then the target process can directly send data to the original process using the received channel name.

There is also the *remote data concept* where data can be released by a process to be fetched by a remote process. In this case, a handle is transferred from the owner to the receiver process, which can then be used to directly send data from

the producer to the receiver. Furthermore, many-to-one communication can be modeled using a pre-defined (necessarily non-deterministic) merge function.

Arbitrary parallel computation schemes, like master-worker systems or cyclic communication topologies like rings and tori, can be defined in an elegant way. Eden supports an equational programming style where recursive process nets can simply be defined using recursive equations. Using the `PA` (parallel action) monad, it is also possible to adopt a monadic programming style, when it is necessary to ensure that series of parallel activities are executed in a given order.

As an example, calculate digits of π by approximating the integral

$$\pi = \int_0^1 f(x)dx \ \texttt{where}\ f(x) = \frac{4}{1+x^2}$$

in the following way:

$$\pi = \lim_{n\to\infty} pi(n) \ \texttt{where}\ pi(n) = \frac{1}{n}\sum_{i=1}^{n} f\left(\frac{i-0.5}{n}\right)$$

The corresponding program is presented in Figure 15.92. This is based on a simplified map-reduce scheme that can be given in Haskell easily as it is shown in Figure 15.93.

```
import Control.Parallel.Eden
import Control.Parallel.Eden.EdenSkel.MapRedSkels
cpi :: Integer -> Double
cpi n = offline_parMapRedr (+) 0 (f . index) [1..n] / fromInteger n
  where
    f :: Double -> Double
    f x = 4 / (1 + x * x)
    index :: Integer -> Double
    index i = (fromInteger i - 0.5) / fromInteger n
```

Figure 15.92: *(Haskell):* Eden function for the calculation of π.

```
mapRedr :: (b -> c -> c) -> c -> (a -> b) -> [a] -> c
mapRedr g e f = (foldr g e) . (map f)
```

Figure 15.93: *(Haskell):* Implementation of the `mapRedr` function.

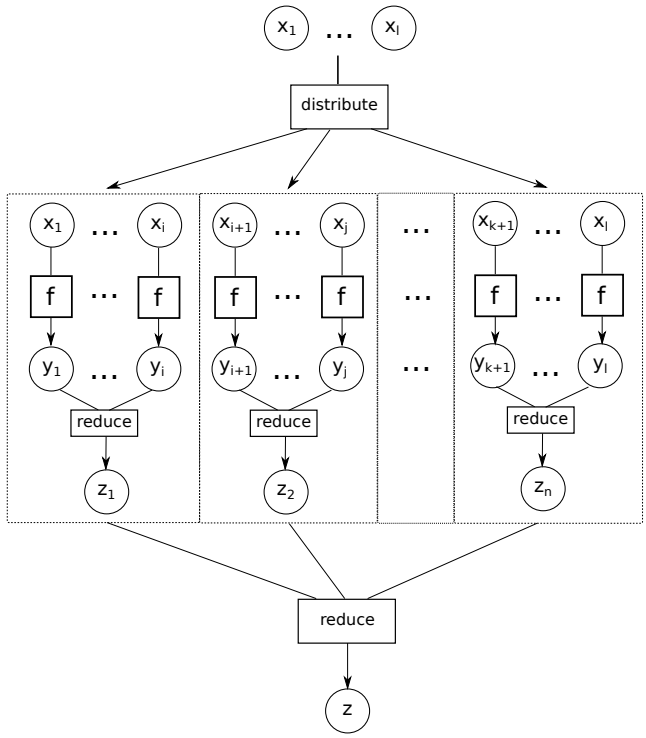


Figure 15.94: The parallel map-reduce scheme in Eden.

If the parameter function `g` is associative with type `b -> b -> b` and the neutral element `e`, reduction may also be performed in parallel by splitting the list into sublists and pre-reducing them within the parallel processes. Afterwards only the subresults from the processes have to be combined by a main process (Figure 15.94). So the `parMapRedr` function can be used instead of `mapRedr`, which is listed in Figure 15.95.

```
parMapRedr :: (Trans a, Trans b)
  => (b -> b -> b) -> b -> (a -> b) -> [a] -> b
parMapRedr g e f xs =
  if noPe == 1
    then mapRedr g e f xs
    else (foldr g e) . (parMap (mapRedr g e f)) . (splitIntoN noPe)
```

Figure 15.95: *(Haskell):* Parallel implementation of the `mapRedr` function.

Note that parallel processes are only created if there at least 2 processors present in the system. On a single processor the original `mapRedr` sequential

scheme is executed. This is determined from the value of the `noPe` constant which gives the number of available processors.

This solution uses the `parMap` skeleton where a separate process is created for each function application, that is, as many processes be created as the number of list elements. The input parameter as well as the result of each process will be transmitted via communication channels between the generator process and the process created by `parMap`.

```
parMap :: (Trans a, Trans b) => (a -> b) -> [a] -> [b]
```

In the signature of `parMap`, the Eden-specific type context – containing the `Trans` type class – indicates that both types are *transmissible* values. The another skeleton employed here is `splitIntoN`:

```
splitIntoN :: Int -> [a] -> [[a]]
```

This function distributes the input list blockwise into as many sublists as the first parameter determines. The length of the output lists differs by the most one. The inverse function of `splitIntoN` is the standard Haskell function `concat` which simply concatenates all lists in the given list of lists.

The input lists of the processes are evaluated by the parent process and then communicated via automatically created communication channels between the parent and the `parMap` processes. In Eden, lists are transmitted as streams where each element is sent in a separate message. This causes a severe overhead for very long lists. The `offline_parMapRedr` variant in Figure 15.96 avoids the stream communication. Only the process identification number is communicated and used to select the appropriate part of the input list. The whole (unevaluated) list is incorporated in the worker function which is mapped onto the identification numbers. This may cause redundancy in the input evaluation, but overall it substantially reduces communication overhead.

```
offline_parMapRedr :: (Trans a, Trans b)
  => (b -> b -> b) -> b -> (a -> b) -> [a] -> b
offline_parMapRedr g e f xs =
  if noPe == 1
    then mapRedr g e f xs
    else foldr g e (parMap worker [0 .. (noPe - 1)])
  where
    worker i = mapRedr g e f ((splitIntoN noPe xs) !! i)
```

Figure 15.96: *(Haskell):* Offline parallel implementation of `mapRedr` function.

15.10.3 Parallel and distributed language constructs of JoCaml

JoCaml [FFMS01] is not a purely functional language – it contains imperative and object-oriented elements as well. In JoCaml, mobile code is represented

by agents. Design of the language is based on three main demands: to provide tools for creating secure agents, to express complex distributed computations, and to describe semantics of the abstract language constructs precisely. The two most important abstract language constructs are the concepts of channel and abstract place. Synchronous and asynchronous channels are created by the `let def` statement, identifiers of asynchronous channels are followed by a `!` (bang) symbol. Data for asynchronous channels is handled by assigned processes.

```
let def echo! X = print_int x;
val echo : <<int>>
```

Figure 15.97: *(JoCaml):* Definition of a trivial asynchronous channel.

The `spawn` keyword creates an expression out of process instances assigned to an asynchronous channel, whose evaluation leads to the launching of a concurrent process. Message passing is demonstrated in Figure 15.98, where values of channels assigned to concurrent processes are printed – that is, the 1, 2, 3 integer values – in an undefined order.

```
spawn{
   let x = 1 in
   {let y = x + 1 in echo y | echo (y + 1)} | echo x }
```

Figure 15.98: *(JoCaml):* Message passing.

Complex control and data handler patterns of concurrent programming are described by synchronization patterns. This way it is possible to express synchronization or conflicts between communication events associated with channel sets. Synchronization patterns (Figure 15.99) always refer to channels defined by the same `let def` statement in parallel. Patterns can be declared by the `|` parallel composition operation; multiple alternative patterns associated with channel subsets can be joined by the `or` operator. An example of concurrent counters is the `count` asynchronous channel, whose received messages are synchronized with messages sent to the `inc` and `get` synchronous channels. After the initial `count 0` message, the concurrent system created this way increases the counter when an `inc` message is received by sending itself a new `count` message storing the counter's new value there. When a `get` message is received, the counter sends itself a `count` message, warranting that the operation remains correct.

Distributed programs can be also written in JoCaml. Processes may migrate from one machine to another. In the beginning, there is no connection between the processes running on different machines in the beginning, they find each other by using a name server.

Concept of abstract place incorporates both the agent and its actual physical location. This way JoCaml offers abstract tools for handling composition and

```
let def count! n | inc () = count (n + 1) | reply to inc
     or count! n | get () = count n | reply n to get
spawn {count 0}
```

Figure 15.99: *(JoCaml):* Synchronization patterns.

communication independently of the actual place where the internal state is stored during migration of code, and for calling remote procedures. Components of agents migrate together with the containing parent agent; communication channels remain established between agents changing their locations over time, and agents resume their activities from their post-migration state.

In the next code, one of the processes registers the `f` shared resource under the name `square`, and another finds and uses it.

```
spawn {
  let def f x = reply x * x in
  Ns.register "square" f vartype;
}

spawn {
  let sqr = Ns.lookup "square" vartype in
  print_int(sqr 2);
  exit 0;
}
```

Abstract places can be created by the `let loc` ⟨*identifier*⟩ `do {}` construction. Physical places can be identified as actual values of abstract places, that is, as actual physical places of agents. The `here` empty agent is used for this purpose. The `mobile` agent queries data of the `here` agent from the name server, then it migrates itself to the actual physical place of `here` and performs computations as a sub-agent. Downloadable agents can have a parameter to learn where to migrate when activated, but data-driven migration can be also implemented.

```
let loc here do {
  Ns.register "here" here vartype;
  Join.server ();
}
let loc mobile
  do {
    let here = Ns.loopkup "here" vartype in
    go here;
    let sqr = Ns.lookup "square" vartype in
    let def sum (s,n) =
      reply (if n = 0 then s else sum(s + sqr n, n - 1)) in
    let result = sum (0,5) in
    print_string ("q: sum 5 = " ^ string_of_int result ^ "\n");
    flush stdout;
```

```
}
```

15.11 Summary

In this chapter, we have offered a brief overview of the features of contemporary functional languages. In conjunction with this, the functional programming style has been introduced to show the major differences between the functional programming paradigm and the imperative program development. We have looked at issues such as the referential transparency, strong static typing, use of higher-order functions, partial application of functions, recursive functions and data structures, lazy evaluation and working with infinite lists, list comprehensions, pattern matching, algebraic data types, module systems, various I/O models, and the application of the so-called off-side rule. To demonstrate all these features, we have used source code examples written in some of the most influential languages, such as SML, Miranda, Clean, and Haskell.

During this presentation, we have considered some of the current challenges of the real world. We have also shown examples for how error are dealt with in functional programs, how dynamically-typed expressions may be inserted in the strong statically typed environment, and for how parallel, concurrent, and distributed software are developed. We believe that the discussed solutions have revealed that research on functional programming has not stopped and it is a thriving field of research, highly fruitful topic for both the academia and industry. Functional languages have always had influence on mainstream programming languages and they have greatly contributed to the evolution of the most popular programming concepts and techniques. Hence, the knowledge of the functional programming approach should not be an optional but a must-have tool in the toolkit of a professional programmer.

15.12 Exercises

Exercise 15.1. Determine prime factors for an integer.

Exercise 15.2. Determine if a given integer is a perfect number or not.

Exercise 15.3. Sort elements of a sequence in ascending order.

Exercise 15.4. Find the smallest element in a sequence. The ordering relation has to be an argument of the function.

Exercise 15.5. Determine the average of elements of a sequence.

Exercise 15.6. Determine if a string is contained by another one.

Exercise 15.7. Define an abstract algebraic data type to represent the ”bag” data structure.

Exercise 15.8. Write an interactive functional program that displays a menu on the top of the screen and opens a dialog window from one of the pull-down submenus. The dialog window has to contain two editable input boxes, a label to display the result, and a button. Read two integers in the input boxes and render the result of their addition when the button is pressed.

15.13 Useful tips

General advice. Try to break the solutions into smaller, well-characterized problems, and give a function definition for each of them. It may help if the types for these functions can be given.

Tip 15.1. Try to generate the list of all integers first and then keep only the ones that are considered primes. The resulting list then may be used to test its elements for divisibility with the parameter. Store the ones that divide the parameter, while keep it decrementing with the values of the stored integers.

Tip 15.2. Generate the list of proper divisors, sum its elements and determine if this is equal to the number itself. The sum can be calculated in a recursive fashion where the head of the list is added to the sum of the tail. (Bear in mind that the tail of the list is a list itself, so the function can be applied it, thus making it recursive.)

Tip 15.3. Pick a sorting algorithm and implement it. For functional languages, a good choice would be Quicksort because it is recursive, and makes it easier to implement it. While here, try to create a polymorphic function that works with all the ordered types.

Tip 15.4. Since the ordering relation has to be an argument to the function, this assumes implementation of a higher-order function. That is, the ordering relation can be taken as a function that takes two elements of the same type and returns some value that indicates their relation to each other (for example, negative number: the first is lesser than the second, zero: they are equal, positive number: the first is greater than the second). Try to make this function polymorphic, that is, make it work with all types that are ordered.

Place the elements of the sequence in a list, and use the relation function to walk the elements of the list recursively. Note that the list can be split into a ”head” and a ”tail” section, so finding the minimum element likely becomes a comparison (via the relation function) between the head and the minimum of the tail. (Bear in mind that the tail of the list is a list itself, so the function can be applied to it, hence making it recursive.) The recursion should stop when the list has only a single element – in this case, the minimum is the element itself. The function should not be defined for empty lists.

Tip 15.5. Place the elements of the sequence in a list and add them up. The sum can be calculated in a recursive fashion where the head of the list is added to the sum of the tail. (Do not forget that the tail of the list is a list itself, so the function can be applied it, hence it becomes recursive.)

Determine the length of the list which technically corresponds to the number of elements in the sequence. Length can be calculated by recursively walking the list and adding one to the length of the "tail" each time when a "head" is found. The length of the empty list is zero by definition.

Finally, divide the sum of the elements with the length. Bear in mind that some of the languages strictly distinguish between integer and floating-point division.

Tip 15.6. Take the strings as a list of characters. Generate all the possible suffixes for the second string. This can be implemented by recursively generating a list of lists where the "head" of the resulting sublists is always dropped.

Write a function that checks if a string is a prefix to another one. This can be done by recursively walking the two lists in parallel and comparing elements with the same indices. If the first list is fully consumed and all the elements match, the result is true, otherwise it must be false.

Finally, combine the previous two functions: generate all the possible suffixes for the second string and check if the first string is a prefix for any of them.

Tip 15.7. Define a data type to represent the bag. For example, this can be a list that stores pairs of elements and their quantity. Implement the following operations on this data structure: create an empty bag, convert a list of pairs of elements and quantities to a bag and vice versa, insert the element into a bag, delete the element from a bag, check if an element is in the bag, determine if all elements of a bag contained by another bag, count the number of bag elements, scale values of elements by a given natural number, union of bags.

Tip 15.8. Find a GUI library that helps with building the graphical application. For Clean, this is the Object IO library, or in case of Haskell, these are wxHaskell and Gtk2Hs, there may be other libraries. In order to write the program, it is recommended to be familiar with programming with side effects.

15.14 Solutions

Please note that all the solutions have been given in Haskell, but there are many other solutions as well, the collection of applicable techniques is growing continuously. Feel free to experiment with other languages as well.

Solution 15.1.

```
import Prelude hiding (null)

-- Determine prime factors for an integer.
primeFactors :: Integer -> [Integer]
```

```
primeFactors n | n > 1 = f n primes
  where
    f n ps@(x:xs)
      | x^2 > n       = [n]
      | x `divides` n = x : f (n `div` x) ps
      | otherwise     = f n xs
primeFactors _ = []

x `divides` y = y `mod` x == 0

iSqrt = round . sqrt . fromIntegral

isPrime n = null [ x | x <- [3..(iSqrt n)], x `divides` n ]
  where
    null :: [a] -> Bool
    null [] = True
    null _  = False

primes = 2 : [ x | x <- [3,5..], isPrime x ]
```

Solution 15.2.

```
import Prelude hiding (sum)

-- Determine if a given integer is a perfect number or not.
isPerfectNumber :: Integer -> Bool
isPerfectNumber n
  | n > 0 = sum (properDivisors n) == n
isPerfectNumber _ = error "isPerfectNumber: not a positive integer"

x `divides` y = y `mod` x == 0

properDivisors n = 1 : [ x | x <- [2 .. (n `div` 2)], x `divides` n]

sum :: [Integer] -> Integer
sum []     = 0
sum (x:xs) = x + sum xs
```

Solution 15.3.

```
-- Sort elements of a sequence in ascending order.
-- (Implemented by the "quick sort" algorithm, although Data.List.sort
-- provides a standard sorting function for Haskell.)
sortAscending :: Ord a => [a] -> [a]
sortAscending (x:xs) = (sortAscending left) ++ [x] ++ (sortAscending right)
  where
    left  = [ y | y <- xs, y <= x ]
    right = [ y | y <- xs, y > x ]
sortAscending []     = []
```

Solution 15.4.

```
-- Find the smallest element in a sequence.  The ordering relation has to be
-- an argument to the function.

-- This function uses the standard definition of `Ordering` in Haskell:
-- data Ordering = LT | EQ | GT
-- (This is implemented by the `Data.List.minimumBy` function.)
minimumBy :: (a -> a -> Ordering) -> [a] -> a
minimumBy f [x]    = x
minimumBy f (x:xs) =
  case (x `f` minimumBy f xs) of
    LT -> x
    _  -> minimumBy f xs
minimumBy _ []     = error "minimumBy: empty list"
```

Solution 15.5.

```
import Prelude hiding (sum, length)

-- Determine the average of elements of a sequence.
average :: Fractional a => [a] -> a
average [] = error "average: empty list"
```

```
average xs = sum xs / fromIntegral (length xs)

sum :: Num a => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs

length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

Solution 15.6.

```
import Prelude hiding (any)

-- Determine if a string is contained by another one.
-- (This is implemented by the `Data.List.infixOf` function.)
isSubstringOf :: String -> String -> Bool
x `isSubstringOf` y = any (isPrefixOf x) (tails y)

tails :: [a] -> [[a]]
tails [] = [[]]
tails xs = xs : tails (tail xs)

[]      `isPrefixOf` _      = True
_       `isPrefixOf` []     = False
(x:xs)  `isPrefixOf` (y:ys) = (x == y) && (xs `isPrefixOf` ys)

any :: (a -> Bool) -> [a] -> Bool
any _ []     = False
any p (x:xs) = p x || any p xs
```

Solution 15.7.

```
-- Define an abstract algebraic data type to represent the ``bag'' data
-- structure.

-- (This is implemented by Data.MultiSet, here a simplified implementation
-- is given.)

module Bag
  ( Bag
  , empty
  , insert
  , delete
  , contains
  , isSubBag
  , count
  , scaledBy
  , union
  , fromList
  , toList
  ) where

data Bag a = Empty | Cons a Integer (Bag a)

instance (Eq a) => Eq (Bag a) where
  x == y = (x `isSubBag` y) && (y `isSubBag` x)

instance (Eq a) => Ord (Bag a) where
  x `compare` y
    | x == y           = EQ
    | (x `isSubBag` y) = LT
    | otherwise        = GT

fromList :: (Eq a) => [(a,Integer)] -> Bag a
fromList []         = Empty
fromList ((x,n):xs) = Cons x n (fromList xs)

toList :: Bag a -> [(a,Integer)]
```

```
toList Empty          = []
toList (Cons x n xs)  = (x,n):(toList xs)

-- An empty bag.
empty :: Bag a
empty = Empty

-- Insertion of a new element (with multiplicity of 1) or increasing the
-- multiplicity of an already stored element.
insert :: (Ord a) => a -> Bag a -> Bag a
x `insert` Empty          = Cons x 1 Empty
x `insert` (Cons y n ys)
  | x == y    = Cons y (n + 1) ys
  | x > y     = Cons y n (x `insert` ys)
  | otherwise = Cons x 1 (Cons y n ys)

-- Removal of an occurence for an element, removing it completely if
-- occurence falls below 1.
delete :: (Eq a) => Bag a -> a -> Bag a
delete Empty         _  = Empty
delete (Cons y n ys) x
  | (x == y) && (n > 1) = Cons y (n - 1) ys
  | x == y              = ys
  | otherwise           = Cons y n (delete ys x)

-- Check whether the element is present (at least once) in the bag.
contains :: (Eq a) => Bag a -> a -> Bool
b `contains` x = (count b x) > 0

-- Checks whether each element in the first bag occurs no more often
-- than it occurs in the second bag.
isSubBag :: (Eq a) => Bag a -> Bag a -> Bool
Empty         `isSubBag` _      = True
_             `isSubBag` Empty  = False
(Cons x n xs) `isSubBag` b      = (n <= count b x) && (xs `isSubBag` b)

-- Return the number of times that the element occurs in the bag
count :: (Eq a) => Bag a -> a -> Integer
count Empty         _ = 0
count (Cons y n ys) x
  | x == y    = n
  | otherwise = count ys x

-- Given a natural number n, return a bag which contains the same elements as
-- the bag, except that every element that occurs in the bag occurs n times
-- more in the resulting bag.
scaledBy :: Bag a -> Integer -> Bag a
Empty         `scaledBy` _  = Empty
(Cons x n xs) `scaledBy` m  = Cons x (n * m) (xs `scaledBy` m)

-- Return a bag that containing just those values that occur in either the
-- first or the second bag, except that the number of times a value occurs in
-- the resulting bag is equal to sum of occurences in both bags.
union :: (Ord a) => Bag a -> Bag a -> Bag a
Empty             `union` ys    = ys
xs                `union` Empty = xs
xs@(Cons x n xs') `union` ys@(Cons y m ys')
  | x == y      = Cons x (n + m) (xs' `union` ys')
  | x < y       = Cons x n (xs' `union` ys)
  | otherwise   = Cons y m (xs `union` ys')
```

Solution 15.8.

```
-- Write an interactive functional program that displays a menu on the top of
-- the screen and opens a dialog window from one of the pull-down submenus.
-- The dialog window has to contain two editable input boxes, a label to
-- display the result, and a button.  Read two integers in the input boxes and
```

```
-- render the result of their addition when then button is pressed.

-- This solution uses the Gtk2Hs toolkit, available at
-- http://haskell.org/gtk2hs/

import Control.Applicative
import Control.Monad
import Data.Maybe
import Graphics.UI.Gtk

-- This function represents the entry point for the entire application.  It
-- creates a main window that contains some (stub) menus and the adder window.
main :: IO ()
main = do
  initGUI

  -- Contents of the main window
  window <- windowNew
  set window
    [ windowTitle         := "Main Window"
    , windowDefaultWidth  := 450
    , windowDefaultHeight := 200
    ]

  box <- vBoxNew False 0
  containerAdd window box

  -- Menus
  fma <- actionNew "FMA" "File" Nothing Nothing
  ema <- actionNew "EMA" "Edit" Nothing Nothing
  oma <- actionNew "OMA" "Operations" Nothing Nothing
  hma <- actionNew "HMA" "Help" Nothing Nothing

  -- Menu items
  newa <- actionNew "NEWA" "New"     (Just "Just a Stub") (Just stockNew)
  opna <- actionNew "OPNA" "Open"    (Just "Just a Stub") (Just stockOpen)
  sava <- actionNew "SAVA" "Save"    (Just "Just a Stub") (Just stockSave)
  svaa <- actionNew "SVAA" "Save As" (Just "Just a Stub") (Just stockSaveAs)
  exia <- actionNew "EXIA" "Exit"    (Just "Just a Stub") (Just stockQuit)

  cuta <- actionNew "CUTA" "Cut"   (Just "Just a Stub") (Just stockCut)
  copa <- actionNew "COPA" "Copy"  (Just "Just a Stub") (Just stockCopy)
  psta <- actionNew "PSTA" "Paste" (Just "Just a Stub") (Just stockPaste)

  hlpa <- actionNew "HLPA" "Help"  (Just "Just a Stub") (Just stockHelp)
  adda <- actionNew "ADDA" "Adder" (Just "Calls the adder") Nothing

  agr <- actionGroupNew "AGR"
  forM_ [fma, ema, oma, hma] $ actionGroupAddAction agr
  forM_ [newa,opna,sava,svaa,cuta,copa,psta,hlpa,adda] $ \act ->
    actionGroupAddActionWithAccel agr act Nothing

  actionGroupAddActionWithAccel agr exia (Just "<Control>e")

  ui <- uiManagerNew
  uiManagerAddUiFromString ui uiDecl
  uiManagerInsertActionGroup ui agr 0

  maybeMenubar <- uiManagerGetWidget ui "/ui/menubar"
  let menubar = case maybeMenubar of
                  Just x  -> x
                  Nothing -> error "Cannot get menubar from string."
  boxPackStart box menubar PackNatural 0

  maybeToolbar <- uiManagerGetWidget ui "/ui/toolbar"
```

```
  let toolbar = case maybeToolbar of
                  Just x  -> x
                  Nothing -> error "Cannot get toolbar from string."
  boxPackStart box toolbar PackNatural 0

  actionSetSensitive cuta False
  onActionActivate exia (widgetDestroy window)
  forM_ [fma,ema,hma,newa,opna,sava,svaa,cuta,copa,psta,hlpa] $ \a ->
    onActionActivate a $ do
      name <- actionGetName a
      putStrLn ("Action Name: " ++ name)

  onActionActivate adda (newAdderWindow >> return ())

  widgetShowAll window
  onDestroy window mainQuit
  mainGUI

-- Creates a new window with a simple adder application in it.
newAdderWindow :: IO Window
newAdderWindow = do
  window <- windowNew
  set window [windowTitle := "Adder", containerBorderWidth := 10]

  vb <- vBoxNew False 0
  containerAdd window vb

  hb <- hBoxNew False 0
  boxPackStart vb hb PackNatural 5

  -- Text entry fields.
  n1s <- entryNew
  n2s <- entryNew
  boxPackStart hb n1s PackRepel 5
  boxPackStart hb n2s PackRepel 5
  button <- buttonNewWithLabel "Add"
  boxPackStart vb button PackNatural 5
  -- A label as response.
  response <- labelNew (Just "No result.")
  boxPackStart vb response PackNatural 5

  widgetShowAll window

  onPressed button (addNumbers n1s n2s response)
  return window
-- Add contents of two text entries.  If the corresponding strings cannot
-- be parsed then it displays that the input is invalid.
addNumbers :: Entry -> Entry -> Label -> IO ()
addNumbers e1 e2 r = do
  [str1,str2] <- entryGetText `mapM` [e1,e2]
  let result = (+) <$> maybeRead str1 <*> maybeRead str2
  let answer =
        case result of
          Just n -> "Result: " ++ show n
          _      -> "Invalid input."
  labelSetText r answer
  return ()
-- A helper function for parsing types from the Read class to Maybe values.
maybeRead :: (Read t) => String -> Maybe t
maybeRead = fmap fst . listToMaybe . reads
-- GUI structure of the main window (as XML).
uiDecl = "<ui>\
\          <menubar>\
\            <menu action=\"FMA\">\
\              <menuitem action=\"NEWA\" />\
```

```
\              <menuitem action=\"OPNA\" />\
\              <menuitem action=\"SAVA\" />\
\              <menuitem action=\"SVAA\" />\
\              <separator />\
\              <menuitem action=\"EXIA\" />\
\            </menu>\
\            <menu action=\"EMA\">\
\              <menuitem action=\"CUTA\" />\
\              <menuitem action=\"COPA\" />\
\              <menuitem action=\"PSTA\" />\
\            </menu>\
\            <menu action=\"OMA\">\
\              <menuitem action=\"ADDA\" />\
\            </menu>\
\            <separator />\
\            <menu action=\"HMA\">\
\              <menuitem action=\"HLPA\" />\
\            </menu>\
\          </menubar>\
\          <toolbar>\
\            <toolitem action=\"NEWA\" />\
\            <toolitem action=\"OPNA\" />\
\            <toolitem action=\"SAVA\" />\
\            <toolitem action=\"EXIA\" />\
\            <separator />\
\            <toolitem action=\"CUTA\" />\
\            <toolitem action=\"COPA\" />\
\            <toolitem action=\"PSTA\" />\
\            <separator />\
\            <toolitem action=\"HLPA\" />\
\          </toolbar>\
\        </ui>"
```

16 Logic programming and Prolog

Provided that our knowledge about a problem is modeled by axioms, we can pose queries in the form of statements to be proved, in order to find objects satisfying these statements to be proved. The process of formal, constructive proof or deduction is some kind of computation. If this computation is controlled by an algorithm, then our statements together with the algorithm form a program, and writing such programs is logic programming (LP). In other words, a logic program is a set of axioms and a statement to be proved + a machine controlling deduction. It is similar to an automated theorem prover. The main difference is the simplicity of the control component of logic programs: the process of computing is trackable, controllable, predictable. Its termination can be ensured. Its time and space complexity can be calculated: an LP language is a general purpose programming language, and effectivity is a main point.

In this chapter we survey the development of logic programming, its key concepts, and its (up till now) most important realization: the Prolog language. We discuss the programming methodology of Prolog, give examples, consider some extensions, and new trends. We emphasize the practical methods of Prolog programming; how to write valid and effective programs.

16.1 Introduction

The roots of logic programming (LP) reach as far as Hilbert, who proclaimed his program to axiomatize mathematics because the naive set theory had been found burdened with contradictions. Also he proposed to work out the methods of *automated theorem proving*.

The birth of the resolution algorithm [Rob65] is an important milestone of research, because we can extract answers from the run of the algorithm. Therefore it is a tool for constructive proof, and the deduction is a program searching or computing objects with given properties.[1]

Provided an automated theorem prover, the algorithm of proof or computation consists of two components:

1. The logical (declarative) description of the problem;
2. And the control component of the deduction or computation.

Shortly:

Algorithm = Logic + Control.

After some unsuccessful american efforts, in the beginning of the seventies Robert Kowalski realized that a general purpose programming language can be based on these principles [Kow79]. Alain Colmerauer and his colleagues designed and implemented the first LP language with acceptable performance in Marseille, 1972. This was the first version of Prolog. It was implemented as an interpreter.

The practical purpose of the developers was a natural language interface for a data base handler, and this became the first practical application of Prolog.[2]

[1] Familiarity with first order logic, and with resolution algorithm ([Nil82] and [Kow79]) is supposed, although the main flow of this chapter is understandable even without them.

[2] The name Prolog is an abbreviation of the French expession **Pro***grammation en* **log***ique*.

The first effective Prolog compiler was developed by David H. D. Warren and his colleagues at Edinborough in the second half of the seventies. Considering effectivity, it was comparable with best Lisp implementations of the time. It became a *de facto* standard of Prolog, determined the direction of its later development, and indirectly even its ISO standard, which was developed in the middle of nineties. Unfortunately, still there are compatibility problems between the module systems of the major Prolog compilers. And there are the extensions of the language allowed by the standard. Consequently, program code written on one Prolog compiler will not necessarily work on others, except if only the core of the language is used. Whenever we go beyond this, we sign it and we follow SICStus Prolog 4 [Car12], because nowdays this is one of the most popular implementations, and it follows closely the standard. (Our example programs have been tested in SICStus Prolog 4.2.3.)

Up till now, Prolog is the most widely used LP language. Its designers proved a practical sensibility. The compromises they made were theoretically criticized. Nevertheless, Prolog helps us in a good programming style, and in the fast development of programs with just a few errors, producing effective codes. This is the secret of its relative success.

Maybe there is no other language in which good programming style is so basic from the point of view of producing robust and effective code. Therefore, in this short chapter we concentrate on the *basic* features of the Prolog language, and on its *programming methodology*. We prefer this to a general introduction to LP, because we believe that this approach gives more to the newcomer. And we hope, at the end of the chapter we give good pointers to the reader yearning for wider knowledge.

16.2 Logic programs

A logic program is a set of logic *axioms* referring to a model, and a *query* referring to this model. The axioms describe the properties and relations of the objects of this model. For example, given statements defining who is who's father. These axioms describe the relation `father` with arity two. If we tell who are male, the appropriate axioms describe the relation `male` with arity one (16.2.1).

A subset of axioms describing a relation is called a *predicate*. The run of a program is a constructive proof of a theorem being the consequence of the axioms. In other words, the run of the program computes an answer to a *query* or *goal*. During this computation or proof the predicates work as procedures.

In today's LP languages, any axiom is a *fact* or a *rule*. The axioms and queries (i.e. goals) are called *sentences* (although the queries may not be part of the source program). In some LP languages, the sentences can also be *declarations* referring to the predicates, and *directives* to be performed while loading the program. The declarations modify the run of the predicates, i.e. procedures, while the directives may modify the workspace of the program. The declarations

describe special properties of the predicates they refer to, while the directives are goals containing predicate invocations.

In Prolog, the declarations and the directives have the same form:
$:-goal.$
Each sentence is terminated by a dot, and at least one whitespace character.

The axioms of a logic program are also called *definite clauses*. The *statements* (the axioms and queries together) of a logic program are also called *Horn clauses*, but we will simply use the word *clauses*, because we will speak just about *Horn clauses*, and this abbreviation is quite common in logic programming.

16.2.1 Facts

The simplest logic programs consist of facts only.

The facts are formally atomic formulas. In the next example `father(x,y)` means that x is the father of y, while `male(x)` means that x is a male.

```
father('Abraham','Isaac'). father('Abraham','Ishmael').
father('Abraham','Anon').
father('Isaac','Jacob').   father('Isaac','Esau').

mother('Sarah','Isaac').   mother('Hagar','Ishmael').
mother('Rebeka','Jacob').  mother('Rebeka','Esau').

male('Abraham'). male('Isaac'). male('Ishmael').
male('Jacob').  male('Esau').

female('Sarah').     female('Hagar').  female('Rebeka').
female('Anon').
```

These facts express simple properties and relations of objects. The objects or entities are represented by their names: an identifier starting with a lower-case letter, or any sequence of characters delimited by single quotes or by single back-quotes. The name of a relation follows the same syntax.

Examples for the simplest queries:[3]

```
| ?- father('Abraham','Isaac').
yes
| ?- mother('Sarah','Jacob').
no
```

[3] The `| ?-` is the Prolog prompt: the Prolog environment waits for our query. We can type our commands and queries to this prompt. A command or query is always finished by a dot and `<Enter>`. The response of the Prolog system is `yes`, if it finds that our query as a statement is implied by the program, but its response is `no` if it finds that it is not implied by it. The proof is just one step here: either we find the fact identical to the query or not.

If you want to ask: *Is there some X, whose father is Isaac?* – then you receive two solutions according to the program above. X indicates the unknown entity in the query, i.e. an existentially quantified logical variable:

```
| ?- father('Isaac',X).
X = 'Jacob' ? ; X = 'Esau' ? ;
no
| ?-
```

Note: The names of logic variables are written as identifiers starting with an upper-case letter or underscore character.

The results above come from matching goal `father('Isaac',X)` against the appropriate facts.[4] During the process of matching, the variables of the goal are substituted by the appropriate nonvariables of the fact. Such substitutions represent the solutions of the goals.

Up till now we posed atomic goals, that is atomic queries to the Prolog environment. We can form *compound goals* as conjunctions of atomic ones. An atomic query in a compound goal is called its *subgoal*. For example, let us find Abraham's daughters:

```
| ?- father('Abraham',X), female(X).
X = 'Anon' ? ;
no
```

The solutions of a goal are the common solutions of its subgoals. We can say, a subgoal solved later provides a selection on the solutions of a subgoal solved earlier (*iff* they have common logic variable(s)).[5]

In logic programming, the variables are always logic variables. They stand for unknown objects. They may be universally or existentially quantified. The variables of goals are always existentially quantified. The run of the program produces constructive answers to the query, that is, it computes possible values of its variables.

Warning: Because a logic variable stands for an unknown object, destructive assignment statements like `X := X + 1` do not have stand in pure logic programming. (For example, in `X := X + 1`, which value of X stands for the unknown object?)

16.2.2 Rules

A proper conjunction of subgoals defines a new relation. For example, the conjunction "`father('Abraham',X), female(X)`" refers to the daughters of Abraham.

[4] The question marks refer to the questions of the SICStus Prolog environment, whether we ask for another solution. Our response `;` means that we ask for that. (If we do not need further solution, we press `<Enter>`. Then the system finishes the current conversation with `yes`.) In the example above, the finishing `no` abbreviates *no more solutions*.

[5] The word *iff* abbreviates the expression *if and only if*.

Assigning a name to this new relation, we receive a *rule*. Beside facts, rules form the second class of axioms in LP (logic programming).

```
'Abraham's daughter'(X) :- father('Abraham',X), female(X).
```

This reads: X is Abraham's daughter **if** Abraham is father of `X` **and** `X` is female. (This rule does not contain the information that no one has more fathers.) The logical variables of a rule are *universally* quantified, as it is with the rule above. The general form of a rule:

$A :- B_1, B_2, \ldots, B_n.$ $\quad (n > 0)$
(A, B_1, ..., B_n are atomic formulas.)

The consequence part of a rule (above A) is the *head* of the rule. The condition part of a rule (above $B_1,\ldots,B_n$) is the *body* of the rule.

Note: The facts have only head. A *fact* can be considered a *rule with empty body* or a rule with the condition part `true`. A *proper rule* is a rule which is not a fact.

Even a fact may contain (universally quantified) logic variables. Then it is called a *universal fact*. For example, the next fact says that everybody likes Sarah.

```
likes(_anybody,'Sarah').
```

And the next one says that anything is equal to itself. (Note that this predicate, i.e. $'='/2$ is a standard built-in of Prolog. Notice also that its name can be written between its parameters. This is posible, if infix operator notation (16.9) is defined to the name.)

```
X=X.
```

In such a way, we do not have to repeat the appropriate axiom for each element of the universe of the program. Anyway, the universe of a practical logic program is usually infinite, so we are not able to do this. There is another important difference between a set of facts and the appropriate universal fact. If you ask, who likes Sarah:

```
| ?- likes(Who,'Sarah').
true ? ;
no
```

The answer `true` means that the solution found did not substitute the logical variable `Who`. We may interpret this as a generic answer:

The statement `likes(Who,'Sarah')` is true for each element of the universe, that is, the unsubstituted variable can be substituted by any element "x" of the universe, and the statement `likes(x,'Sarah')` still remains true. It is quite natural that the Prolog machine did not find any other answer, just this generic one.

A *relation* and the *predicate* defining it can be specified by its name and arity, in the form *name/arity*. (The *arity* is the number arguments (i.e. parameters) of a predicate. An *argument* is a position in the program text, where a parameter is written.)

A relation is often defined by more axioms or rules. (Remember that facts can be considered as special rules with empty or `true` body.) In such a case, this relation is the union of the relations defined by the individual rules. The scope of a logical variable is the sentence containing it. It is visible in the whole sentence, because there are neither more local, nor more global variables. (It is visible exacly in its scope, because it cannot be hidden by other variables: there is no hierarchy of scopes, unlike in first-order logic, or in block-structured languages.) For example, in the next program the relation `parent/2` is the union of the relations `mother/2` and `father/2`:

```
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).

son(X,Y) :- parent(Y,X), male(X).

daughter(X,Y) :- parent(Y,X), female(X).

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

On the other hand, the relation `son/2` is the intersection of `parent/2` and `male/1`, and we receive similarly relation `daughter/2`, too. The last rule differs from the others, because it contains a new logical variable in the rule body. In this way, it has two different, but equivalent readings:

- For each `X`, `Y`, `Z`, `grandparent`(`X`, `Y`), *if* `parent`(`X`, `Z`) *and* `parent`(`Z`, `Y`).
- For each `X`, `Y`, `grandparent`(`X`, `Y`), *if* there exists a `Z` so that `parent`(`X`, `Z`) *and* `parent`(`Z`, `Y`) hold.

16.2.3 Computing the answer

The above readings of rules correspond to the *declarative* meaning of them. This is contrasted by the *procedural* meaning, which corresponds to the run of logic programs, i.e. to the process of the constructive proof of goals. In each case we pose a query. This is the goal to be proved. Now the Prolog system has to prove the subgoals, i.e. it has to eliminate them. The proof is finished when each subgoal has been eliminated (proved), and so just an empty *conjunction of goals* has been remained. The goal may be proved in *top-down*, or in a *bottom-up* manner. There are still other strategies that can be studied in [Kow79]. In each cases, the elementary step is the unification of two atomic formulas. Unification means the calculation and application of the most general unifier substitution

(*mgu*) of the atomic formulas. This is a substitution of the variables of the formulas, making them identical. Not each pair of atomic formulas are unifiable. In order to unify them, it is necessary, but insufficient, that they must refer to the same relation (same name and arity), and they must not contain different constants (or functors) at the same position. For example, the *mgu* of the atomic formulas $p(a, b)$ and $p(X, Y)$ is $\{X = a, Y = b\}$. The *mgu* of $p(a, Y)$ and $p(X, b)$ is also $\{X = a, Y = b\}$. The *mgu* of $p(a, Y)$ and $p(X, Z)$ is $\{X = a, Y = Z\}$, although $\{X = a, Y = b, Z = b\}$ and $\{X = a, Y = a, Z = a\}$ unify them, too. But the first one is more general than the others: $\{X = a, Y = b, Z = b\} = \{X = a, Y = Z\}\{Z = b\}$ and $\{X = a, Y = a, Z = a\} = \{X = a, Y = Z\}\{Z = a\}$. Next, $p(a, b)$ and $p(b, Y)$ do not have *mgu*: a substitution like $\{a = b, Y = b\}$ is NOT possible, because only variables can be substituted. (We do not know anything about the identity of two constants.) Similarly $p(a, b)$ and $p(Y, Y)$ do not have *mgu*, because $\{Y = a, Y = b\}$ is not a substitution: a variable like Y denotes something unknown entity of the universe, but cannot refer to two or more of them. (More details about unification can be found in any introductory material about first-order logic.) Now let us consider the *top-down*, and the *bottom-up* proofs of goals.

1. The *bottom-up* proof means that the conditions of the rules are unified with facts, and so they are eliminated one by one. In this way we receive new facts from the rules, until the facts unify with the subgoals of the original goal. When each subgoals have been eliminated the original goal have been proved. However, using this *bottom-up* method it is hard to direct the proof to the goal. Therefore in practice the *top-down* method is more often used. Here we start from the goal and go to the facts. This approach has been adopted up till now in most of the logic programming systems, for example in Prolog.
2. Shortly, the *top-down* proof consists of *steps of reduction*. One step of reduction means that one of the atomic formulas (i.e. subgoals) of the goal is unified with a *fact* or with a *head of a rule*. (Just like in resolution, the sentences taking part in the unification must not share common variables. This can be ensured by the appropriate renaming of the variables of the rule [i.e. fact or proper rule] taking part in the unification.) If we unify a fact, the appropriate subgoal is eliminated from the goal. If we unify the head of a proper rule, the appropriate subgoal is substituted by the rule body. In both cases, the unifying substitution is applied to the resulting conjunction of subgoals. This is a *step of reduction*. If a sequence of such steps of reduction, i.e. a top-down proof results in an empty conjunction of subgoals, then the original goal has been proved. This is a constructive proof. During the process, substitutions of the logical variables of the clauses have been performed. The results of these substitutions referring to the variables of the original goal provide for us the objects (terms) satisfying the conditions defined by the original goal.

For example, let us suppose that given a goal which is a $Q_1, Q_2, \ldots, Q_n$ conjunction of subgoals. And given a fact A sharing no common variable with this goal. Now, if $Q_1\theta = A\theta$, where θ is the *mgu* of the atomic formulas A and Q_1, then it is enough to prove $(Q_2, \ldots, Q_n)\theta$ in order to prove the original goal. This is the first case of a *step of reduction* referring to the goal to be proved.
The other case goes as follows: Consider our original goal $Q_1, Q_2, \ldots, Q_n$, and rule $A: -B_1, \ldots, B_m$ sharing no common variable with this goal. Let us suppose that θ is the *mgu* of A and Q_1. Then it is enough to prove $(B_1, \ldots, B_m, Q_2, \ldots, Q_n)\theta$ in order to prove the original goal. This is the second case of a *step of reduction* referring to the goal to be proved.
In both cases, if the result of the step of reduction is a C conjunction of subgoals, which we prove by the substitution φ, that is $C\varphi$ is proved to be true, then $(Q_1, Q_2, \ldots, Q_n)\theta\varphi$ is proved, too. This means that the substitution $\theta\varphi$ is a solution of the original query.

There is a theorem that both of the top-down and the bottom-up method of proof is correct and complete, i.e. exactly the goals following from the program can be proved if one of these strategies is used.

However, we will prefer the *top-down* method here, because it is easier to control, and it is preferred by logic programming languages, too. Corresponding to this method, there is a *procedural reading* of rules, i.e. facts and proper rules:

- The fact A means that subgoal A can be directly solved.
- The proper rule $A :- B_1, B_2, \ldots, B_n$ means that subgoal A can be solved by solving the subgoals $B_1, B_2, \ldots, B_n$.

Note that the subgoal, and the fact or rule head are rarely identical in practice. However, they must refer to the same relation (same name and arity). They must be unifiable as well. Let us suppose, that θ is the *mgu* of the atomic formulas A and Q. Then

- the fact A means that subgoal Q can be directly solved by the substitution θ, and
- the rule $A :- B_1, B_2, \ldots, B_n$ means that subgoal Q can be solved by solving the subgoals $B_1\theta, B_2\theta, \ldots, B_n\theta$. Provided that the substitution φ is their common solution, $\theta\varphi$ is a solution of Q.

For example, the rule "`grandparent(X,Y) :- parent(X,Z), parent(Z,Y).`" has the following procedural meaning: *In order to solve goal* `grandparent(X,Y)`*, solve goals* `parent(X,Z)`*, and* `parent(Z,Y)`*.*

16.2.4 Search trees

A subgoal may be unifiable by many facts and/or rule heads of our program. Therefore the possible goal reductions may have many branches forming a tree.

The root of this tree is the original goal, its nodes are conjunctions of subgoals derived through the top-down proof processes, its edges are the steps of different reductions, and its leaves are the empty conjunctions of subgoals, the *solution leaves*, and those nodes, where the subgoal selected for reduction cannot be reduced: these are the *fail leaves*. The solutions of the original goal correspond to the solution leaves. This tree is called *derivation tree*, *proof tree*, *search tree* or *search space*.

In the example in Figure 16.1 the levels of this tree are shown by the indentation. In each step we select the first subgoal for reduction. We try to eliminate it then, unifying it with a fact, or substitute it with the body of an appropriate rule (after unifying the subgoal with the head of the rule). This is the *leftmost subgoal selection strategy*. In the search tree we show only the substitutions referring to the variables of the actual goal. They are called *output substitutions*. They are shown in the form *variable*$<-$*substituting_term*.

For simplicity, if during the unification of a subgoal and a rule head or fact two variables must be unified, we always substitute the variable of the subgoal into the variable of the rule head or fact.

It is clear that in the steps of reduction where we select the first subgoal of the actual goal, we could choose another subgoal, and we would receive different search trees, if we applied different subgoal selection strategies.

The question is this: Whether we would receive different solutions or not? Fortunately, it is true that although the different subgoal selection strategies lead to different search trees, but each search tree contains the same solutions. Unfortunately, the different search trees usually have different sizes, as it can be checked easily by the reader, if he or she selects another subgoal selection strategy in the example in Figure 16.1. (Therefore the effectivity of the computation can be different, if our subgoal selection strategy is changed.)

The leftmost subgoal selection strategy has produced a search tree with minimal size here. However, if we posed the query `grandparent(X,'Jacob')`, the minimal search tree would be produced by a strategy selecting always the *rightmost* subgoal.

Notice that if we pose a query of the kind `grandparent(X,Y)`, after the first step of reduction we receive a goal like `parent(X,Z)`, `parent(Z,Y)`. Next it is better to choose the subgoal containing less variables. This method is often useful, even in other programs, because a subgoal containing less variables often leads to a search tree with fewer branches. But this is just heuristics.

In general, (when we allow recursive rules) it may happen that one search tree of the same query is finite, while the other is infinite. (We will see examples later.) In this case the search may go to an infinite branch, which leads to infinite computation. This means that we should choose a subgoal selection strategy producing finite search trees, if it were possible. Fortunately, if our rules are nonrecursive, the search tree is clearly finite.

```
?- grandparent('Abraham',X).
    parent('Abraham',Z),parent(Z,X).
       mother('Abraham',Z),parent(Z,X). % fails
       father('Abraham',Z),parent(Z,X).
         { Z <- 'Isaac' }
           parent('Isaac',X).
             mother('Isaac',X).      % fails
             father('Isaac',X).
               { X <- 'Jacob' }      % 1. solution
               { X <- 'Esau' }       % 2. solution
         { Z <- 'Ishmael' }
           parent('Ishmael',X).
             mother('Ishmael',X).    % fails
             father('Ishmael',X).    % fails
         { Z <- 'Anon' }
           parent('Anon',X).
             mother('Anon',X).       % fails
             father('Anon',X).       % fails
```

Figure 16.1: search tree of query `grandparent('Abraham',X)`

16.2.5 Recursive rules

Let us suppose that we want to describe the following relation.

```
ancestor(X,Y) :- X is ancestor of Y.
```

It is clear that this new relation is a generalization of the union of the relations `parent`/2, `grandparent`/2, '`great − gandparent`'/2, and so on:

```
parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).

'great-grandparent'(X,Y) :- parent(X,Z), grandparent(Z,Y).
```

After all, X is ancestor of Y iff X is parent of Y or X is parent of some ancestor of Y. This means that there are two cases and the second one is recursive. Therefore this may be expressed by a non-recursive and a recursive rule:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

This is declaratively simple, but the question is, if there are infinite search trees

or not. And how to select the appropriate subgoal selection strategy to ensure the finiteness of the search tree.

For example, let us suppose temporarily, that the next fact (and only the next fact) defines the relation `parent/2`.

```
parent('First','First').
```

Now let us ask about the descendants of First, and look at the rightmost branch of the search tree (applying the leftmost subgoal selection strategy):

```
?- ancestor('First',X).
         parent('First',Z1), ancestor(Z1,X).
           { Z1 <- 'First' }
             ancestor('First',X).
                parent('First',Z2), ancestor(Z2,X).
                  { Z2 <- 'First' }
                    ancestor('First',X).
                        ...
```

Clearly, it is infinite. And it is infinite with any subgoal selection strategy.

Now, let us reconsider the relation `parent/2`. We either consider the original relation or the temporary one, it defines a directed, finite graph. Its edges are given by the relation `parent/2`, and its nodes are the endpoints of the edges. The temporary graph consists of a trivial loop, but the original one is acyclic. Then the relation `ancestor/2` corresponds to nonempty, finite paths in the graph. These paths in the temporary graph are looping, but in the original one they contain no cycle, and their length has an upper limit. (These later properties of the original graph are based on the earthly nature of relation `parent/2`.) Now, let us omit the temporary definition of `parent/2`.

And let us consider the query `ancestor('Abraham',X)`. It is easy to see, that it is useful to apply the leftmost subgoal selection strategy: In this case in the subgoals of the form `parent(X,Y)` the first parameter will be always known. This property makes the search tree more slim. And it is more important, that the search tree will be finite, because in each recursive call we process one additional edge of a directed, finite path starting from node `'Abraham'`.

Now let us consider the query `ancestor(X,'Isaac')`. In order to generate the search tree, it may seem useful to choose the rightmost subgoal selection strategy, because the second parameter of the rightmost subgoal will be always a constant. However, the rightmost subgoal will be always a recursive call, and we generate an infinite search tree. If we prefer the second rule in goal reduction, we find its infinite branch immediately. If we prefer the first rule, first we find the solutions, and then go to the infinite branch. In general, a computation does not know, when it has found the last solution. Therefore this can be troublesome.

However, if we reconsider the query `ancestor(X,'Isaac')`, and we prefer the *leftmost* subgoal selection strategy, then the first `parent(X,Y)` call selects an edge from the graph, and then the searching of the path to the node `'Isaac'` goes on

from its right endpoint. Therefore, the search tree remains finite, although it will be probably fatter, than with the previous query. And it can be seen similarly, that here, the search tree of any query of the form `ancestor(X,Y)` is finite.

One might suggest another subgoal selection strategy now: given a conjunction of subgoals, we should select a subgoal defined by a nonrecursive predicate. This strategy is often useful, but there are some cases, when it does not help:

```
ancestor00(X,Y) :- parent(X,Y).
ancestor00(X,Y) :- ancestor00(X,Z), ancestor00(Z,Y).
```

This definition of relation `ancestor00/2` is logically equivalent with predicate `ancestor/2`. However, a query of the form `ancestor00(X,Y)` always has an infinite search tree, regardless of the actual parameter values, regardless of the subgoal selection strategy.

In addition, the more complex a subgoal selection strategy is, the harder it is to follow its behavior, to prove the finiteness of the corresponding search tree, and to calculate the effectivity of our logic program.

After all, we can conclude that good formulation is more essential than sophisticated subgoal selection strategy. The formulation of a logic program must match the expectable queries, and ensuring this is easier if the subgoal selection strategy is simple.

16.3 Introduction to the Prolog programming language

Therefore the invertors of the Prolog programming language decided in favor of the leftmost subgoal selection strategy.

The program is performed by the Prolog machine, which is a virtual machine. It may work as an interpreter or emulator, but it may be built into an independent, executable file, too.

The search tree is traversed by backtracking, but its branches are not built up in advance, and the actual branch is always destroyed when Prolog backtracks from it. In such a way, always just one branch of the search tree is stored in the call stack containing the call frames of the predicate invocations and the choice points for the alternative branches. In this way, Prolog tries to minimize the memory needed by the run of the program. The facts and rules are tried in their order.

Prolog serves for **Pro***gramming in* **log***ic*. In reality, Prolog is a very high level, general purpose programming language, and it is not a tool for automated theorem proving. Its logic formulas and its inference machine are too simple for the later purpose. Yes, the Prolog machine is simple enough to be controlled by the programmer, so that he or she can prove the finiteness of the search tree, and calculate the time and space complexity of the program.

The correctness of a Prolog program is always related to goals to be solved. Partial correctness simply means correct formalization. In order to prove the

termination of the program it is enough to prove, that the search trees of the possible goals are finite.

If there is no (directly or indirectly) recursive rule in the program, then the finiteness of the search trees is a trivial statement. If there is a (directly or indirectly) recursive predicate, then we consider the possible goals invoking it, and prove the finiteness of the related search trees: usually we define a terminator function whose value is a nonnegative integer for each node of the tree, and its values are strictly decreasing while going down in the tree. These properties make sure that value of the function at the predicate invocation is an upper bound of the depth of the related search tree.

For example, in the case of the predicate `ancestor/2` the length of the unprocessed path is an appropriate terminator function (16.2.5). Surely, this is decreasing by processing a subgoal of the form `parent(X,Z)`, and it cannot be negative. Therefore, the search tree is finite, and the predicate invocation terminates.

However, in the case of the predicate `ancestor00/2` the search tree is infinite (16.2.5), and there is no terminator function.

A Prolog system normally offers the user an interactive programming environment, using an internal or external text editor.

It is typical that the Prolog system and a standard text editor (for example Emacs) communicates through an interface supported by both of them, and the Prolog environment starts inside the text editor, benefiting from the services of it, like automatic indentation and text coloring of the source code, highlighting the match of different kinds of brackets, compiling and loading programs, source level debugging, and so on.

The Prolog environment normally starts in a special console window (although it may have a GUI). We type our commands and queries at the Prolog prompt of this console, and we can read the standard output of the Prolog system on it. If we want to make an executable file [Car12], the program must contain at least one directive (a goal to be performed while loading the program). This or these provide a primary control of its execution. A source program may consist of many files. We can variously load the program files into the Prolog environment. If we want to run the program in an interpreted way we can use the `consult(File)` command: we can type it at the Prolog prompt (?−), where `File` must be a Prolog source file. (Traditionally, `.pl` is the de facto standard extension of Prolog sources, and this extension can be omitted.) Any predicate of the program loaded can be queried.

For example, let us suppose that the Prolog predicates defined in this book are in the source file `lpp.pl`.

```
| ?- consult(lpp).
% consulting c:/documents and settings/pl/book/lpp.pl...
% consulted c:/documents and settings/pl/book/lpp.pl
%          in module user, 0 msec 8 bytes
yes
| ?- grandparent('Sarah',GrandChild).
GrandChild = 'Jacob' ? ;
GrandChild = 'Esau' ? ;
no
| ?- ancestor(A,'Jacob').
A = 'Rebeka' ? ;
A = 'Isaac' ? ;
A = 'Sarah' ? ;
A = 'Abraham' ? ;
no
```

Therefore, any predicate of the program loaded can be directly tested. We need no testbed.[6]

16.4 The data structures of a logic program

The data structures of a logic program are called *terms* (like in mathematical logic). According to ISO Prolog, a Prolog term can be a logical variable called `var` (referring to an unknown term) or a partially or properly known term called `nonvar`.

A `nonvar` can be a constant called `atomic`, or a structured term called `compound`.

An `atomic` term can be a name constant called `atom`, or it can be a `number`.

A `number` can be an `integer` or a `float`. (The syntax of numbers will be familiar to C, C++ or Java programmers.)

A name constant or `atom` is syntactically an identifier starting with a lowercase letter, a sequence of characters between quotes or backquotes, special character sequences of the characters $+ - * / \backslash \wedge < > = : . ? @ \# \& \$$ (for example: $=<$, $@>=$, $?-$, $*\$$, etc.), or one of the following extras: ; (called *or*, *else*, or *elsif*), ! (called *cut*), [] (called *nil*), and {} (called *empty*).

The name of a variable is syntactically an identifier starting with an uppercase letter or underscore sign. A variable denotes an unknown entity, like the variables of mathematical equations and formulas. They are very different from the variables of procedural (OOP) languages like Pascal, C, C++, and Java. The aim of the computations is the determination of the possible values of the variables of the queries, like in the case of the mathematical equations. Therefore

[6] In LP languages, like Prolog, any data structure can be described at the source code level, and even the private predicates of a module can be accessed.

it is not reasonable, and it is not possible to write assignment statements. When a logical variable has been subtituted by a `nonvar` term, then it cannot be distinguished from the substituting term, except if the program backtracks before the substitution, when the variable looses its value. This may be strange for a programmer used to procedural programming languages, but after some practice in logic programming it becomes natural. As we work with variable substitutions instead of assignment statements, we get rid of the most dangerous source of programming mistakes.

Summary: A `var` is a logical variable which has not been substituted by a `nonvar` (on the actual branch of the search tree).

It is a tradition that a variable the value of which is important is denoted by an identifier starting with an upper-case letter, but a variable the value of which is not important is denoted by an identifier starting with an underscore sign:

```
father(SomeBody) :- father(SomeBody,_Child).
```

If there is a sentence in a source file of a logic program, and a variable of this sentence has just one occurence in this sentence, then the value of this variable is not important, because there is no place where to pass its value.

Note: Therefore, many Prolog environments suppose that an identifier starting with an upper-case letter denotes an important variable. If there is a sentence in the source code with just one occurence of such a variable, then it sends a warning, in order to help us to get rid of typing mistakes.

On the other hand, if we type a query at the Prolog prompt, and it contains a variable starting with an underscore sign, then its value is not automatically printed when the query has been solved:

```
| ?- grandparent(GrandParent,GrandChild), male(GrandParent).
GrandChild = 'Jacob', GrandParent = 'Abraham' ? ;
GrandChild = 'Esau', GrandParent = 'Abraham' ? ;
no
| ?- grandparent(GrandParent,_GrandChild), male(GrandParent).
GrandParent = 'Abraham' ? ;
GrandParent = 'Abraham' ? ;
no
```

There is the *anonymous variable* consisting of just an underscore sign, the occurrences of which are different logical variables, even inside a single sentence. Therefore it can denote only unimportant variables. *The scope of any other variable is the sentence containing it.* For example:

```
father_and_son(SomeBody) :-
   father(SomeBody,_), parent(_,SomeBody).
```

In order to store structured information we can use *compound terms* of the form $f(t_1, ..., t_n)$, where f is an `atom`, $t_1, ..., t_n$ are arbitrary terms, and the *function symbol* or *functor* is f/n, which unites the terms $t_1, ..., t_n$ into a single compound term (although this strict form is sometimes relaxed by so called *syntactic sugars*). Therefore the functors are specified by their *name/arity* pairs, like relations and predicates. Any parameter t_i may be `atomic`, `var`, or `compound`, too.

A term which is not `compound`, is called a *simple term*. This means that a *simple term* is `atomic` or `var`.

A *callable term* is a `compound` or an `atom`, because these terms may represent the goals (queries) of a program.

A *ground term* is a term containing no `var`. Recursively, a ground term is an `atomic`, or a `compound` with ground term parameters.

Therefore the compound terms may represent recursive data structures like lists and trees. For example, an empty tree may be represented by the *atom* [], and a nonempty tree by the *compound term* .(*root*, $\mathtt{t_1}$, ... , $\mathtt{t_n}$),
where each t_i denotes a direct subtree.

Now a list is a unary tree. For example, a list of the numbers 1, 2, and 3 is .(1, .(2, .(3, []))). (Syntactic sugars will make the notation more convenient, especially here.)

And the tree .(4, .(2, .(1, [], []), .(3, [], [])), .(5, [], [])) is a binary search tree with depth two.

Surely, we may choose any other notation. The important thing is to apply it consistently. For example, the empty tree may be denoted by the atom `o`. The functor name of the nonempty trees may be the atom `t`, and, especially in binary trees, it may be convenient to put the root in the middle, in the form `t`(*leftSubTree*, *root*, *rightSubTree*).
Using this notation, the previous binary seach tree looks like
t(t(t(o, 1, o), 2, t(o, 3, o)), 4, t(o, 5, o)).

16.5 List handling with recursive logic programs

We have seen, that we can represent lists as unary trees. Now, before discussing the basic list handling predicates, let us see the different kinds of lists and list-like structures.

A *list* may be *proper* or *partial*. A *non-list* is a term which is not a list. A *proper list* may be *empty* or *nonempty*. The standard representation of the *empty list* is the atom [] read as *nil*. The standard constructor of a nonempty list is $'.'/2$. A term .(`X`, `Xs`) is proper list *iff* `Xs` is proper list.[7]

[7] In this chapter, the two-letter identifiers `As, Bs, Cs, ..., Xs, Ys, Zs` usually refer to lists.

According to this, the list [1,2,3] can be written as .(1,.(2,.(3,[]))). Fortunately, we have three standard syntactic sugars to allow an easier-to-read notation. (The notation convention implied may seem too complicated at the first glance, but it is easy to learn and useful.)

In order to introduce these syntactic sugars, still we need some definitions:

`X == Y` means that the terms `X` and `Y` are *identical*. (Actually, `'=='/2` is a built-in predicate in Prolog with the same meaning.)

A *list-term* is a compound term with the main functor `'.'/2`. (Therefore, the notion of list-term is a generalization of the notion of nonempty list: The list .(1,.(2,.(3,[]))) is a list-term, and the non-list .(1,.(2,.(3,non_nil))) is a list-term, too.)

`X` is the (first) head and `Ys` is the (first) tail of the term `T`, *iff* `T == .(X,Ys)`.

`X` is the $(i+1)^{\text{th}}$ head and `Ys` is the $(i+1)^{\text{th}}$ tail of term `T`, *iff* `T == .(A,Bs)`, and `X` is the i^{th} head and `Ys` is the i^{th} tail of term `Bs`, where i is a positive integer number. The i^{th} head of a term is also called its i^{th} element.

Now let us see the standard syntactic sugars allowing the easier-to-read notation.

1. A list-term `.(X,Xs)` can be written as `[X|Xs]`.
2. The front elements of a list-term `T` can be separated by commas: if `X1,X2,...,Xn` are the first `n` heads of `T`, and `Ys` is its $\texttt{n}^{\text{th}}$ tail, then `T == [X1,X2,...,Xn|Ys]`.
3. "`|[]`" may be omitted in `[X1,X2,...,Xn|[]]`.

In this way

```
| ?- .(X,Xs)==[X|Xs], [X1|[X2|Xs]]==[X1,X2|Xs],
    [X1,X2,X3|[]]==[X1,X2,X3],
     .(1,.(2,.(3,[])))==[1,2,3].
true
```

An element of a list is an arbitrary term. It may be even a variable or a list. For example, `[a|[_]]` is a proper list, and `[[_]|[_]]` is also a proper list, but `[[_]|a]` is not a list, because the atom `a` is not a list.

A term is *partial list,* iff it is a `var` or it is a term of the form `[X|Xs]`, where `Xs` is a partial list. A `var` is an empty partial list.

Therefore a nonempty partial list always has the form `T == [X1,X2,...,Xn|Vs]`, where `Vs` is a `var`. And a term is partial list, *iff* it is not a proper list but after an appropriate substitution it becomes a proper list. Actually, if the `var` at the end of a partial list is substituted by a proper list, then the partial list becomes a proper list. If the `var` at the end of a partial list is substituted by another partial list, then the first partial list still remains a partial list. If the `var` at the end of a partial list is substituted by a non-list, then the partial list becomes a non-list, because a list-term is a non-list, *iff* at its end

there is a `nonvar` different from the empty proper list, that is []. No substitution maps a non-list to a list.

Conventionally, iterative data structures are represented by lists in Prolog, and there are a lot of built-in and library predicates supporting list handling in the different implementations.

At last, a *ground list* is a variable free list, i.e. a list which is a ground term. Therefore the ground lists form a proper subset of proper lists.

For example, let us suppose that Xs is a `var`. Now `Xs` and [1, 2|`Xs`] are *partial lists*, but [`Xs`] and [1, 2, `Xs`] are proper lists of one and three elements. Neither of these four lists is a ground list. However, if `Xs` == 3, then neither `Xs` nor [1, 2|`Xs`] is list, but [`Xs`] and [1, 2, `Xs`] are ground lists.

Consider the next predicate.

```
list([]).
list([_X|Xs]) :- list(Xs).
```

If the goal `list(Ys)` is parameterized by a proper list, it will be successful with no variable substitution. Parameterized by a partial list, this goal will have infinite search tree, and infinite number of solutions. These solutions will be the most general proper list samples of `Ys`. For example:

```
| ?- list(Ys).
Ys = [] ? ;  Ys = [_A] ? ;  Ys = [_A,_B] ? ;
Ys = [_A,_B,_C] ? ; Ys = [_A,_B,_C,_D] ? <Enter>
yes
```

The goal `list(Ys)` parameterized with a non-list will fail.

Now, using the notations above, let us consider some list handling predicates. And we discuss some useful programming methods used in logic programs which handle lists and recursive data structures in general.

We need a definition: The *precondition of a predicate* specifies the set of goals allowed to invoke that predicate. For example, we may prescribe that the first parameter must be a proper list, the second parameter must be positive integer, and so on.

16.5.1 Recursive search

First let us consider the classic predicate `member_`/2.[8] [9]

```
% PreCond: Xs is a proper list.
% member_(X,Xs) :- X is a member of list Xs.
member_(X,[X|_Xs]).
member_(X,[_X|Xs]) :- member_(X,Xs).
```

[8] The character `%` followed by any sequence of characters up to the end of the line is comment.

[9] `member/2` and `append/3` are the built-in counterparts of our `member_/2` and `append_/3` (16.5.2) in many Prolog implementations. Therefore we cannot redefine them.

So the members of a list are its head, and the members of its tail. Invoking this predicate, the appropriate item is found directly, or through recursive calls. Therefore this programming method is called *recursive search.*

Let us consider the precondition of `member_(X,Xs)`: If Xs were a partial list, the search tree of the query would be infinite. If it is a proper list, its length is strictly decreasing throughout the recursion. This guarantees the finiteness of the search tree. For example:

```
| ?- member_(X,[1,2,3]).
X = 1 ? ;   X = 2 ? ;   X = 3 ? ;   no
| ?- member_(2,[1,2,3,X,4]).
true ? ;   X = 2 ? ;   no
```

Notice that also `ancestor/2` applies recursive search.

16.5.2 Step-by-step approximation of the output

The next predicate is the standard predicate for appending or splitting lists. Notice that we have two rules: One of them refers to the case when the first parameter is an empty list, and the other handles nonempty lists in the first argument, recursively, while the length of a proper list parameter is strictly decreasing through the recursion, guaranteeing the finiteness of the search tree. Therefore this is quite a usual way of organizing list handling predicates in Prolog.

```
% PreCond: Xs or XsYs is a proper list,
%          the other two parameters are lists.
% append_(Xs,Ys,XsYs) :-
%   appending lists Xs and Ys we receive list XsYs.
append_([],Ys,Ys).
append_([X|Xs],Ys,[X|Zs]) :- append_(Xs,Ys,Zs).
```

Appending the empty list and any other lists we receive the other list. Appending a nonempty list `[X|Xs]` and another list `Ys`, the head of the connected list is `X`, while the tail of the connected list is the result of appending `Xs` and `Ys`. Notice that in the recursion the length of the first parameter, and the length of the third parameter is also decreasing. Therefore, the precondition above is strong enough: If one of the first and third parameters is a proper list, the search tree is finite. (However, if both of them are partial lists, the search tree is clearly infinite.) The second parameter may be proper or partial list. Notice that if the first parameter is a non-list, the call will surely fail: this case is not handled. And if the second or third parameter is a non-list, we may fail or receive non-list results.

Consider now the search tree of query `append_([1,2],[3,4],Zs)` in Figure 16.2. (We suppose that the LP system renames each variable `V` of a rule to `V`i, where i is the sequence number of the actual step of the goal reductions.)

```
append_([1,2],[3,4],Zs)
  { Zs <- [1|Zs1] }
    append_([2],[3,4],Zs1)
      { Zs1 <- [2|Zs2] }
        append_([],[3,4],Zs2)
          { Zs2 <- [3,4] }          % (solution)

% After all:
Zs=[1|Zs1]=[1|[2|Zs2]]=[1,2|Zs2]=[1,2|[3,4]]=[1,2,3,4]
```

Figure 16.2: Search tree of query `append_([1,2],[3,4],Zs)`

```
| ?- append_([1,2],Ys,Zs).
Zs = [1,2|Ys] ? ;
no
| ?- append_(Xs,Ys,[1,2,3]).
Xs = [], Ys = [1,2,3] ? ;
Xs = [1], Ys = [2,3] ? ;
Xs = [1,2], Ys = [3] ? ;
Xs = [1,2,3], Ys = [] ? ;
no
```

Figure 16.3: Queries of `append_/3`

Notice that the search tree is linear (unary) tree here. Everywhere just the head of one of the rules have unified the goal. On the other hand, the appended proper list has been approximated in more steps, through partial lists. In each steps, we substituted the `var` tail of the actual partial list, computing bigger and bigger part of the output list, until we received the appended proper list as a result.

Here we have built up the result data structure in a top-down manner. First, we built its topmost level, but we left some details undefined. Then these details were refined by variable substitutions in the same manner again and again. Therefore, this way of computing the result is called the *step-by-step approximation of the output.*

Exercise: Try to draw the search tree of both of the queries in Figure 16.3. Observe the process of the approximation of the output.

In the first query the result is a partial list: its front is equal to the first parameter, and its tail is the second parameter, whatever it may be.

In the second query we receive a search tree with branches, and the results (at the leaves of the tree) are the possible cuts of the list (in the third parameter) into two parts.

16.5.3 Accumulator pairs

Accumulators are used to build the result in a bottom-up manner. Their usage is shown through the following example.

```
% PreCond: Xs is a proper list, Ys and Zs are lists.
% rev_app(Xs,Ys,Zs) :-
%     Xs reversed and appended before Ys provides Zs.
rev_app([],Ys,Ys).
rev_app([X|Xs],Ys,Zs) :- rev_app(Xs,[X|Ys],Zs).
```

The empty list reversed and appended before another list provides this other list. The nonempty list [X|Xs] reversed and appended before another list provides Xs reversed and appended before [X|Ys].

The length of the first parameter strictly decreases through the recursion. Therefore, if Xs is a proper list, the search tree of the query rev_app(Xs, Ys, Zs) is finite. If Xs is a partial list, the search tree is clearly infinite. Both of Ys and Zs may be proper or partial list, especially var. Let us consider the search tree of the query rev_app([1, 2], [3, 4], Zs).

```
rev_app([1,2],[3,4],Zs)
  rev_app([2],[1,3,4],Zs)
    rev_app([],[2,1,3,4],Zs)
      { Zs <- [2,1,3,4] }            % solution
```

The resulting data structure is built up in the second argument, in bottom-up manner, through the recursion. Therefore the second argument is an *accumulator*. The result is completed at the bottom of the recursion. In order to return the result we need a third argument, a *tunel*: it is passed through the recursion without change, and it is substituted by the result at its bottom. An accumulator and a tunel argument form an *accumulator pair*, because they are found in the logic programs always in pair.

In practice, we often write a predicate containing more accumulator pairs, and/or parameters calculated by step-by-step approximation.

For example, notice that in the query ? − append_(Xs, Ys, [1, 2, 3]) the first parameter is calculated by step-by-step approximation. The second argument is a tunel, while the third one behaves like a so called negative accumulator: the data structure to be returned through the tunnel is not built up, but appropriately pulled down in it.

16.5.4 The method of generalization

The next predicate is an example of *generalization*. In such a case the problem is generalized, solved, and its program is used to handle the original, more specific problem. This method is often used in human problem solving, especially in programming, and even more specially in LP. It is often used when we write a recursive procedure: It is able to solve a more general problem, and it is appropriately parameterized when it is called:

```
% PreCond: Xs is a proper list.
% reverse(Xs,Ys) :- the reverse of Xs is Ys.
reverse(Xs,Ys) :- rev_app(Xs,[],Ys).
```

The reverse of list `Xs` is `Ys`, if `Xs` reversed and appended before the empty list provides Ys.

Therefore, this predicate inherits the condition of finiteness from `rev_app/3`: If the first parameter of goal `reverse`$(\texttt{Xs}, \texttt{Ys})$ is a proper list, then the search tree of this goal is finite. But if `Xs` is a partial list, then this search tree is infinite.

For example, if `Xs` is a `var`, then the invocation `reverse`$([1,2,3], \texttt{Xs})$ calculates its only solution $\texttt{Xs} = [3,2,1]$, and its search tree is finite. Naturally, the query `reverse`$(\texttt{Xs}, [1,2,3])$ has the same, only solution. But its search tree is infinite.

Especially, in the Prolog predicate `rev_app/3`, the fact finishing the recursion precedes the recursive rule. Therefore, we will find the solution of the query `reverse`$(\texttt{Xs}, [1,2,3])$. Nonetheless if we instruct the Prolog environment to find another solution, it always backtracks, and tries to unify longer and longer lists of `var`s with the second parameter of the goal, but fails.

Let the reader explain the following behavior now.

```
| ?- reverse([X,Y,Z],[1,2,3]).
X = 3, Y = 2, Z = 1 ? ;
no
```

The four methods discussed above are used while handling recursive data structures in LP, especially lists and trees.

16.6 The Prolog machine

Full Prolog contains meta-logical and extra-logical constructs, too. Up till now we have omitted these extensions and concentrated on *pure Prolog*, which is the core of the language: it is based on pure mathematical logic. In this section we provide a detailed explanation of the abstract interpreter performing pure Prolog programs. However, we remain mainly at the abstract level, and explain just a few implementation details. We try to help Prolog programmers rather than Prolog implementers.

A pure Prolog program is a set of the Prolog programmer's predicates, and a query or goal to be answered or solved: the predicates manifest our knowledge in the topic. Each predicate is a set of rules at the declarative level, and a sequence of rules at the procedural level. Each rule is a fact or a proper rule. No rule body or query contains any subgoals referring to a built-in predicate.

16.6.1 Executing pure Prolog programs

Executing a program, the Prolog machine performs preorder traversal of the search tree (16.2.4) of the query. During the traversal just the actual branch of the tree is stored. This branch is the path from the root to the actual node. A node of this branch is labeled by the appropriate goal, by the predicate its first subgoal refers to, and by the untried rules of this predicate in their original order. The edges of the actual branch represent the goal reduction steps leading to the actual goal. An edge is labeled by the rule used in this step of reduction, and by the variables of the parent goal substituted in the connected unification. Notice that we do not have to remember the unifying substitutions. See the algorithm of the execution of Prolog programs coming here. We use the word *matching* instead of *unifying*. This will be explained in (16.6.2).

1. In the beginning, the root of the search tree is labeled only by the original query (goal). Now, the root is the actual node, and only this root is stored. (Each time the node furthest from the root on the actual branch is the actual node.)
2. If the actual node is labeled by the empty conjunction of subgoals, we have found a solution: this is the actual substitution of the variables of the original query. In this case we print the solution, and ask the user, whether he or she asks for another solution.
   If so, we continue from (9). (Backtracking.)
   If not, *we have finished the search successfully.*
3. If the actual node is labeled by a nonempty goal, let us consider its first subgoal. Let us label this node also with the predicate referred to by this subgoal, and with the list rules of this predicate. (Note that if this list contains two or more rules, it forms a *choice point*, which remains *living* while it is nonempty.) Now let us rename the variables of the rules, so that they share no variable with the goal. (It is the most effective to generate completely new variable names.)
4. If the actual list of clauses is empty, we have arrived at a dead end, that is, at a fail node, and we continue form (9). (Backtracking.)
5. Let q be the first item of the actual list of rules. Delete q from this list. (If the list becomes empty, the choice point possibly generated above lives no more.)
6. Try to match the first subgoal of the actual goal with the head of q (see 16.6.2).

7. If this matching fails, we continue from (4).
8. If this matching is successful, we perform a step of reduction: The variables substituted during the matching are appropriately substituted everywhere in the goal and in the rule. (This is usually ensured by a linked representation of the variables.) The actually stored branch of the search tree is extended by a new node. The edge leading to this node is labeled by the actually substituted variables of the actual node, and by q. In order to compute the first label of the new node, we substitute the first subgoal of the actual goal with the (possibly empty) body of q, where the matching substitution has already been performed on each participants. Next the new node becomes the actual node. Then we continue from (2).
9. *Backtracking:* If the actual node is the root of the tree, then the execution of the program finishes with *fail*.
   Otherwise, we delete the actual node of the actually stored branch of the search tree, and its parent will be the actual node. During this the substitution of the variables labeling the edge between this two nodes are also deleted. Then we continue from (4).

16.6.2 Pattern matching

We have seen that the basic operations of the Prolog machine are goal reduction and backtracking. And the key of a step of reduction is the unification of the actual subgoal and rule head. Theoretically, the Prolog machine should compute the *mgu* of them (see Section 16.2.3); it should try to unify atomic formulas with the same name and arity. Therefore, it should unify two sequences of formulas of the same length. Procedurally, a predicate is a procedure, and this unification is a kind of parameter passing. Now, we face the problem that computing the *mgu* is too expensive to be used as parameter passing. Provided that it is adopted, logic programs become unacceptably slow. Therefore, a simplified unification called *pattern matching* is adopted in Prolog:

1. If the too sequences of terms are empty, We are ready. Otherwise, we match the pair of the first elements of the two sequences according to (2). Then we continue with matching the rests of the two sequences according to (1).
2. Provided that we have two `atomic` terms (see 16.4), they match, *iff* they are identical. If they are identical, they match without variable substitution. If they are different, then this algorithm of pattern matching *fails*.
3. If one of the terms is `atomic`, but the other one is compound, then this algorithm of pattern matching fails.
4. If both of them are `compound` terms, and their functor names and arities are the same, then their sequences of parameters are matched according

to (1). If both of them are `compound` terms, but their functornames or arities are different, then this algorithm of pattern matching fails.

5. Provided that we have two `var` terms, any of them may be substituted by the other one. (However, in our examples, we will always substitute the `var` term of the rule head with the `var` term of the goal, because it is easier to follow, and it usually allows a more effective implementation.)
6. If just one of the terms is a `var`, then we substitute it with the other term, which is an `atomic` or a `compound`.

The variable substitutions of the algorithm above are performed on each occurrence of these variables in the goal and rule taking part in the actual step of reduction.

We keep a record of each substitution of the variables of the goal. If the pattern matching eventually fails, the substitutions of the variables of the goal are deleted according to our records. If the pattern matching succeeds, then the appropriate edge of the actual branch of the search tree is labeled by this recorded set of variables (16.6.1).

In the first point of this algorithm, the pattern matching of the appropriate term pairs of the two term sequences may be performed in arbitrary order, or even in parallel.

Notice that in point (6) of the algorithm of pattern matching above we substitute a `var` with a `compound` unconditionally. Nonetheless in the original algorithm of unification, this is allowed *iff* the `compound` does not contain the `var`. The obligatory check of this condition before a `var` is substituted by a `compound` is called *occurs check* ([FGN90], [Kow79], [DEDC96], [ISO95] and [SS94]). The Prolog machine omits the occurs check for reasons of efficiency, because its operational complexity is $O(size(compound))$, and the size of the `compound` may be extremely large in practice.

For example, consider the built-in predicate $'='/2$ (see Section 16.2.2, i.e. fact "$\mathtt{X} = \mathtt{X}$."). Query $\mathtt{Y} = \mathtt{f(Y)}$ should fail, at least according to mathematical logic. However, according to a typical Prolog implementation the solution of the query is the substitution $\{\mathtt{X} < -\mathtt{f(Y)}, \mathtt{Y} < -\mathtt{f(Y)}\}$. According to mathematical logic, *this is not a unifying substitution*. According to the Prolog standard, the result is undefined, if during the pattern matching a `var` faces a `compound` containing it ([DEDC96] and [ISO95]). Our algorithm of pattern matching is more special than that of the Prolog standard ([DEDC96] and [ISO95]), wich may even abort or go to an infinite loop in this situation. First we substitute `X` with `Y`, and then `Y` with `f(Y)`. Therefore, we avoid the explicit computation of the unifying substitution. In this way, we avoid infinite loops and program aborts, and we may say (following Colmerauer [Col82]), that the substitution $\{\mathtt{Y} < -\mathtt{f(Y)}\}$ generates a cyclic term $\mathtt{Y} = \mathtt{f(Y)}$. Some Prolog implementations even give us tools to handle cyclic terms [Car12]. However, we will not take benefit of such tools here, and even try to avoid generating cyclic terms, because

it may be difficult to port a program working with cyclic terms between Prolog platforms.

After all, in programming practice it is extremely rare that we face the problem that Prolog omits occurs check. Most of the applications work well even if we are not aware of the possibility of the unhappy situations following this omission. One must admit that this omission is one of those brilliant compromises which make Prolog a practical programming language: Without this compromise the cost of unifying a `var` and a `compound` would be proportional to the size of the `compound`, and this cost would arise in each predicate invocation where we work with structured terms. Many applications work with long lists and/or big trees. Then the cost of a simple procedure call would be proportional to the length of the list or size of the tree. However, in a typical Prolog environment a `var` and a structured term are matched with constant cost which is independent from the size of the structure.

After all, the Prolog standard handles the occurs check problem in an elegant and effective manner. And now we are going to discuss this topic.

16.6.3 NSTO programs

In NSTO programs the *occurs check* can be omitted safely. Now, we consider how to write programs where even the original algorithm of unification would perform no successful occurs check. Such predicate invocations, that is subgoals are called *NSTO* (Not Subject To Occurs check), otherwise we speak of *STO* subgoals. If each of the subgoals of a program is *NSTO*, then the program is also *NSTO*. Otherwise the program is *STO*.

Note that these subgoals may occur in the query at the Prolog prompt, in the bodies of the rules of the program, and in the directives of the program. A subgoal may refer to a programmer's predicate, or to a built-in predicate, too.

Based on [DM93], the following simple, sufficient condition of the NSTO property of a subgoal is suggested here, which is sophisticated enough for most practical cases.
If a subgoal g referring to a predicate p satisfies any of the following conditions, then this subgoal is NSTO.

1. g does not contain double `var` (a `var` with more than one occurrence).
2. No head of any rule of p contains double variable.
3. The actual parameters of g are *simple terms*, and each rule head of p
   - contains just *simple terms* in its formal parameters,
   - or does not contain double variable.

Provided that we want to write an NSTO program, the conditions above call our attention to the critical rule heads. If a head of a rule contains double variable, and some subgoal referring to it contains double `var`, and the rule head or this subgoal contains `compound` parameter, then this goal may be STO, and then the whole program may be STO.

If we find a rule head like this, we may still prove the NSTO property of the goals referring to it, considering their special features. Or we can follow the method shown in the next example. Let us suppose that we coded the predicate `member_/2` in the usual way:

```
% member_(X,Xs) :- X is member of list Xs (?NSTO?).
member_(X,[X|_Xs]).
member_(X,[_X|Xs]) :- member_(X,Xs).
```

Clearly, the first rule may imply STO property. Its head contains the double variable `X`, and even the compound `[X|_Xs]`. Considering programming practice, in this case the first sufficient condition of the NSTO property is almost always satisfied, as the subgoals of the form `member_(Y,Ys)` contain no double `var`.

However, if some goal of this form might contain double `var`, and we cannot prove that this goal is NSTO (or it is surely STO), we can produce a safe version of `member_/2`, and make this goal refer to it:

```
% safe_member(X,Xs) :- X is member of list Xs (NSTO)
safe_member(X,[Z|_Xs]) :- unify_with_occurs_check(X,Z).
safe_member(X,[_X|Xs]) :- safe_member(X,Xs).
```

In predicate `safe_member/2` goal `unify_with_occurs_check(X,Z)` invokes the appropriate built-in predicate of Prolog, which calculates the *mgu* of `X` and `Z`. (If they do not have unifier, it fails.) For example:

```
| ?- member_(X,[f(Y,Y),f(X)]).
X = f(Y,Y) ? ;
X = f(f(f(f(f(f(f(f(f(...)))))))))) ? ;
no
| ?- safe_member(X,[f(Y,Y),f(X)]).
X = f(Y,Y) ? ;
no
| ?- member_(f(X,X),[f(Y,g(Z)),f(Y,g(Y))]).
X = g(Z), Y = g(Z) ? ;
X = g(g(g(g(g(g(g(g(g(...)))))))))),
Y = g(g(g(g(g(g(g(g(g(...)))))))))) ? ;
no
| ?- safe_member(f(X,X),[f(Y,g(Z)),f(Y,g(Y))]).
X = g(Z), Y = g(Z) ? ;
no
```

In general, if we use compound terms together with a predicate, then we consider the rules containing double variables in their head. If we cannot prove the NSTO property of the goals referring to them, we make the following transformation on these rules: while we find double variable (critical from the point of NSTO property) in the head of the rule (let it be `X`), we rename one of its

occurrences to a fresh variable (let it be `Z`), and then we insert the subgoal `unify_with_occurs_check(X,Z)` as the first subgoal of the body of the rule.

In this way each predicate invocation of our program becomes NSTO, except the subgoals of the form `unify_with_occurs_check(X,Z)` generated by the program transformation above. Then our Prolog program works as if everywhere mathematical unification were used, but it runs much faster, because mathematical unification is used only when it is necessary. Otherwise, we use its much simpler and much more effective version: pattern matching.

Note: Considering the built-in predicates of Prolog, in most cases the subgoals referring to them are automatically NSTO. Selecting those mentioned in this work, the list of exceptions is the following: `(=)/2` and `(\=)/2` (16.7), `arg/3` (16.8.3), `read/2` (16.10.2), `retract/1` and `retractall/1` (16.10.3), `findall/3` (16.11), and `catch/3` (16.12).

The NSTO property of the subgoals referring to the built-in predicates listed here can be checked and handled with a method similar to the one shown here. For example, if `built_in(X,X)` is an unsafe subgoal, in many cases it can be replaced by `built_in(X,Z)`, `unify_with_occurs_check(X,Z)`, where `Z` is a fresh variable. (It might cause a problem that the search tree of the call `built_in(X,Z)` may be (much) bigger than that of `built_in(X,X)`.)

In the next two subsections we go on with two *de facto standard* optimizations of the Prolog machine. Taking them into consideration, we can significantly reduce the runtime and memory needs of our Prolog programs.

16.6.4 First argument indexing

During goal reduction, first argument indexing may significantly reduce the costs of searching through the rules of the appropriate predicate. According to a rough criteria it selects a subset of the rules of a predicate, appropriately narrowing the search tree. It often selects just one rule, so increases the efficiency dramatically, as generating a choice point and handling it is quite expensive.

However, the predicate invocations of the Prolog programs are often deterministic. This means that just one rule head matches the goal.[10] Provided that the Prolog machine recognizes the determinism of a predicate invocation, it generates no choice point. First argument indexing supports it to recognize determinism. Let us see the details: Consider point (3) of the abstract Prolog interpreter described in (16.6.1). We labeled the actual node of the search tree with the list of each of the rules of the predicate to be invoked there. Provided that this list contains more than a single rule, we generate a choice point there,

[10] A predicate invocation is called *nondeterministic,* iff more rule heads match it, because in this case the run of a logic program can continue on the different branches of the search tree, although a standard Prolog environment resolves this nondeterminism with backtracking on these branches.

where the search tree branches according to rule heads possibly matching the predicate invocation.

Clearly, it would be enough to put the rules whose head match the subgoal onto this list. However, it would be too expensive to produce this list before we try the branches. Therefore, most Prolog implementations apply a compromise here. This is called *first argument indexing*:

1. Provided that the first actual parameter of the predicate invocation is `atomic` term, only those rules are put onto the list, where the first formal parameter of the rule head is the same `atomic` or a `var`.
2. Provided that the first actual parameter of this invocation is `compound` term, only those rules are put onto the list, where the first formal parameter of the rule head is a `var`, or a `compound` whose functor is identical with the functor of the first actual parameter of the subgoal.
3. Provided that the first actual parameter of this invocation is a `var`, each rule of the predicate to be invoked is put onto the list.

The possible lists of rules are usually generated compile time, and a table is made of them. Then the run time selection of the appropriate list needs minimal, constant time. The variable renaming of the rules is solved like in the case of procedural languages. On the other hand, multiple argument indexing is rarely used, because the size of the table to be generated becomes too large.
Let us suppose that any of the predicates `list/1`, `append/3`, `rev_app/3` (16.5) is called with proper list actual parameter in its first argument. Then neither its call nor its recursive calls generate any choice point while it runs. Namely, if the first actual parameter is an empty list, we will index on the first rule; and if it is a nonempty list, we will index on the second rule. The run of predicate `member_/2` is not affected by first argument indexing, because the first formal parameters of its rules are variables.

16.6.5 Last call optimization

Normally, each predicate invocation needs some space in the call stack. However, when this optimization is applied, the called predicate can reuse the memory needed by the caller. Eventually, a constant size of memory may be enough to run a recursive procedure.

In order to understand how it works, consider point (5) of the abstract Prolog interpreter described in (16.6.1). If the actual list has contained just one rule there, then after removing this single rule, an empty list of clauses remains. This means that a later backtracking to this node results in another backtracking from here. Therefore in point (8), instead of extending the momentarily represented part of the actual branch of the search tree, the actual node may be overwritten. The set of variables labeling the edge pointing to this node is extended by the set of variables which would label the edge pointing to the new node in the original

algorithm. Therefore, in this case the actually represented branch of the search tree does not become longer.

Therefore, given the last subgoal of a rule body, consider the call of this subgoal. If there is no living choice point from the node and time, where and when the predicate containing this rule was invoked, than that node of the optimized representation of the search tree is equal to the node which is the result of invoking this last subgoal. Therefore, in this case this last subgoal can reuse the memory needed by its parent subgoal (i.e. the invocation of the predicate containing this last call). This memory reuse is called last call optimization.

Especially, if this last subgoal invokes the predicate containing it, this is *tail recursion*, and it can work like a loop in procedural languages. Then last call optimization can be simplified to the so called *tail recursion optimization.*

For example, because during the run of the predicates

`list/1`, `member_/2`, `append_/3`, and `rev_app/3` (16.5)

there is no living choice point when they are called recursively, in each case tail recursion optimization can be applied. However, when predicate `reverse/2` invokes `rev_app/3`, last call optimization can be applied.

16.7 Modifying the default control in Prolog

Language pure Prolog contains no possibility to express any form of negation. For example, it is easy to express that two terms can be matched (`A = B`), because we can define it with the universal fact `X = X`. However, we have no tool to express its negation (`A\ = B`).[11]

Similarly, we have defined predicate `member_/2` in (16.5.1), but we have no tool to define predicate `nonmember_/2`. Now it follows, that still we are not able to phrase the union or intersection of two unsorted lists, because we should say what to do when an element of one of the lists is not found on the other list.

In general, our rules are not suitable to derive negative information from our programs. At best, we can decide if a statement follows from our program or not. For example, if the search tree of a query is finite, then we can decide, whether it follows from the program.

For example, given a ground term `X`, and a ground list `Xs`, the search tree of the query `member_(X, Xs)` is finite. Therefore we can decide, if the statement `member_(X, Xs)` follows from our program or not. But this statement is true, *iff* it follows from the program. In practice, there are many similar statements. (For example, consider the queries referring to the list handling predicates of (16.5).) Therefore, if we could tell what to do when a goal were successful, and what to do when it failed, we would have a limited but practical tool to manage negation.

[11] In fact, `(=)/2`, and `(\=)/2` are built-in predicates ([DEDC96] and [ISO95]) of standard Prolog, and we cannot overdefine them. When we use them, we can write them in infix manner, i.e. between their parameters (16.9).

16.7.1 Disjunctions

To define this limited but practical negation, the first step is to introduce disjunctions. We already have implicit disjunction:

```
sibling(X,Y) :- sister(X,Y).
sibling(X,Y) :- brother(X,Y).
```

However, in Prolog we have explicit disjunction, too:

```
sibling(X,Y) :- sister(X,Y) ; brother(X,Y).
```

There is a strong tradition here: the ";" sign used to express explicit disjunction is written with *French spacing*, i.e. there is a blank before it. In addition, it is never put at the end of a line, but at the beginning of the next line.

Both of conjunction ",", and disjunction ";" are right associative. Conjunction "," binds stronger than disjunction ";", so the parentheses in the next example are essential.

```
son(Y,X) :- ( mother(X,Y) ; father(X,Y) ), male(Y).
```

However, the overuse of disjunctions is discouraged in Prolog. A rule should be as simple as possible. The earlier definition in (16.2.2) is more structured, and in many implementations it performs a bit better.

16.7.2 Conditional goals and local cuts

Perhaps the most important control structures are the *conditional goals*. They support structured programming, and define negation as failure. Their basic form is the following:

(*if* −> *then*
; *else*
)

Here the *if*, the *then*, and the *else* parts are arbitrary Prolog goals. The *if* part is the decisive condition. If goal *if* is successful, then the solutions of the conditional goal are the solutions of goal *then*. Otherwise the solutions of the conditional goal are the solutions of goal *else*.

It follows that it is not possible to backtrack into goal *if*. If it is successful, the Prolog machine will calculate just its first solution. For example:

```
% PreCond: Xs and Ys are ground lists.
% union(Xs,Ys,Zs) :-
%     Those members of Xs which are not members of Ys
%     concatenated in order before Ys produce list Zs.
union([],Ys,Ys).
union([X|Xs],Ys,Zs) :-
   ( member_(X,Ys) -> union(Xs,Ys,Zs)
   ; Zs = [X|Us], union(Xs,Ys,Us)
   ).
| ?- union([1,2,3,4,5],[1,3,5],Us).
Us = [2,4,1,3,5] ? ;
no
| ?- union([1,2,3],[1,1,3,3,5],Us).
Us = [2,1,1,3,3,5] ? ;
no
```

The second test shows that just the first solution of condition `member_(X,Ys)` is considered. If we forget the arrow, this happens:

```
malfunctioning_union([],Ys,Ys).
malfunctioning_union([X|Xs],Ys,Zs) :-
   ( member_(X,Ys), malfunctioning_union(Xs,Ys,Zs)
   ; Zs = [X|Us], malfunctioning_union(Xs,Ys,Us)
   ).
| ?- malfunctioning_union([2,3],[1,3,3,5],Us).
Us = [2,1,3,3,5] ? ;
Us = [2,1,3,3,5] ? ;
Us = [2,3,1,3,3,5] ? ;
no
```

The first solution is found twice, because goal `member_(3,[1,3,3,5])` finds number `3` on list $[1,3,3,5]$ twice. The second solution was found, because goal `member_(X,Ys)` is not a decisive condition here. Instead of a conditional goal, we coded just a disjunction.

In the conditional goals, the arrow "$->$" is a *local cut*, and the semicolon is the operator of disjunction. Considering the three operators forming compound Prolog goals, their (decreasing) priority order is "$, \ -> \ ;$". And each of them is right associative. Procedurally speaking, when a disjunction of the form
(*if* $->$ *then* ; *else*)
is called, a choice point with two alternatives is generated. Its first alternative is goal *if* $->$ *then* containing the local cut, and its second alternative is goal *else*. Provided that goal *if* succeeds, the local cut is performed, and it cuts the choice points (if any) left by goal *if*, and the choice point left by the disjunction.

Then goal *then* is called, and there is no possibility to backtrack neither into goal *if* nor to goal *else*. Provided that goal *if* fails, we backtrack, remove the choice point left by the disjunction, and call goal *else*. However, if any of goals *then* or *else* succeeds, it is still possible to backtrack into it, and find its alternative solutions. Because the operator ";" is right associative, goals
(`if1 -> then1 ; if2 -> then2 ; else`), and
(`if1 -> then1 ; ( if2 -> then2 ; else ) )` are equivalent.
We may also omit the " ; `else` " part. Then " ; `fail` " is the default, where `fail/0` is a standard built-in predicate of Prolog, and it always fails. Similar is `true/0`, but it always succeeds.

Now we can solve the problems posed at the beginning of this section (16.7):

```
% \+=(X,Y) :- X does not match Y.
\+=(X,Y) :- ( X=Y -> fail ; true ).

% PreCond: Xs is a proper list.
% does_not_have(Xs,Y) :- no member of Xs matches Y.
does_not_have([],_Y).
does_not_have([X|Xs],Y) :-
   ( X = Y -> fail
   ; does_not_have(Xs,Y)
   ).

% nonmember_(X,Xs) :- does_not_have(Xs,X).
nonmember_(X,Xs) :- ( member_(X,Xs) -> fail ; true ).

% PreCond: Xs and Ys are gound lists.
% intersection(Xs,Ys,Zs) :-
%    those members of Xs which are found also on Ys,
%    form in order proper list Zs.
intersection([],_Ys,[]).
intersection([X|Xs],Ys,Zs) :-
   ( member_(X,Ys) ->
        Zs = [X|Ms], intersection(Xs,Ys,Ms)
   ; intersection(Xs,Ys,Zs)
   ).
```

Notice that the order of the arguments of the recursive predicates
`does_not_have/2`, `intersection/3`, and `union/3`
is chosen to exploit first argument indexing (16.6.4). Tail recursion optimization can be applied to each recursive call (16.6.5).

Note on assignment statements

In predicates `intersection/3`, and `union/3` any goal of the form `Zs = [X|Vs]` tries to *match* its two parameters. Namely, standard Prolog does *not* have any form of *assignment statement*, because any logic variable refers to an unknown term (16.4), [DEDC96], [ISO95], [O'K90] and [SS94]: it refers to an object unknown when the program is coded, but it refers to an object to be computed while the program is being executed. Therefore, when a variable is substituted by a `nonvar`, its value (determined by the relation coded in our program) is computed, and it does not have sense to overwrite it. (If there were an assignment statement like `Xs := [Y|Xs]`, which occurence of `Xs` would stand for the unknown object?)

A typical error of a programmer starting to study logic programming is the use of Prolog goals like `Xs = [Y|Xs]`. This goal will surely fail to perform the assignment desired. For example, if `Xs` is a proper list, the pattern matching between `Xs` and `[Y|Xs]` will clearly fail. If Xs is a `var`, the goal is STO, and a cyclic list will be generated, which is not the desired case now. If `Xs` is a partial list, it may fail or generate a cyclic term wich is not the behaviour desired.

How to get just one solution from a query?

At last we show two solutions for checking whether a term is member of a list. Namely, goal `member_(X,Xs)` may have many solutions, even if `X`, and `Xs` are ground terms (provided that `Xs` has multiple occurrences of `X`). After producing a solution, goal `member_(X,Xs)` always leaves a choice point which needs extra memory, and may prevent last call optimization. However, if we need just member checking, we need a goal with at most one solution which does not leave choice points.

Our first solution, predicate `member1/2` shows that a conditional goal can help us even to get rid of unwanted choice points. The second one, predicate `member_check/2` shows that failing rules should not be programmed explicitly. If we want to emphasize that we have not forgotten those cases, it is better to put the failing rules into comment, in order to keep efficiency.

```
member1(X,Xs) :- ( member_(X,Xs) -> true ).

% member_check(_X,[]) :- fail.
member_check(X,[Y|Ys]) :-
    ( X = Y -> true
    ; member_check(X,Ys)
    ).
```

16.7.3 Negation and meta-goals

Let us suppose that predicates `student/1`, and `married/1` are given, and we want to define predicate `unmarried_student/1`. Based on the previous subsection, the following solution springs up.

```
unmarried_student(X) :- student(X), unmarried(X).
unmarried(X) :- ( married(X) -> fail ; true ).
student('Peter'). student('John').
student('James').
married('Peter').   married('Joseph').
```

Notice that the common scheme of predicates
(`\ =`)/2, `nonmember_/2`, and `unmarried/1` follows.

```
not(P) :- ( P -> fail ; true ).
```

The question is, whether this more general scheme of negation is a Prolog predicate or not.

Considering it in a context free manner a Prolog goal is a callable term, i.e. a `compound` term or an `atom`. This makes it possible to compute and call it with the program: The program part computing it handles it just like any `compound` or `atom`. Then it is called like a *meta-goal*. A trivial example: goal `X = p(Y), X` is equivalent with goal `p(Y)`.

Considering a *meta-goal* in the source code in context-free manner, it is a logical variable. But in context-dependent manner, it is a goal: it is part of the body of a rule, query, or directive. The important thing is that by the time it is invoked, it must be substituted by a callable term which can be interpreted as a valid goal of the program. (The appropriate predicates must be defined.)

A meta-goal appearing in the body of a rule may be formal parameter of the head of the rule, like in predicate `not/1` above. The appropriate argument of the head of that rule is a *meta-argument*. A predicate consisting of such rules is a *meta-predicate*. When a meta-predicate is invoked, its meta-arguments can be parameterized by goals. The parameter in a meta-argument is called *meta-parameter*.

For example, predicate `unmarried_student/1` can also be defined as follows.

```
unmarried_student(X) :- student(X), not(married(X)).
```

The only formal parameter of meta-predicate `not/1` above is `P` which is invoked in the rule body as a meta-goal. If `P` (for example `married(X)` in the previous example) is successful, the *local cut* cuts its choice points (if any), and also cuts the *else* branch of the conditional goal. Then on the *then* branch goal `fail` is performed and invocation `not(P)` (for example `not(married(X))`) fails. If `P` (for example `married(X)`) fails, we backtrack to the *else* branch of the conditional goal, goal `true` succeeds, and also invocation `not(P)` (for example `not(married(X))`) succeeds. Summarizing this, goal `not(P)` succeeds, *iff* goal `P`

fails (and therefore there is no solution of goal `P`). And goal `not(P)` fails, *iff* goal `P` succeeds.

This is *negation as failure*. It is *not* a logical negation, but it can be implemented quite effectively,and usually it can be used instead of that, if applied by some care.[12]

Notice the first four properties of goal `not(P)`:

1. It never leaves any choice points.
2. Independently from goal `P`, goal `not(P)` never substitutes its variables.
3. `not(P)` succeeds, *iff* the search tree of `P` is finite, and it contains no solution.
4. `not(P)` fails, *iff* the search tree of `P` contains some solution, but there is no infinite branch before the first solution.

(Notice that these observations are valid even if the negation is implicit, when it is programmed with conditional goals like in the like in the cases of predicates (\ =)/2, `nonmember_/2`, and `unmarried/1`.)

It is clear from (2) that a negated goal produces no solution. One cannot use them to produce any (partial) solution, just to test and validate (partial) solutions. According to this, it is not all the same, where the negated goal is inside a compound goal: the desired behavior of the negated goal usually assumes that its variables have been substituted. For example:

```
| ?- unmarried_student(X).
X = 'John' ? ; X = 'James' ? ; no
```

This is the desired behavior, independently from the actual one of the two definitions of predicate `unmarried_student/1` above. However, if we exchange the order of the subgoals in its body:

```
| ?- unmarried_student(X).
no
```

Namely, in this case goal `unmarried(X)` or `not(married(X))` fails, because `married(X)` succeeds (provided that `X` is still `var`).

In general, it is a sufficient condition of the logical soundness of Prolog negation that the negated goal must be ground when it is invoked, and its search tree must be finite.[13]

[12] Let us note that although goals `not(married(X))`, and `unmarried(X)` mean the same, in today's Prolog implementations usually `not(married(X))` is the less effective one, because most often the metagoals cannot be optimized while we compile the Prolog source code.

[13] Considering negation we adopt the *closed world assumption*. This means, we suppose that we have enough information: each statement of the actual model which does not follow from our axioms is *false*.

This negation defined with predicate `not/1` is implemented in Prolog more effectively with the built-in predicate (\+)/1. It can be used as a prefix operator like in the next version of predicate `married_student/1`.

```
unmarried_student(X) :- student(X), \+ married(X).
```

16.7.4 The ordinary cut

The local cut introduced in Subsection (16.7.2) is considered usually a better alternative of the more traditional ordinary cut to be discussed here. However, there are millions of lines of Prolog code with ordinary cuts around. In order to allow the reader to understand such code, we are going to present them through a simple example now.

Let us suppose that we want to define the relation `max(X,Y,Z)`, where X and Y are integer values given in advance, and it is true iff Z matches their maximum. Procedurally speaking, if X and Y are integer numbers, the invocation `max(X,Y,Z)` tries to match Z with the maximal one. Let us suppose that the arithmetic relations $'>'/2$ and $'=<'/2$ are given as usual. Our "zeroth" solution does not use any cuts:

```
max0(X,Y,X) :- X > Y.
max0(X,Y,Y) :- X =< Y.
```

Notice that in the first rule we have condition `X > Y` instead of `X >= Y` in order to avoid duplicated solutions. Clearly, this program is correct. But it is not effective: if X and Y are integers, X is greater than Y, Z matches X, and `max0(X,Y,Z)` is invoked, the query will be successful with the first rule of the predicate, and it leaves a choice point referring to the second rule.[14] Thus, if this call is element of a sequence of invocations, and we backtrack to it from a later one, we know that it will fail. Therefore, in this case the subgoal `max0(X,Y,Z)` leaves an *unnecessary choice point* which needs extra space, its handling eats extra runtime, and it may prevent last call optimization(s) in the program using this predicate.

In order to solve the problem of *unnecessary choice points*, the inventors of Prolog introduced a control tool, the (ordinary) *cut* statement into the language. It is denoted by "!". Let us see the improved version of predicate `max/3`:

```
max1(X,Y,X) :- X > Y, !. % green cut after test selects the rule
max1(X,Y,Y) :- X =< Y. % contra test
```

The cut (i.e. "!") statement is a procedural construction: a predicate is considered as a procedure. When it is called, and while it runs, it may generate many choice points, many branches of the program. When the cut is invoked, it tells Prolog that the actual branch of the predicate containing it is the only possible winner. Therefore, the other branches are pruned. We inform Prolog that if any, the rule

[14] For example, consider the subgoal `max0(3,2,M)` where M is a `var`.

containing the cut will perform the calculation associated by the predicate, and in the rule body the subgoals before the cut serve to test this selection. Prolog replies and prunes each choice point generated since the call of the predicate:

1. If a choice point was left when the predicate containing the cut was invoked, and it exists, then it is pruned.
2. All the choice points left by the subgoals preceding the cut in the actual rule body or query are pruned.

Let us consider predicate invocation `max1(X,Y,Z)` where X and Y are integers and Z is a `var`. If `X > Y`, the cut is called, and it prunes the choice point left by invoking the `max1/3`. The test `X > Y` did not leave a choice point. But if it left any choice point, the cut would prune it, too. If the test `X > Y` fails, Prolog backtracks from the first rule, automatically prunes the choice point left by the predicate invocation, and selects the second rule. Then the contra test `X =< Y` succeeds. There is no cut statement after the contra test, because there is no choice point to be pruned. In both cases the predicate invocation succeeds, and it does not leave any choice point, reflecting the fact that there is just one maximum of two numbers.

However, this cut has been inserted into a correct predicate, and it does not alter its semantics. It only implies some optimization in the code. Therefore, it is called a *green cut*. We remain close to "pure logic programming": first we write the program without cuts, and then we insert the green cuts necessary to prune the redundant choice points.

Let us reconsider the predicate invocation `max1(X,Y,Z)` where X and Y are integers. Notice that the first rule fails even if `X > Y`, provided that Z is a `nonvar` different from X. The second rule fails even if `X =< Y`, provided that Z is a `nonvar` different from Y.

Nevertheless, some Prolog predicates contain *red cuts* modifying their semantics. Considering the code of predicate `max1/3`, one may think that the contra test is superfluous: "if `X > Y` fails, and we go to the second rule, it is just waisting time to test whether `X =< Y` stands, because it is surely true". Then the following solution may be proposed.

```
max2(X,Y,X) :- X > Y, !. % red cut
max2(_X,Y,Y).            % contra test omitted, procedural code
```

This is a red cut: without the cut, even goal `max2(3,2,M)` (where M is a `var`) would have two solutions, namely `M = 3` and `M = 2`. With the cut, the only solution is `M = 3`. Therefore, predicate `max2/3` is no more a logic program, just a procedure written in Prolog. There is one even more serious problem. Provided that X and Y are integers, goal `max2(X,Y,Z)` should succeed iff Z matches their maximum. However, the reasoning which is the base of predicate `max2/3` implicitly supposes that condition `X > Y` is always evaluated. Nonetheless it is true only if Z is a `var` or Z is identical with X. Otherwise query `max2(X,Y,Z)` tries the

first rule, but its head does not match it and fails.[15] Then it tries the second rule without evaluating `X > Y`. For example, goal `max2(3,2,2)` succeeds. In general, simply omitting the contra test is dangerous.

Clearly, if we want to spare the contra test, we can use a conditional goal:

```
max3(X,Y,Z) :-
    ( X > Y -> Z = X
    ; Z = Y
    ).
```

Unlike here, the contra test is often expensive, and such solutions can save runtime. Fortunately, if one uses a conditional goal, rules representing different cases are melted into a single one, and the programmer must have formal parameters general enough to cover each case. In this way, the condition is evaluated, and the contra test can be safely omitted, except if the condition part contains more than the condition selecting the appropriate branch of the conditional:

```
max4(X,Y,Z) :-
    ( X > Y, Z = X -> true. % WRONG SOLUTION!!!
    ; Z = Y
    ).
```

This predicate tries to match the output before the local cut, and this matching becomes part of the condition. So we have the same problem with the predicates `max2/3` and `max4/3`: if Z does not match X, the second alternative is tried, independently from condition `X > Y` (although there is no early failure here).

But this kind of error is quite rare. Even beginners intuitively use the conditional goals in the correct way. The main disadvantage of using conditionals is that we have to restructure the "pure logic programs" in order to include the local cuts. Although most programmers do not feel this to be a problem.

Nevertheless, sometimes ordinary cuts can be preferred. Therefore, we will show the safe use of ordinary red cuts. Reconsidering the conditional goals used here and in (16.7.2) it is easy to see that the local cuts are usually red cuts: the control test in the next branch of the disjunction is normally omitted. And the output matching is done normally after the local cut. This is useful even with ordinary cuts. Therefore let us see the correct use of ordinary red cuts:

```
max5(X,Y,Z) :- X > Y, !, Z = X. % output matching after the cut.
max5(_X,Y,Y).      % contra test omitted: procedural, but safe.
```

Clearly, this solution prevents early failure. Condition `X > Y` really selects the appropriate clause. The method can be used in any "deterministic" predicate for pruning unnecessary choice points while avoiding redundant contra tests. (A predicate is called *deterministic* iff for any given data just one branch of it can succeed.)

[15] This is called *early failure*.

The general rule of inserting cuts is this: *cut as early as possible*, i.e. where it is already known that the actual branch of the program is the only candidate to perform the actual computation. If we cut earlier, we select that branch before the tests necessary to select it have been completed. If we cut later, we may try alternative branches of the program, even when these branches should not run.

16.8 The meta-logical predicates of Prolog

In practical Prolog programming we often need information about the actual state of the variables and other components of our program, but such questions cannot be formulated with the tools we already have. We often have to compute the numeric value of an expression, but numeric computations still cannot be performed effectively with these tools. Also we have to make symbolic computations on terms with unknown functors, i.e. terms coming from external sources, but we are able to formulate rules on terms with known functors only. In order to solve such problems, we need meta-logical predicates.

16.8.1 Arithmetic

The usual symbols of arithmetic operations, like $+, -, *, /, **$ etc., are only function symbols in Prolog, and these symbols can be written especially in infix or prefix mode. Therefore they are only constructors of compound data structures, and they force no computation. This is necessary because Prolog supports symbolic computations, for example, the derivation or integration of polynomials like $-\mathtt{x}^2 + 3 * \mathtt{x} - 4$. It follows that, for example, the expression $2 + 3$ means just the Prolog term $+(2, 3)$, and the Prolog environment does not evaluate it, except if it occurs in an *arithmetic argument* of an invocation of an *arithmetic predicate*. These are:

- The right-hand side argument of the built-in predicate `is/2`;
- And both arguments of the arithmetic comparison predicates
  . $(=:=)/2$, $(= \backslash =)/2$, $(<)/2$, $(>)/2$, $(=<)/2$, $(>=)/2$.

When we invoke any of these predicates its name can be used with infix notation. Goal `Term is Exp` first evaluates the arithmetic expression `Exp`, next it tries to match the result with `Term`. An arithmetic comparison compares the arithmetic values of its parameters.

An arithmetic expression is a number (`integer` or `float`), or a compound term with arithmetic functor, and arithmetic expressions in the arguments. (The arithmetic functors are the usual arithmetic operators, logarithmic, trigonometric function symbols etc. [Car12], [DEDC96] and [ISO95]) This means that the variables of the *arithmetic arguments* of a Prolog goal must be appropriately

substituted by the time of invoking this goal. For example (// denotes the integer divison, `floor` denotes the integer part):

```
| ?- X is -2**4, Y is floor(cos(0))+3*(8-7//2), Z is -2+1.0.
X = 16.0,  Y = 16,  Z = -1.0
| ?- X = -2**4, Y = floor(cos(0))+3*(8-7//2), Z = -2+1.0.
X = -2**4,  Y = floor(cos(0))+3*(8-7//2), Z = -2+1.0
| ?- 5 is 2+3, 5.0 =:= 2+3, -4+1 =< -2-1.0.
yes
| ?- 2+3 is 2+3.
no % 5 does not match 2+3 which is a compound.
| ?- 5.0 is 2+3.
no % 5 does not match 5.0 although they are arithmetically equal
```

A typical error of Prolog beginners is the goal "`X is X + 1`". Nonetheless this goal never performs the operation desired. If `X` has been substituted by a number, the value of `X + 1` does not match `X`, because these are two different constants. If `X` is a compound arithmetic expression, the value of `X + 1` does not match `X`, because a constant never matches a compound. If `X` is not an arithmetic expression (for example, `X` is a `var` or `atom`), the evaluation of `X + 1` raises the appropriate exception (16.12).

Note that the *arithmetic comparisons* just compare the values of proper arithmetic expressions. For example, goal `Y =:= X + 1` cannot be used to get the result of `X + 1`. A goal like `Y is X + 1` serves for this purpose:

```
| ?- Y is 6+1.
Y = 7
| ?- Y =:= 6+1.
{INSTANTIATION ERROR: _157=:=6+1 - arg 1}
```

Finally, we consider some classic arithmetic computations. The next two predicates calculate the sum and scalar product of two vectors represented by lists of numbers of the same length.

```
% add(V1,V2,V) :-
%     Vector V is the sum of vectors V1 and V2.
add([],[],[]).
add([X|Xs],[Y|Ys],[Z|Zs]) :-
   Z is X+Y, add(Xs,Ys,Zs).


% mult(V1,V2,S) :-
%     S is the scalar product of vectors V1 and V2.
mult([],[],0).
mult([X|Xs],[Y|Ys],S) :-
   mult(Xs,Ys,S0), S is S0+X*Y.
```

The predicates above are deterministic according to first argument indexing. In case of `add/3` even tail recursion optimization can be applied, and it is not hard to transform `mult/3` accordingly: we introduce an accumulator in order to collect the partial sums, that is, we generalize the original problem in order to add the scalar product to some initial value.

```
mult(V1,V2,S) :- mult(V1,V2,0,S).

% mult(V1,V2,A,S) :- S = A+V1*V2.
mult([],[],A,A).
mult([X|Xs],[Y|Ys],A0,S) :-
    A1 is A0+X*Y, mult(Xs,Ys,A1,S).
```

16.8.2 Type and comparison of terms

The standard types of Prolog terms are organized according to the hierarchy of the different kinds of terms introduced in (16.4).

The list of standard types: `var`, `nonvar`, `atomic`, `number`, `float`, `integer`, `atom`, `compound`.

The names of these types are the names of the standard type-checking predicates, too. Each of them has arity 1. For example, the following test is successful.

```
| ?- var(X), var(Y), var(Z), var(U), X=1, Y=1.0, Z=a, U=f(a),
    nonvar(X), nonvar(Y), nonvar(Z), nonvar(U), compound(U),
    atomic(X), atomic(Y), atomic(Z), atom(Z),
    number(X), number(Y), integer(X), float(Y).
```

In the next example the effectivity of predicate `grandparent/2` (16.2.2) is increased: If the grandson is known, and the grandparent is to be computed, we exchange the order of the subgoals, in order to narrow the search tree. Otherwise, we leave the original order and take advantage of first argument indexing.

```
grandparent(X,Y) :-
   ( var(X), nonvar(Y) -> parent(Z,Y), parent(X,Z)
   ; parent(X,Z), parent(Z,Y)
   ).
```

In Prolog two arbitrary terms are comparable. The identity of two terms can be tested with the built-in `(==)/2` and `(\==)/2`. For example, the following test is successful, because the two logical variables are originally different, but having matched them they are identical: `| ?- X \== Y, X = Y, X == Y.`

In the **standard order of terms** a `var` is *smaller* than a `float`, this is *smaller* than an `integer`, this is *smaller* than an `atom`, and this is *smaller* than a `compound`. goal `A @< B` is successful **iff** `A` precedes `B` in the standard order of terms, that is `A` is *smaller* than `B`

In the standard order, the *floats* are compared arithmetically, and the *integers*, too. But do not forget that a `float` is always *smaller* than an `integer`:

```
| ?- 5 < 5.1, 5.1 @< 5, 9.9e99 @< -999999999.
yes
```

The names or *atoms* are compared lexicographically. And their characters are compared according to their code values, in the actual coding system (for example, latin-1, utf-8, etc.). The structures or *compounds* are compared first according to their *arities*, next according to their *functor names*, and at last according to their parameter lists, lexicographically. The parameter pairs are compared according to the standard order of terms, recursively.

The standard order of the *variables* is implementation defined. The standard order of two terms may be changed by substitution, if any of them is a `var`, or any of them is a `compound` containing some `var`(s). For example, the following test is successful.

```
| ?- X@<10.0, X=1, X@>10.0.
```

According to their standard order the terms can be compared with the built-ins (@ >)/2, (@ <)/2, (@ >=)/2, (@ =<)/2. For example:

```
% PreCond: Ys is a proper list sorted increasingly
%         according to the standard order.
% sorted_insert(Ys,X,Zs) :-
%    Zs is received by the sorted insert of X into Ys.
sorted_insert([],X,[X]).
sorted_insert([Y|Ys],X,Zs) :-
   ( X @=< Y -> Zs = [X,Y|Ys]
   ; Zs = [Y|Us], sorted_insert(Ys,X,Us)
   ).
| ?- sorted_insert([b(z(1,2)),a(X,Y),a(2,1)],a(2,3.14),Zs).
Zs = [b(z(1,2)),a(X,Y),a(2,3.14),a(2,1)]
```

16.8.3 Term manipulation

Up till now we have supposed that the functors, i.e. constructors of the terms processed by our programs are known in advance. However, in many applications this is not true (for example, when we process Prolog terms coming from an input file or channel). In such cases the input terms can be analized and new terms can be synthetised using the built-in predicates manipulating terms.

In order to handle `compound` terms constructed with unknown functors, the most important built-ins are `functor`/3, and `arg`/3.

If `Term` is a `nonvar`, we can determine or test the functorname and/or arity of `Term` using the invocation `functor`(`Term`, `Functorname`, `Arity`). (The arity of an `atomic` is zero.) If `Term` is a `var`, `Functorname` is an `atom`, and `Arity` is a

positive `integer`, the invocation generates an appropriate `compound` term with fresh variables in its arguments and substitutes it into `Term`:

```
| ?- functor(t(a,b),F,N), functor(T,F,N).
F = t,   N = 2,   T = t(_A,_B)
```

The goal `arg(I,Structure,Arg)` matches `Arg` with the $\texttt{I}^{\text{th}}$ parameter of the `compound` term `Structure`, where `1 =< I =< Arity`. Therefore this predicate can *read* and/or *fill* the parameters of a `compound`:

```
% substitute(T0,X,Y,T) :-
%    T is a copy of T0 except that each occurence of X in T0
%    has a corresponding occurence of Y in T.
substitute(T0,X,Y,T) :-
   ( T0 == X -> T = Y
   ; compound(T0) ->
      functor(T0,F,N), functor(T,F,N),
      substitute_args(N,T0,X,Y,T)
   ; T = T0
   ).
substitute_args(N,T0,X,Y,T) :-
   ( N > 0 ->
      arg(N,T0,A), substitute(A,X,Y,B), arg(N,T,B),
      N1 is N-1, substitute_args(N1,T0,X,Y,T)
   ; true
   ).

% Test:
| ?- substitute(a(X,nil,b(nil,2,B,nil)),nil,[],T).
T = a(X,[],b([],2,B,[]))
```

Constants, i.e. `atomic` terms can be taken into pieces and put together using the built-in predicates
`atom_codes(Atom,ListOfCharacterCodes)`, and
`number_codes(Number,ListOfCharacterCodes)`.
(Further term manipulators in [DEDC96], [ISO95] and [Car12].) Using these predicates, any textual manipulation of a constant can be reduced to manipu-

lating the appropriate list of character codes. For example:

```
% aA(A,B) :- B is a copy of atom A except that
%     each lower-case letter in A is substituted
%     by the appropriate upper-case letter in B.
aA(A,B) :-
   atom_codes(A,Cs), cC(Cs,Ds), atom_codes(B,Ds).

cC([],[]).
cC([C|Cs],[D|Ds]) :- bB(C,D), cC(Cs,Ds).

bB(C,D) :- % the code of character a is 0'a
   ( 0'a=<C, C=<0'z -> D is C-0'a+0'A
   ; D = C
   ).

% Test:
| ?- aA('How beautiful is She!',Unknown).
Unknown = 'HOW BEAUTIFUL IS SHE!'
```

16.9 Operator symbols in Prolog

We have mentioned that the names of some predicates (like the names of arithmetic comparisons) can be written in infix mode. Built-in negation can be written like a prefix operator (16.7.3). And some arithmetic function symbols (like $+, -, *, /$) can be written as operator symbols. However, considering these predicate names and function symbols in a context-free manner, they are just functors of `compound` terms. This means that the notion of *operator (symbol)* in Prolog is just notational convenience. It is a completely syntactic category. A Prolog operator *does not do anything*, unlike the operators of the procedural (C,C++, etc.) and functional (Lisp, Haskell, etc.) languages. It is just a syntactic sugar, an optional notation, like the list notation, although it is very important: without operators it is hard to write easy-to-read Prolog programs.

An *operator symbol* is *predefined* or *user-defined*, and it has three basic properties: *priority*, *mode*, and *name*. Its name is a Prolog `atom`.

There are 1200 *priority* levels. The operators of the 1^{st} level bind the strongest and those of the 1200^{th} level bind the weakest. For example, the infix operators $+$, and $*$ have priorities 500 and 400. Therefore $\mathtt{a}+\mathtt{b}*\mathtt{c} == +(\mathtt{a},*(\mathtt{b},\mathtt{c}))$. Each Prolog expression has priority, which is zero, if it is a `var` or `atomic`, it is directly in brackets, it is written with list notation, or it is a `compound` written in the standard functional notation in the form $f(t_1,\ldots,t_n)$. The priority of an expression is equal to the priority of its main functor, if this functor is written as

operator, and the expression is not directly in brackets. For example, the priority of `a + b * c` is 500.

Considering roughly the *mode*, an operator can be *infix* (if it is written between the operands), *prefix* (if it is put before the operand), or *suffix* (if it comes after the operand). At the same time two operators may exist with the same name. In this case, one of them is infix, and the other is prefix or suffix.

Let us consider first the infix operators. And let us suppose that p and q are two infix operators where $pr(p)$ and $pr(q)$ are their priority levels, respectively. Let us consider the expression $apbqc$. If $pr(p) < pr(q)$ then $apbqc = (apb)qc$, because p binds stronger. If $pr(p) > pr(q)$ then $apbqc = ap(bqc)$, because q binds stronger. If $pr(p) = pr(q)$, the interpretation of $apbqc$ depends on the associativity of the operators. There are three different associativities.

The exact mode or associativity of an infix operator is
yfx (left-associative, for example `a - b - c` == `(a - b) - c`), or
xfy (right-associative, for example `(a; b; c)` == `(a; (b; c))`), or
xfx (non-associative: `a = b = c` must be parenthesized explicitly).

In this notation f symbolizes the functor (i.e. operator) name, x and y symbolize the parameters; y=yes: on this side of the operator there must be an expression with the same or smaller priority level. x=no: on this side of the operator there must be an expression with smaller priority level, and an expression with the same priority level is not allowed. However, there are some ambiguous expressions yet: if the right side of the first operator and the left side of the second one are denoted by y, too. In these cases, the expression is considered right-associative.

Based on these rules, this is the default bracketing of $apbqc$, provided that $pr(p) = pr(q)$, $m(p)$ denotes the mode or associativity of operator p, and ? stands for x or y:

- $apbqc = (apb)qc$ **if** $m(q) = yfx \wedge (m(p) = yfx \vee m(p) = xfx)$,
- $apbqc = ap(bqc)$ **if** $m(p) = xfy \wedge m(q) = ?f?$,
- $apbqc$ must be explicitly bracketed in any other case.

The exact mode or associativity of a prefix operator is fx or fy. And that of a suffix operator is xf or yf. The meaning of f, x, and y is the same as before.

It follows that **if** $pr(p) = pr(q)$,

- $pbq = (pb)q$ **if** $m(q) = yf \wedge m(p) = fx$,
- $pbq = p(bq)$ **if** $m(p) = fy \wedge m(q) = ?f$,
- pbq must be explicitly bracketed **if** $m(p) = fx \wedge m(q) = xf$.

The default bracketing of expressions like $apbq$, $pbqc$, etc. goes in a similar way.

A new operator is created by the statement `op(Priority,Mode,Name)`, or it can be overdefined in the same way.[16] (`Name` can be an `atom` , or a proper list

[16] `op/3` is an example of extra-logical predicates (16.10).

of atoms.) If `Priority` == `0`, the operator with the given name and mode is deleted. This statement is most often used in directives (see 16.2). For example:

```
:- op( 1200, xfx, [ :-, -> ]).
:- op( 1200, fx, [ :-, ?- ]).
:- op( 1100, xfy, [ ; ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]). % The priority of ',' is 1000.
:- op( 900, fy,  \+ ).
:- op( 700, xfx, [ =, \=, ==, \==, @<, @>, @=<, @>=,
              =. ., is, =:=, =\=, <, >, =<, >= ]).
:- op( 500, yfx, [ +, -, /\, \/ ]).
:- op( 400, yfx, [ *, /, //, mod, rem, «, » ]).
:- op( 200, xfx, ** ).  :- op( 200, xfy, ^ ).
:- op( 200, fy, [ -, \ ]).
```

Now we have introduced the predefined operators of Prolog according to the ISO standard ([DEDC96] and [ISO95]).

Notice that the comma symbol is a predefined right-associative infix operator, too. An extra rule: if it is used as infix operator, it must be written without apostrophes. However, commas separate the elements of a list-like term, and the expressions of a parameter list, too. Therefore, in order to know the right interpretation of the operator and separator commas in a Prolog expression, the priority of an element of a (parameter) list must be less then 1000. (If the priority of such an element is greater or equal to 1000, it must be put between brackets.

We have already known that each parameter of any atomic formula is a term in mathematical logic, logic programming, and Prolog. Notice now that the signs used to formulate the Prolog sentences ($':-'$, $','$, $';'$, $'->'$, and $'\backslash+'$) are predefined operators. This means that the goals, queries, heads and bodies of rules, and even the sentences (rules, directives, and declarations) of the Prolog programs are terms besides the parameters of atomic formulas. For example, term `p(X)` can be a piece of data, a query, a head or body of a rule, or a fact of a program depending on its context.

Another example: if negation were not a built-in predicate, we could define it with the following rule.

```
\+ Goal :-
    ( Goal -> fail
    ; true
    ).
```

In this rule, the first occurrence of logical variable `Goal` means a piece of data, especially a formal parameter, but its second occurrence denotes a Prolog goal. (Notice that in this case the brackets of the conditional goal can be omitted, but in order to increase the readability of the programs, we will always use such brackets.) Because a rule considered in a context-free manner is just a term,

using the functional notation of compound terms, the previous rule could be written as follows (although this notation would destroy readability).

```
:-(\+(Goal),;(->(Goal,fail),true)).
```

We can conclude that in Prolog there is no strict difference between program and data. This property of the language helps us to write language processors, programs manipulating other programs like interpreters, compilers, program transformators, intelligent programs which are able to learn and forget things, and so on. Surely, it is the easier case of the language processors when the sentences of the source and target languages are also fitting the term notion of Prolog ([SB04] and [War80]). (If this is not the case, we can help us with *logic grammars* [DM93], [Car12], [SS94], [SB04] and [War80].)

16.10 Extra-logical predicates of Prolog

These predicates do not have declarative reading like pure Prolog programs. These have *only* operational semantics. Theoretically, they lie outside the logic programming model [SS94]. The predicates of pure Prolog, and the meta-logical predicates communicate with their programming environment only through their parameters, and their effects are backtrackable. The extra-logical predicates access global information, usually have side-effects, and typically these side-effects are not backtrackable. Therefore they allow us to pass information among the different branches of the search tree of a Prolog query, so we need them, if we want to write practical applications. They are responsible for the program interfaces, and for the access and manipulation of the Prolog environment. The different kinds of the interface predicates are those loading (and saving) programs, the I/O predicates, the foreign language interfaces (C/C++, Java, .NET, Tcl/Tk, etc.), data base handler, GUI, operating system, web, and other interfaces ([Car12] and [SB04]).

In this chapter we consider only loading programs, I/O predicates, and manipulating the program loaded.

16.10.1 Loading Prolog programfiles

Loading a program means loading a Prolog source or object code into the Prolog environment (where it is activated through queries).

The only standard predicate of this category is `ensure_loaded/1`, whose actual parameter can be a filename, or a proper list of filenames. For each file it checks the time of its last modification, and loads it only if necessary. Some Prolog implementations may allow the use of this predicate only in *directives*, but usually it can be invoked without restrictions, even during the run of a query. It compiles the loaded code into the memory.

Most Prolog environments still provide the traditional predicates of loading in interpreted mode (`consult/1`), and in compiled mode (`compile/1`); file-to-file compilation producing object code, loading object code; there are special predicates for loading module files (`use_module/2` ... (16.13.1)), etc. ([Car12] and [SB04]).

The run of the predicates loaded in interpreted mode can traced (`trace/0`, `notrace/0`, etc.), while loading in compiled mode results optimized code 5-20 times faster ([Car12] and [SB04]).

16.10.2 Input and output

We consider only the handling of sequential textfiles, and only its basic predicates ([DEDC96], [ISO95] and [Car12]). A textfile can be opened in `read`, `write`, or `append` mode, using the query `open(File,Mode,Stream)`. Then our operations refer to the `Stream`, until we close it using `close(Stream)`. The standard I/O streams should not, and must not be closed. They can be referred to through the name `user`. The I/O predicates read or write a single character or a whole term with no restriction on the complexity of that term.

Consider first the character I/O. The goal `peek_code(S,C)` is true if the code of the actual character of stream `S` is `C` (it tries to match it with `C`). The goal `get_code(S,C)` is similar, but as a side-effect, the actual character becomes the next one, even if the code of the old actual character does not match `C`, and `get_code(S,C)` fails. The goal `at_end_of_stream(S)` is true, if stream `S` has already been read until its end (`S\ = user`). The goal `put_code(S,C)` writes the character with code `C` on stream `S`, while `nl(S)` sends a newline onto `S`. E.g.:

```
% PreCond: F1 and F2 are textfiles with appropriate access.
% appf(F1,F2) :- the content of F2 is inserted at the end of F1.
appf(F1,F2) :-
    open(F1,append,A), open(F2,read,R),
    af(A,R),
    close(A), close(R).


af(A,R) :-
    get_code(R,C),
    ( C == -1 -> true     % end of R
    ; put_code(A,C), af(A,R)
    ).
```

Consider now the term I/O. The goal `read(S,T)` reads a whole term from stream `S`. Then it tries to match it with `T`. Because any term can be continued with an infix operator and a connected term, the term to be read must be terminated with a dot, and at least one whitespace character (like the sentences of a program):

```
| ?- read(user,X), read(user,Y).
|: 12.  a+b*c.
X = 12,  Y = a+b*c
```

The goal `write(S,T)` writes term `T` on stream `S` without the terminating dot. (Term `T` can be any term, even a whole list.) But the atoms are never put between quotes or backquotes, because these usually represent messages to be printed:

```
| ?- write(user,'Value of Pi = '), read(user,Pi).
Value of Pi = 3.14159265358979323846.
Pi = 3.141592653589793
```

If we need an output readable for other Prolog programs, we use `writeq(S,T)`:

```
| ?- writeq(user,father('Isaac','Jacob')), write(user,'.\n').
father('Isaac','Jacob').
```

16.10.3 Dynamic predicates

Up till now we have considered only static Prolog predicates: a predicate is static by default, i.e. its code cannot change when it has been loaded. However, a predicate may be defined dynamic. For example, if we have the following declaration: `:-dynamic(p/2).` predicate `p/2` will be dynamic, i.e. its code will be variable even while the program runs. In this way, the running program can dynamically learn new rules and forget obsolete ones.

A decaration must precede the first rule of the predicate referred to. But a declared (and therefore existing) predicate does not necessarily have any rule. If there is a predicate with no rule, each goal invoking this predicate silently fails. But any goal invoking a non-existing predicate raises an exception called `existence_error` (16.12).

We can add new rules to a dynamic predicate. Goal `asserta(R)` adds a fresh copy of rule `R` to the predicate identified by the head of `R` as its new first rule, while `assertz(R)` adds this copy of `R` to that predicate as its new last rule.[17] Let us notice that this making a fresh copy of `R` is necessary because the further run of the program may substitute some logical variables of term `R`, and later backtracking may also delete some actual variable bindings in term `R`. Clearly such modifications of term `R` must not modify the rule added to the program.

Consider now a simple example of dynamic predicates: let us suppose that file `lpp.pl` contains predicate `connected/2`:

[17] A fresh copy is a copy of the term where the variables are substituted by fresh (i.e. new) variables. If a variable has more occurrences, the same fresh variable is used for each occurrence, but the different variables are substituted with different fresh ones. And the variable bindings to nonvar terms (i.e. variables substituted by nonvar terms) inside the original term are substituted with direct references to the appropriate terms in the fresh copy.

```
:- dynamic(connected/2).
connected(X,X).
```

and it is processed as follows. (We suppose that predicate `edge/2` defines an acyclic directed graph.)

```
| ?- consult(lpp).
% consulting c:/documents and settings/pl/book/lpp.pl...
% consulted c:/documents and settings/pl/book/lpp.pl
%          in module user, 0 msec 1536 bytes
yes
| ?- assertz((connected(X,Y):-edge(X,Z),connected(Z,Y))).
yes
| ?- listing(connected/2). % print the clauses of the predicate
connected(X, X).
connected(A, B) :-
       edge(A, C),
       connected(C, B).
```

A dynamic rule may be deleted by a goal like `retract(RulePattern)` where the *head* part of `RulePattern` must be a Prolog `atom` or `compound` identifying the dynamic predicate referred to by the `retract` statement. This goal deletes the first rule of the predicate referred to which matches term `RulePattern`. Forcing backtracking even each rule matching `RulePattern` can be deleted:

```
| ?- retract((connected(X,Y):-Body)).
Y = X,   Body = true ? ;
Body = edge(X,_A),connected(_A,Y) ? ;
no
```

Based on this example – if it were not a built-in predicate – we could define predicate `retractall/1` as follows.

```
retractall(P) :- retract((P:-_)), fail.
retractall(_P).
```

If we call any of the program manipulating predicates above, and it refers to a non-existing predicate, then this predicate is created as a dynamic one:

```
% my_consult(F) :- read the sentences of file F,
% omit the declarations and directives, and
% create dynamic predicates from the rules.
my_consult(F) :-
   open(F,read,S),
   consulting_loop(S),
   close(S).
consulting_loop(S) :-
   read(S,Sentence),
   ( Sentence == end_of_file -> true % end of S
   ; Sentence = :-(_) -> consulting_loop(S)
   ; assertz(Sentence), consulting_loop(S)
   ).
```

Dynamic predicates are typically used to memorize lemmas and negative lemmas derivable from the actual run of a program, in order to avoid recomputing this information ([SS94], [O'K90] and [SB04]).

For example, if predicate `edge/2` defines a graph with complex structure, and we are interested, which nodes can be reached from a given one, the following program may be useful, because it remembers the visited nodes, it avoids infinite looping, and if it fails to achieve goal from an internal node, after backtracking and arriving at it again it does not recompute the paths going out from that node. (Note: `ground/1` may not be a built-in in your Prolog, but it is easy to program it: see Exercise 16.7 in Section 16.7.)

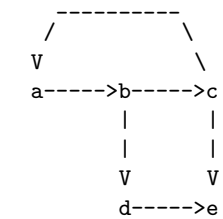
```
% Z can be reached form a given A.
reach_from(A,Z) :-
   ground(A), % A must be ground term
   retractall(visited(_)), asserta(visited(A)),
   reach_2(A,Z).

% do not call directly:
reach_2(A,A).
reach_2(A,Z) :-
   edge(A,B), \+ visited(B),
   asserta(visited(B)), reach_2(B,Z).
```

However, we cannot encourage the use of dynamic predicates to emulate global variables, especially if their usage can be avoided. This can lead to procedural programming style. And the overuse of dynamic predicates implies high runtime costs. For example, each modification of a dynamic predicate forces rebuilding the tables used for first argument indexing, each *assert* call makes a fresh copy of the whole rule to be inserted, independently from the size of the data structures stored in it, etc.

Especially in our previous problem, if we need even the paths going form node A to Z, or the structure of the graph is not so complex, or node A may not be given in the query, the next solution is suggested.

```
% Example graph:
edge(a,b). edge(b,c). edge(b,d).
edge(c,a). edge(c,e). edge(d,e).
```



```
% path(A,Z,Path) :-
%     Path is an acyclic list of nodes from A to Z.
path(A,Z,Path) :- path_2(A,[],Z,Path).
path_2(A,Ancestors,A,Path) :-
   reverse([A|Ancestors],Path).
path_2(A,Ancestors,Z,Path) :-
   edge(A,B), B\=A, \+member(B,Ancestors),
   path_2(B,[A|Ancestors],Z,Path).
```

16.11 Collecting solutions of queries

A Prolog query may have many solutions, but these are at leaves of different branches of the search tree of the query. Therefore, we can receive these solutions through backtracking, and the Prolog environment forgets one, when it backtracks to search for another. So we would never have them all together, but Prolog has built-in predicates ([DEDC96], [ISO95], [O'K90], [Car12], [SS94] and [SB04]) for collecting the solutions. The most important, and also the simplest one is `findall(Solution,Goal,Results)`, where `Goal` is any query (if it is a conjunction or disjunction, it must be bracketed), `Solution` is the term to be collected from the solutions of `Goal`, and `Results` is the list of the collected terms. Usually `Goal` and `Solution` shares variables. Typically `Solution` is one of the variables of `Goal` or it is a `compound` consisting of some variables of `Goal`. Procedurally speaking, the following algorithm computes `Results`.

```
Results := [];
while( still there is (another) solution of Goal )
   { solve Goal; % substituting some vars of Solution
     append a copy of Solution to the end of Results;
     backtrack Goal; % deleting the bindings of its vars
   }
```

For example, using the example programs in the previous subsection (16.10.3):

```
| ?- findall(X,edge(X,Y),Xs).
Xs = [a,b,b,c,c,d]
| ?- findall(e(X,Y),edge(X,Y),XYs).
XYs = [e(a,b),e(b,c),e(b,d),e(c,a),e(c,e),e(d,e)]
| ?- findall(Z,reach_from(a,Z),Zs).
Zs = [a,b,c,e,d]
| ?- findall(Z,reach_from(d,Z),Zs).
Zs = [d,e]
| ?- findall(Path,path(_,_,Path),Ps).
Ps = [ [_A],[a,b],[a,b,c],[a,b,c,e],[a,b,d],[a,b,d,e],
       [b,c],[b,c,a],[b,c,e],[b,d],[b,d,e],
       [c,a],[c,a,b],[c,a,b,d],[c,a,b,d,e],[c,e],[d,e] ]
```

Considering the first test of `findall(Solution, Goal, Results)`, we can see that the multiple solutions of `Goal` are put on the list `Results` in general. If we want to eliminate the duplications, we need something like predicate `sort(Xs, Ys)` which sorts list `Xs` into strictly increasing order according to (@ <)/2 (removing duplications) and try to match the result with `Ys`. (It is a built-in of most Prolog implementations. Anyway, it is not hard to write our own version: see `unionsort/2` in Section 16.6 in the solution of Exercise 16.6. `sort(Xs, Ys)` is very effective: its computational complexity is $\Theta(n * log(n))$ where n is the length of `Xs`.) Predicate `collect/3` is similar to `findall/3` except that it removes the multiplications of the solutions:

```
% collect(Solution,Goal,Results) :- Result is a
%     strictly increasing proper list of the Solutions of Goal.
collect(Solution,Goal,Results) :-
   findall(Solution,Goal,ListOfSols), sort(ListOfSols,Results).
```

Considering the directed graph defined by predicate `edge/2` we can compute the set of nodes having successor as follows (compare to the first test of `findall/3`):

```
| ?- collect(X,edge(X,_Y),Xs).
Xs = [a,b,c,d]
```

16.12 Exception handling in Prolog

Error and exception handling is standard part of modern programming languages. Therefore, it is also important part of the ISO Prolog standard.

In Prolog a query typically fails, succeeds (possibly leaving one or more choice points) or goes into infinite recursion. However, sometimes a predicate invocation cannot be interpreted. For example, this is the case with an invocation referring to a non-existing predicate (in the given scope), the query `arg(I,T,A)` if `I` is a negative `integer` or `T` is a *simple term* (see Section 16.4), the goal `X is Exp` where `Exp` is not an arithmetic expression, and goal `read(S,X)` if it finds a non-term in `S`. In such cases the invocation raises the appropriate *exception* like in other high-level programming languages. And the call finishes with this *exception* instead of success or fail.

A *Prolog exception* can be represented with an `atomic` or `compound` (i.e. a `nonvar`). It can be raised also with the call `throw`(*exception*). The unhandled exceptions become *runtime errors*. However, the Prolog development environment handles these errors (at the level of its interactive shell) printing the appropriate error message, and the Prolog prompt (| ?−).

If a Prolog goal raises an exception but does not handle it, we say that the goal *propagates* the exception. Therefore, we can say that *runtime errors* are exceptions propagated to the Prolog shell.

Exceptions can be handled with the built-in predicate `catch`(*Goal*,*ExceptionPattern*,*HandlerGoal*).

A *catch* invocation may handle just the exceptions raised (but not handled) during the evaluation of the *Goal* in its first argument. Therefore, the first actual parameter of a *catch* invocation is a *protected goal*.

A *catch* invocation first calls its first parameter.

If *Goal* propagates no exception, then it works as if it were called bare (without the *catch* protecting it).

However, if *Goal* propagates an exception E, first a fresh copy F of E is made. Next, each variable bindings performed through the evaluation of *Goal* are deleted, together with each choice point, each call frame pushed into the call stack, and each backtrackable events. (However, the side-effects of the extra-logical predicates are not deleted.)

Next F and *ExceptionPattern* are matched.

If this pattern matching is successful, then *HanderGoal* is called, and its evaluation is equal to the further evaluation of the *catch* invocation.

Otherwise the *catch* invocation propagates exception F.

The standard built-in predicates always raise exceptions of the form `error(IsoErr, ImpErr)` where term `IsoErr` is defined by the standard, while `ImpErr` is implementation dependent. For example:

```
| ?- catch( X is a, error(IsoErr,ImpErr), true ).
ImpErr = domain_error(_A is a,2,expression,a),
IsoErr = type_error(evaluable,a/0)
```

In the next example, predicate `read1term(Term)` can read any term (terminated by a dot and a whitespace character) from the standard input, and skips the remainder of the actual input line. Nonetheless if the standard input starts with a non-term (a term with syntax error), it prints the error-term defined by the ISO standard, and repeats reading.

```
read1term(Term) :-
    catch( ( read(user,Term), skip_line(user) ) ,
           error(Err,_), ( write1term(Err), read1term(Term) )
         ).

write1term(Term) :- writeq(user,Term), nl(user).

| ?- read1term(Term).
|: a*(b+ .
syntax_error('. cannot start an expression')
|: a*(b+c.
syntax_error(') or operator expected')
|: a*(b+c).
Term = a*(b+c)
```

16.13 Prolog modules

Although the "general core" of ISO Prolog ([DEDC96] and [ISO95]) is widely accepted and supported by Prolog implementers, and there is also an ISO standard for Prolog modules [ISO00], this latter is considered a failure, and as far as we know, there is no implementation of this part of the standard. Therefore, here we introduce only the module system of SICStus Prolog [Car12] which is flat and predicate-based: it is widely accepted, and it is a starting point of many new developments.

16.13.1 Flat, predicate-based module system

Predicate-based module system means that the Prolog terms are global, and visible in each module. But each module has its own predicate space, and the predicates are local to their module by default. However, they can be defined as

public predicates, and then they can be imported (from their module) into other modules, becoming part of these modules, and accessed directly in them.

Flat module system means that there is no hierarchy, and all the modules of a program are at the same level. One file may contain just one module, but one module may be spread into several files.

There is a default module called `user`. Each source file is loaded into it by default. The default type-in module of the queries at the Prolog prompt is also module `user`. Therefore, if one does not want to use the module system, one does not have to use it. However, programming in large makes it necessary.

The first sentence of a module file must be a module declaration in the following form.
: − `module`(*ModuleName*, *PublicPredicates*).
ModuleName must be a Prolog atom. *PublicPredicates* must be a proper list of predicate specifications in the form *name/arity*. The predicates encountered here are the *public* predicates of the module. The predicates of a module may be defined in that module. Or these may be imported into it, using, for example, one of the the built-in predicates
`use_module`(*ModuleFileName*, *ImportList*)
`use_module`(*ModuleFileName*)
called typically in a directive (16.2). *ImportList* must represent a subset of the public predicates of the module defined in file *ModuleFileName*. If we invoke `use_module(MFN)` in module `M`, each of the public predicates of the module defined in file `MFN` is imported into module `M`. These calls load the file *iff* it is necessary.

For example, let us suppose that we have the module file `graph.pl` with the module `graph_of_edges` in it. The only public predicate of this module is `edge/2` defining the edges of the graph. We also suppose that goal `edge(X,Y)` can check or return an edge, that is, there is no restriction on its use. For example:

```
:- module( graph_of_edges, [edge/2] ).
edge(a,b). edge(b,c). edge(b,d).
edge(c,a). edge(c,e). edge(d,e).
```

Let us write module `search` with the only public predicate `path/3` which is able to find simple paths (i.e. paths without loops) on a graph defined by predicate `edge/2`. (Predicate `path/3` has already been defined at the end of (16.10.3), and tested in (16.11).)

```
:- module( search, [ path/3 ] ).


:- use_module( graph, [edge/2] ).
:- use_module( library(lists), [reverse/2] ).
```

```
% path(A,Z,Path) :-
%     Path is an acyclic list of nodes from A to Z.
. . .
```

In order to define the file containing the graph dynamically, we have to change only the first two sentences of the module:

```
:- module( search, [ new_graph/1, path/3 ] ).
new_graph(File) :- use_module( File, [edge/2] ).
. . .
```

We suppose that the latter version of module `search` is defined in the module file called `search_path.pl`. Now we can initialize this module with any module file defining the graph through the public predicate `edge/2`:

```
| ?- use_module(search_path).
% ...module search imported into user...
yes
| ?- new_graph('../book/graph').
% ...module graph_of_edges imported into search...
yes
| ?- findall(Path,path(_,e,Path),Pathes).
Pathes = [ [e],[a,b,c,e],[a,b,d,e],
         [b,c,e],[b,d,e],[c,a,b,d,e],[c,e],[d,e] ]
```

16.13.2 Module prefixing

The strict module system outlined above implies some basic problems.

First, it prevents us from testing our program without altering it, because we cannot directly invoke the local predicates of a module. This problem is simply solved: any goal can be prefixed with a module name in the form *module:goal* overriding the source module of *goal* to *module* (`op(550, xfy, :)`). We can similarly override the source module of a predicate specification, rule head or sentence [Car12].

For example, predicate `search : path_2/4` in the previous subsection (16.13.1) can be invoked from any module using the prefix "`search :`":

```
| ?- search:path_2(d,[b,a],E,Path).
E = d,   Path = [a,b,d] ? ;
E = e,   Path = [a,b,d,e] ? ;
no
```

If there are more module prefixes of some query, rule head or sentence, just the rightmost one has effect. For example, goal `k : m : n : p(X)` is equivalent to goal `n : p(X)`.

If you consult a module file (load it in interpreted mode: see (16.10.1)), even the local predicates of the module loaded can be listed using module prefix. For example:

```
| ?- listing(search:path_2/4).
search:path_2(A, Ancestors, A, Path) :-
       lists:reverse([A|Ancestors], Path).
search:path_2(A, Ancestors, Z, Path) :-
       graph_of_edges:edge(A, B),
       B\=A,
       \+member(B,Ancestors),
       search:path_2(B, [A|Ancestors], Z, Path).
```

This means that this module system cannot really hide a predicate, but it can prevent conflicts of predicates with the same name and arity in different modules.

Note that module prefixing used without sensible control may destroy the module structure of our program.

16.13.3 Modules and meta-predicates

The problem shown through the next example is related to the interactions of *meta-predicates* and modules. Predicate `forall(P,Q)` (below) enumerates the solutions of goal `P` and for each solution it tries to find the first solution of goal `Q`: if `Q` fails, `forall(P,Q)` also fails immediately. It does not collect the solutions. It is used to check whether `P` implies `Q`, or it is invoked to force side-effect `Q` for each solution of `P`:

```
% forall(P,Q) :- for each solution of P, Q can be solved,
%                 that is P implies Q.
forall(P,Q) :- \+ ( P, \+Q ).

 ?- forall(path(b,_,Path),(write(Path),write(' '))).
[b] [b,c] [b,c,a] [b,c,e] [b,d] [b,d,e]
```

The `forall/2` form is a meta-predicate and its arguments are meta-arguments parameterized with meta-goals. If that `forall/2` is a public predicate of module `m` and we invoke `forall(path(b,_,Path),(write(Path),write(' ')))` in module `h`. Probably we want to use predicate `path/3` visible in module `h`. But the body of `forall/2` is invoked in module `m`. Therefore, `path(b,_,Path)` is invoked also in module `m`. But `path/3` may not be visible in module `m`, or even worse, `m:path/3` may differ form `h:path/3`. (It follows that in the first case goal `path(b,_,Path)` raises `existence_error`, in the second case it silently invokes another predicate.)

One possible solution is to prefix the meta-parameters of a meta-predicate invocation with the name of the calling module (notice that in general each meta-

parameter needs module prefix, but in the next example the second parameter refers only to built-in predicates and so this qualification may be omitted):

```
?- forall(h:path(b,_,Path),(write(Path),write(' '))).
```

Nevertheless, this manual qualification is inconvenient, and error-prone: one may forget the qualification, and it may lead even to silent errors; or one may rename the calling module, but forget to rename the module prefix with it. Therefore, it is better if this qualification of the meta-parameters with the calling module is automatic. This automatic qualification is called *module name expansion.*

Therefore, we can declare *module name expansion* for the meta-arguments of the public predicates of the modules. In such a *meta-predicate declaration* a meta-argument must be indicated with a *colon* (or an integer) [Car12]; a non-meta-argument can be indicated with any ground term except the colon and the integers. Then the meta-parameters of an invocation to the meta-predicate are automatically prefixed with the calling module, while the other actual parameters are not expanded:

```
:- module( m, [ forall/2, collect/3 ] ).

:- meta_predicate forall(:,:), collect(?,:,?).

forall(P,Q) :- \+ ( P, \+Q ).

collect(Solution,Goal,Results) :-
   findall(Solution,Goal,ListOfSols), sort(ListOfSols,Results).
```

The Prolog compiler expands the predicate invocation `forall(C,D)` into `forall(h:C,h:D)` provided that `h` is its calling module. Similarly, the predicate invocation `collect(X,G,Xs)` is expanded into `collect(X,h:G,Xs)` if `h` is its calling module.

There is the same case when we call predicates loading program files like `compile(F)` or `consult(F)`, e.g. from module `h`. They must also know that program code `F` must be imported into module `h`. Similarly, if module `m` is defined in file `lpp.pl`, and `use_module(lpp,[forall/2]` is invoked in module `user`, this invocation must also know that predicate `forall/2` must be imported into module `user`. And there are the predicates dynamically modifying the program like `asserta/1`, `assertz/1`, `retract/1`, etc. They must know, into which module to assert, or from which module to delete the rules.

Actually, the built-in predicates of SICStus Prolog are defined in the predefined module `prolog`, and they are its public predicates. They are automatically imported into each module. The predicates loading program files, those dynamically accessing (and possibly modifying) the loaded program, and those working with Prolog goals (like `findall(Sol,Goal,Solutions)`) are defined as meta-predicates specifying the arguments waiting for Prolog goals, rules, file names (or blackboard keys: see [Car12]) as meta-arguments, that is, arguments to which

module name expansion is due. Then these invocations are informed about the module they have to operate on.

16.14 Conclusion

Clearly, it is over the scope of this short introduction to discuss the whole field of logic progamming or even to discuss the full language of Prolog and/or the details of its programming methodology. Nevertheless, we hope we can give you a hand if you want to start a detailed study.

16.14.1 Some classical literature

There are easy-to-read introductions to the logical basis in [FGN90] and [Nil82]. There is a more general introduction to the topic in [Fla94] and [Kow79]. The theoretical aspects of logic programming are detailed in [Llo87].

Originally published in 1981, the first textbook on programming in Prolog was that of Clocksin and Mellish. It became popular because of its comprehensive, tutorial approach, and general programming examples. An updated, extended version is [CM03]. The book of Sterling and Saphiro is considered one of the best handbooks on the programming methodology of Prolog [SS94] which is suggested for students at beginner, and intermediate level. For advanced students we suggest [O'K90], and still some chapters of [SS94].

There is a good description of the Prolog standard in the book of Deransart, Ed-Dbali, and Cervoni ([DEDC96], [ISO95] and [ISO00]). The programs of this chapter were tested in SICStus Prolog 4.2.3. The User's Manual, and more documentation is available at [Car12]. Finally, the *Prolog 1000 database* includes more than 500 Prolog application entries, and it is available on the Internet.

16.14.2 Extensions of Prolog

Logic programming is not separated from other branches of programming. There have been several attempts to unify LP with functional programming, which is the other main declarative programming paradigm. Some of them led to well functioning systems, like implementations of *ALF*, *λProlog*, *Curry*, *Mercury*, etc. ([DL86], [Han94] and [Car13]). However, the practical use of the *functional logic programming* (FLP) languages is extremely rare, maybe because of the lack of a (de facto) standard, and the lack of the programmers properly trained.

There are interesting attempts to extend Prolog with object-oriented features ([Mos94] and [Car12]). These extensions are useful in building simulator programs, and expert systems.

Phil Vasey's *flex* is a frame based extension of *LPA*, and *Quintus* Prolog. Frames are similar to classes of OOP, but they are specialized for building expert

systems. There are important business applications, intelligent query, registering, diagnostics, and scheduling systems.

Prolog implementations typically know the *definite clause grammar* (DCG) formalism as a built-in extension. The DCG formalism is strong enough to define any language described with a Chomsky grammar, Turing automaton, or attribute grammar. Their application makes it much easier to develop parsers, compilers, text generators of formal and natural languages, and intelligent interfaces processing the texts of such languages, although these application areas well fit Prolog in general. When we load a DCG into Prolog, its rules are transformed one by one into Prolog rules using an algorithm of simple syntactic transformations. In this way, the DCG notation is just a de facto standard syntactic sugar in Prolog. Because of its comprehensiveness, it is very popular, and in the Prolog folklore it is used even in application areas seemingly far from text processing, for example in list handling utilities.

16.14.3 Problems with Prolog

Prolog, and library modules of *C++, Java, etc. implementations* containing parts of Prolog are often used in practice.

It is an important problem that only the standard of the core of Prolog is widely accepted ([DEDC96], [ISO95] and [ISO00]). But there is no *de facto* standard of modules, no standard of C/C++, Java, Oracle interfaces, no standard of GUI, etc.; although these are part of any serious Prolog implementation. Clearly, the developers of software packages do not want to depend on a particular Prolog implementation, because any time it may disappear from the market. Therefore, only half a dozen Prolog implementators are present in the business world and they have been there for more than twenty years now, although their number is much higher.

In many Prolog implementations, program tracing and debugging cannot go through the foreign language interface. Therefore, C++, Java, etc. programmers often switch from Prolog to a local package of their own language. (However, logic programming studies are needed to use effectively also these packages.) On the other hand, these local packages rarely have the effectivity and robustness of a professional Prolog implementation.

There is a myth that the Prolog programs are slow. Astonishingly, it is based partly on an unsuccessful USA project in the *sixties of the last century* which was trying to build an effective LP language [SS94]. The key of the successful European projects in the seventies was adopting *pattern matching instead of unification* in goal reduction, especially in parameter passing.[18] On the other hand, a good Prolog programmer needs special programming experiences, and studies. The reputation of the language does not benefit from the fact that it is misused by Prolog hackers.

[18] See the occurs check problem in (16.6.2,16.6.3).

16.14.4 Fifth generation computers and their programs

Maybe Japan's Fifth Generation Computer Systems project (FGCS) of the eighties was the most famous initiative applying some kind of "Prolog" as its main language. It aimed to develop highly parallel computers with many CPUs instead of increasing the complexity of a single CPU. Some prototype computers were created. They can be programmed with a *committed choice concurrent constraint logic programming* language: its predicates consist of guarded Horn clauses where each rule contains a guard which is similar to a Prolog cut. When a predicate has been invoked, the rules matching the invocation start to work in parallel, and when one of them arrives at the guard, it cuts the others. Therefore, there are no search trees like in LP in general, and there is no possibility to find the different solutions satisfying a query while traversing its search tree with backtracking or other strategy.

So it has turned out that the *committed choice* feature conflicts with the fundamental strength of *logic programming* compared to *functional programming*: the applications needing intelligent search cannot take advantage of search trees. And the prototype machines of the FGCS project were soon outperformed by general purpose computers. (The same had happened before to the Lisp Machine and to the Thinking Machine.) But in spite of these failures, for example "multi-core architectures at the low-end and massively parallel processing at the high end" seem to be winner ideas of the FGCS Project now.[19]

16.14.5 Newer trends

Mercury is a promising FLP language (16.14.2) developed at the University Of Melbourne under the supervision of Zoltán Somogyi. The core of the language is similar to pure Prolog; and a query is evaluated through backtracking traversal of its search tree. But it is purely declarative with strict, static type and mode system (in a predicate invocation one parameter can be only purely input or purely output), which allows the highest level of compile-time checks and code optimization while prevents the use of some effective and elegant programming techniques, that is the organisation of data with partial lists and trees ([SS94] and [O'K90]), and therefore the step-by-step, top-down approximation of the output (16.5.2). Probably these later features imply that it could not become the successor of Prolog, although its strong, polymorphic type system, and its module system allow the development of high quality, standard libraries.

The development of effective, parallel Prolog systems retaining backtracking search and in general, the support of conventional Prolog programs so that the goals and/or the branches of the program run in parallel without forcing the programmer to organize parallelism, this is a strong trend of the search in the field of LP. There are two basic approaches. If some branches of the search tree are evaluated in parallel, the system is *or-parallel* (for example, the *Aurora Prolog*

[19] See http://en.wikipedia.org/wiki/Fifth_generation_computer.

by Warren; Gigalips project). If some goals of a goal sequence are reduced in parallel, this Prolog is *and-parallel* (for example, the *Muse* version of SICStus Prolog). Warren's *Andorra-I* unifies both approaches. However, the best sequential Prolog implementations still often overperform these implementations.

Constraint Logic Programming (CLP) originates from the end of the eighties. Today this is one of the most successful areas of LP ([Hen86], [MS98], [SB04] and [Car12]). It is typically used to solve optimization problems. The program code is similar to that of standard Prolog but it is more declarative than that and the search is more intelligent: performing special computations it takes advantage of the problem solving methods of other scientific areas. Beside Prolog goals we have special goals called *constraints* behaving like demons. The variables of these constraints have predefined domains (for example, `X :: [1..5]` means that the domain of `X` is the integer range `1..5`). The run of the program means the backtracking traversal of the search tree, like in Prolog, but there is also a *constraint store*. When the constraints are encountered during the run of the program they are put into the constraint store. When any parameter of a constraint in this store is modified (for example, it is substituted by a value or its domain narrows), the consistency of the constraint store is checked, and if it is found inconsistent, the search backtracks. For example, the consistency check may find that a modification of a variable of a constraint implies the modifications of other variables, and if the domain of some variable contains just one element, it is substituted. Any modification implies further check. If some domain becomes empty, the search backtracks.

For example, SICStus Prolog contains three different constraint solvers implemented in libary modules [Car12]. There are also constraint programming packages built on OO languages, for example the *ILOG Solver* of IBM implemented as a C++ library.

We mention that other newer trends are Abductive, Answer Set, and Inductive Logic Programming.

16.15 Summary

Today – as a result of successes and failures – the place of LP seems to be in the "thinking" components of multiagent systems. Using LP we can effectively develop intelligent data base interfaces, packages needing intelligent search (even on the Internet), text processors, translator programs, natural language interfaces, expert systems, programs of logic puzzles, programs with symbolic computations, optimizations, etc.

The approaches of the problems are different with LP, and different with more traditional programming paradigms. Therefore, LP helps you to find new, elegant solutions.

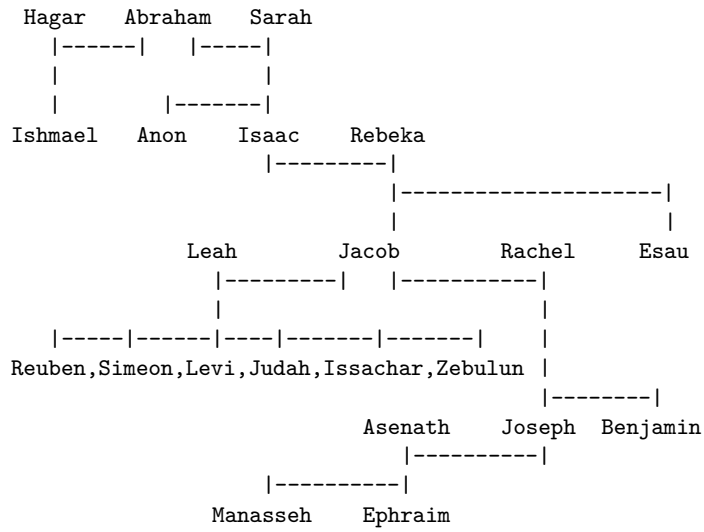


Figure 16.4: family/3 in Exercise 16.2

16.16 Exercises

Exercise 16.1. Apply the algorithm of goal reduction (16.6.1) to this query tested in (16.5.2): `append_(Xs, Ys, [1, 2, 3])`. Draw the corresponding search tree.

Exercise 16.2. Let us suppose that we have facts of the form "`family(Father, Mother, Children)`.". For example:

```
family('Abraham','Sarah',['Isaac','Anon']).
family('Abraham','Hagar',['Ishmael']).
family('Isaac','Rebeka',['Jacob','Esau']).
family('Jacob','Rachel',['Joseph','Benjamin']).
family('Jacob','Leah',['Reuben','Simeon','Levi','Judah',
                       'Issachar','Zebulun']).
family('Joseph','Asenath',['Manasseh','Ephraim']).
```

Define predicates `brother_or_sister/2`, `aunt_or_uncle/2`, and `parent_in_law/2` based on `family/3` with the usual meaning. (In order to help their test, Figure 16.4 illustrates predicate `family/3`.)

Exercise 16.3. Write list handling predicates to receive the prefixes, suffixes, continuous sublists, and possibly discontinuous subsequences of a proper list. Write predicate `divide(Xs, Odds, Evens)` which takes proper list `Xs`; puts the first, third, fifth, etc. items into `Odds`, and the second, fourth, sixth, etc. items into `Evens`, thus generating two subsequences of `Xs` of approximately equal length. Write predicate `sorted_union(Xs, Ys, Us)` producing the sorted union of

`Xs`, and `Ys` in `Us` where `Xs`, and `Ys` are strictly increasing proper lists according to the standard order; and `Us` will have the same properties. Each predicate must be able to check the appropriate property, and it must be able to generate the lists satisfying that property. Take advantage of the programming methods (step-by-step approximation of the output, accumulator pairs, etc.) while coding a predicate. Verify the finiteness of the search tree of each corresponding query.

Exercise 16.4. Consider the representation suggested for binary trees introduced at the very end of Section 16.4 , page 948.Using this representation (`o` is the empty tree, and `t(` *leftSubTree*, *root*, *rightSubTree*) is the scheme of a nonempty tree), define predicate `inorder(Tree,List)` where `List` contains the inorder traversal of `Tree`.

Exercise 16.5. Use the representation suggested above for binary search trees (sorted according to the standard order of the Prolog terms). Write the usual operations of such search trees (creating an empty tree, inserting a given piece of data, checking for it, deleting it: the tree must not contain duplications).

Be careful: do not mix the notion of binary search tree with the notion of the search tree of a Prolog query.

Exercise 16.6. Implement the well-known algorithms of sorting lists like Insertsort and Mergesort in Prolog. Be careful that the computational complexity of the Prolog programs should not be higher than that of the corresponding algorithm. (For example, the computational complexity of Insertsort is $O(n^2)$, and that of Mergesort is $\Theta(n * log(n))$ where n is the length of the proper list to be sorted.)

Note: The computational complexity (i.e. operational or time complexity) of a Prolog program is measured in LI (number of Logical Inferences) which means the number of predicate invocations performed during the run of the program.

Exercise 16.7. Try to implement your own version of `ground/1`: `ground(Term)` is true, *iff* `Term` is a ground term (see Section 16.4).

Exercise 16.8. Try to implement your own version of predicate `findall/3`.
It can be called `find_all/3`. Put it into a module.

Exercise 16.9. Find the description of the standard Prolog predicate `setof/3` ([DEDC96], [ISO95], [Car12], [SS94] and [SB04]).
Compare it to predicate `collect/3` in (16.11).

Exercise 16.10. Implement the standard operations of queues in a module.
`empty(Q)`: `Q` is empty queue (create and check);
`add(Q,X,QX)`: add `X` to the end of `Q`;
`rem(XQ,X,Q)`: remove the first item of queue `XQ`.

Exercise 16.11. In Subsection (16.13.1) we defined predicate `path/3`, which is able to traverse a graph in depth-first (backtracking) manner. Write another predicate which implements breadth-first-search on a graph computing a shortest path between a given start and goal node. Use the queue handling predicates defined in Exercise 16.10. (A shortest path is a path consisting of the least number of nodes here.)

Exercise 16.12. Implement the well-known algorithm of Quicksort on Prolog lists: Quicksort selects an arbitrary item X of a nonempty list, and separates the remainder of the list into two lists. The first list contains the items smaller than X. The second list contains the items greater than X. (The items equal to X can go into either of them.) Then the two lists are sorted recursively. The resulting list is generated from the two sorted lists with X as a middle element: these are concatenated together.

16.17 Useful tips

Tip 16.1. Consider now a search tree of query `append_(Xs,Ys,[1,2,3])` referring to predicate `append_/3`:

```
append_([],Ys,Ys).                                  % a1
append_([X|Xs],Ys,[X|Zs]) :- append_(Xs,Ys,Zs).     % a2
```

In the solution of Exercise 1 we use the following notations. The two rules of `append_/3` are called **a1**, and **a2**. If the depth of a goal in the search tree is **i**, then it is denoted with **gi**. If there is a goal **gi**, a goal reduction is applied to it, and rule **aj** is involved in it, then the goal reduction is denoted with **a(i,j)**. If rule **aj** is involved in a goal reduction **a(i,j)**, then we suppose that the LP system renames each variable **V** of rule **aj** to **Vi**. The $\mathbf{k}^{\text{th}}$ solution is called **sk**. Only the output substitutions (see Section 16.2.4) are shown.

Tip 16.2. You can choose a child using `member/2`.

Tip 16.3. Take care of the base cases and of the recursive cases of the predicates. For each predicate, give a sufficient condition of the finiteness of the search tree ("PreCond"). The finiteness of the search tree of each corresponding query can be based on two facts:

1. The input is proper list;
2. And its length is decreasing through the recursion.

Tip 16.4. A straightforward solution contains two rules: the inorder traversal of the empty tree is empty list, while that of a nonempty tree is that of the left subtree, the root, and that of the right subtree concatenated into a single list. (Try to solve it with single `append/3` call in the rule body.) A more refined version

avoids the `append/3` calls, and it has linear time complexity: we generalize the problem to append the inorder traversal of a tree before a list. (Consider the note of Exercise 16.6.)

Tip 16.5. The time complexity of the solutions should be proportional to the height of the search tree. (Consider the note of Exercise 16.6.) This effectivity can be achieved by distinguishing the cases of empty and nonempty trees: the recursive cases go down into the appropriate subtree.

Tip 16.6. Generalize Insertsort to `sorted_inserts(Xs,As,Ys)` inserting the elements of `Xs` into the sorted list `As`. Try to give tail recursive solutions. In Mergesort the base cases are the empty list, and the lists of a single item. Longer lists are divided into two lists of approximately equal lengths. (Use `divide/3` from Exercise 16.3 to divide a list.) The two lists are sorted with Mergesort, and then results are merged in a sorted way. The code of this sorted merge is similar to the code of `sorted_union/3` from Exercise 16.3, except that the duplicates are preserved.

Note: If we use `sorted_union/3` in Mergesort instead of sorted merge, we receive `unionsort/2` which is equivalent to `sort/2` in Section 16.11.

Tip 16.7. Recursively, a ground term is an atomic or a compound with ground arguments. **Note:** Although predicate `ground/1` is not part of the Prolog standard, many implementations contain it with the obvious meaning.

Tip 16.8. As a first try, one may assert the first parameter of `find_all/3` for each solution of the second argument (the parameterizing goal) into a dynamic predicate local to the module. You may use `assertz/1` for this purpose. Then (using `retract/1`) you can retract the asserted facts while collecting the parameters into a list applying step-by-step approximation of the output (16.5.2).

It is a useful initialization of `find_all/3` to delete each possible clauses of the dynamic predicate used for collecting the solutions. In this way one gets rid of the garbage left by any previous run propagating an exception.

Another problem is risen if one wants to embed a call to `find_all/3` into another invocation to it. Let us see a simple example of the appropriate behaviour (the example is a bit artificial, because `Xs` = `Zs`, but hopefully easy to understand):

```
| ?- find_all( Zs+Ys,
         ( append(Xs,Ys,[1,2,3]),
           find_all(Z,member(Z,Xs),Zs) ),
           As ).
As = [[]+[1,2,3],[1]+[2,3],[1,2]+[3],[1,2,3]+[]]
```

Unfortunately, if the method above were programmed in a straightforward way, the inner invocation of `find_all/3` could modify the partial results of the outer one, because they use the same dynamic predicate to collect their solutions. In order to solve this problem we have to identify levels of the embedded `find_all/3`

invocations, and each invocation works only at its level. In addition, if a goal parameterizing a `find_all/3` call propagates an exception, still we have to restore the outer level.

Tip 16.9. They have similar parameters to those of `findall/3`, and both of them collects (the) solutions of the parameterizing goal into a strictly increasing list omitting multiple solutions. But consider the role of variable quantification in `setof/3`. You may also try to use alternatively `collect/3`, and `setof/3` instead of `findall/3` in Exercise 16.11. Be careful, because in some cases `setof/3` may fail (but it is easy to overcome this difficulty). You may also try to measure the the effeciency of the different solutions. Unfortunately, the way of these measurements is implementation dependent in Prolog.

Tip 16.10. Using the trivial *proper list* representation, the maximal and average computational complexity of the operations `add(Q,X,QX)`, and `rem(XQ,X,Q)` are $O(n)$ where n is the length of the queue. (We suppose that the frequency of these operations is the same.) For example, one may use the trivial proper list representation where a nonempty queue has the form `[X1,X2,...,XN]` and `[]` represents the empty one. Then the computational complexity of `add` is linear, and that of `rem` is constant, therefore the average computational complexity of `add` and `rem` is also linear. (If a nonempty queue had the form `[XN,...,X2,X1]`, the computational complexity of `add` would be constant, but that of `rem` would be linear.)

If we use a clever *double-stack* representation, the average computational complexity of these operations become $\Theta(1)$.

In this representation `d([X1,...,XM],[YN,...,Y1])` represents the abstract queue $<$ `X1,...,XM,Y1,...,YN` $>$; `add` pushes a new item into the second stack (as the new first item of the list representing the second stack) with constant computational complexity; `rem` removes the topmost item of the first stack (that is the first item of the list representing it) also with constant computational complexity, provided that the first stack is nonempty. But if that first stack (i.e. list) is empty, we have to check the second one. If it is empty, the operation fails. If it is nonempty, we can *reverse* this second list and move it into the first one. Then we can remove its first item as before. The problem is that the computational complexity of the *reverse* operation is linear, and therefore sometimes the computational complexity of the `rem` operation is also linear. Besides this, the average computational complexity of these operations is still $\Theta(1)$, because considering

```
rem(  d( [], [Y|Ys] ),  FirstItem,  ResultQueue  )
```

`[Y|Ys]` is as long as the number of `add` calls before the actual *reverse* call inside `rem` (but not before any previous *reverse* call inside `rem`). Therefore, the cost of the actual *reverse* call can be scattered among those `add` calls when we calculate the average computational complexity of `add` and `rem`, and so it is $\Theta(1)$.

If we use a *pair of partial lists* where the second component is the logical variable at the end of the first partial list, we are able to succeed in reducing the maximal computational complexity of these operations to $\Theta(1)$. Using this representation an empty queue has this form: `Z-Z`, and a nonempty queue has this form: `[P1,P2,...,PN|Z]-Z` (`var(Z)` must be true in both cases). If one checks the solution, one finds that the implementation is somewhat tricky and considering a queue which is the input of an `add` operation, its second component is better to be a `var`. But after the `add` call this property of the input queue does not stand. Therefore, it does not clearly fit any `add` operations any more. A `rem` operation does not change the second component of the queue. Therefore, if a `rem` is performed on queue Q resulting in queue Qr, and then an `add` is performed on queue Q resulting in queue Qa, then even the second component of Qr is a `nonvar`, and it should not be used for subsequent queue operations. However, for practical purposes it is usually enough that in a linear sequence of queue operations the output queue of any operation can be used as the input queue of the next operation.

Tip 16.11. A breadth-first-search (BFS) of a graph begins with a chosen start node.[20] Then it explores the children of the start node, then the nodes available through at least two arcs (edges), then those available through at least three arcs, and so on. BFS may have goal node(s). If yes, it stops when a goal node is achieved. Otherwise it goes on until each node accessible from the start node is explored. (Therefore, it terminates *iff* the number of nodes accessible from the start node is finite or there is a goal node accessible from the start node.) In BFS the parent of a node is defined as the node from which it was found first. During the search for each node visited BFS usually records a reference to its parent, in order to record a shortest path from the start node to the actual node through these backward references (and the parent reference of the start node is somewhat extremal). The key operation of BFS is *expanding* a node which means collecting its children. Also we need the set of visited nodes, and a queue containing the nodes already visited but still not expanded. The algorithm initializes the queue with the start node. In each step it removes the first element of the queue, expands it, and puts its unvisited children at the end of the queue. If it is relevant, the goal property of a node is checked immediately before it is put into the queue. Instead of booking the parent references of the visited nodes, in Prolog the queue of BFS may contain nonempty lists: the first element of such a list is the node to be expanded, the second element is its parent, the third one is its grandparent, and so on, the last element of the list is always the start node. Therefore, such a list with a given first node always contains a shortest path to this node.

Tip 16.12. The solution is straightforward. However, one may get rid of the costs of appending the parts of the result at each level of recursion, and this is not

[20] We suppose that the graph is locally finite, i.e. there is no node with infinitely many children.

trivial. One may use the so called *d-lists*, i.e. pairs of (partial or proper) lists where the second list is a suffix of the first one.

16.18 Solutions

Solution 16.1. The search tree of `append_(Xs,Ys,[1,2,3])`

```
%% append_(Xs,Ys,[1,2,3]): apply the algorithm of
%% goal reduction to this query.
%% Draw the corresponding search tree.
%% (explanation of notations at Section "Clues")

?- append_(Xs,Ys,[1,2,3]).      % g0
     { Xs <- [], Ys <- [1,2,3] }      % a(0,1)
Xs = [], Ys = [1,2,3]                              % s1
     { Xs <- [1|Xs0] }                % a(0,2)
     append_(Xs0,Ys,[2,3]).     % g1
       { Xs0 <- [], Ys <- [2,3] }     % a(1,1)
Xs = [1|Xs0] = [1|[]] = [1], Ys = [2,3]            % s2
       { Xs0 <- [2|Xs1] }             % a(1,2)
       append_(Xs1,Ys,[3]).     % g2
         { Xs1 <- [], Ys <- [3] }     % a(2,1)
Xs = [1|Xs0] = [1,2|Xs1] = [1,2], Ys = [3]         % s3
         { Xs1 <- [3|Xs2] }           % a(2,2)
         append_(Xs2,Ys,[]).    % g3
           { Xs2 <- [], Ys <- [] }    % a(3,1)
Xs = [1,2|Xs1] = [1,2,3|Xs2] = [1,2,3], Ys = []    % s4
           { match([],[X3|Zs3]) fails }    % a(3,2)
```

Solution 16.2. Family relationships

```
family('Abraham','Sarah',['Isaac','Anon']).
family('Abraham','Hagar',['Ishmael']).
family('Isaac','Rebeka',['Jacob','Esau']).
family('Jacob','Rachel',['Joseph','Benjamin']).
family('Jacob','Leah',['Reuben','Simeon','Levi','Judah',
                       'Issachar','Zebulun']).
family('Joseph','Asenath',['Manasseh','Ephraim']).



brother_or_sister(X,Y) :-
    family(_,_,Xs),
    member(X,Xs), member(Y,Xs), X \== Y.

aunt_or_uncle(X,Y) :-
    brother_or_sister(X,Z),
    ( family(Z,_,Cs) ; family(_,Z,Cs) ),
    member(Y,Cs).

parent_in_law(X,Y) :-
    ( family(X,_,Cs) ; family(_,X,Cs) ),
    member(C,Cs),
    ( family(C,Y,_) ; family(Y,C,_) ).
```

Solution 16.3. Basic operations on lists

```
%% PreCond: Xs or Ys is proper list.
%% prefix(Xs,Ys) :- Ys is a prefix of Xs.
prefix(_Xs,[]).
```

```
prefix([X|Xs],[X|Ys]) :- prefix(Xs,Ys).

%% PreCond: Xs is proper list.
%% suffix(Xs,Ys) :- Ys is a suffix of Xs.
suffix(Xs,Xs).
suffix([_X|Xs],Ys) :- suffix(Xs,Ys).

%% PreCond: Xs is proper list.
%% sublist_x(Xs,Ys) :- Ys is a continuous sublist of Xs.

%% a: Suffix of a prefix
sublist_a(Xs,Ys) :- prefix(Xs,Ps), suffix(Ps,Ys).

%% b: Prefix of a suffix
sublist_b(Xs,Ys) :- suffix(Xs,Ss), prefix(Ss,Ys).

%% c: Recursive definition of a sublist
sublist_c(Xs,Ys) :- prefix(Xs,Ys).
sublist_c([_X|Xs],Ys) :- sublist_c(Xs,Ys).

%% PreCond: Xs is proper list.
%% subseq(Xs,Ys) :-
%%     Ys is a possibly discontinuous subsequence of Xs.
subseq([],[]).
subseq([X|Xs],[X|Ys]) :- subseq(Xs,Ys).
subseq([_X|Xs],Ys) :- subseq(Xs,Ys).

%% PreCond: Xs is proper list.
%% divide(Xs,Odds,Evens) :- in their original order, the
%%     first, third, fifth, etc. items of Xs are in Odds, the
%%     second, fourth, sixth, etc. items of Xs are in Evens.
divide([],[],[]).
divide([X|Xs],[X|Ys],Zs) :- divide(Xs,Zs,Ys).

%% PreCond: Xs and Ys are proper lists, sorted strictly
%%          increasingly according to the standard order.
%% sorted_union(Xs,Ys,Us) :-
%%     Us contains the sorted union of Xs and Ys.
sorted_union([],Ys,Ys).
sorted_union([X|Xs],[],Us) :- !, Us = [X|Xs].
sorted_union([X|Xs],[Y|Ys],Us) :-
    ( X @< Y -> Us = [X|Zs], sorted_union(Xs,[Y|Ys],Zs)
    ; X @> Y -> Us = [Y|Zs], sorted_union([X|Xs],Ys,Zs)
    ; Us = [X|Zs], sorted_union(Xs,Ys,Zs)
    ).
```

Solution 16.4. Inorder traversals of a binary tree

```
%% PreCond: Tree is a binary tree represented as follows.
%%     o - empty tree
%%     t( LeftSubTree, Root, RightSubTree ) - nonempty tree
%% inorder(Tree,Is) :- the inorder traversal of the data
%%                     in Tree results proper list Is.
inorder(o,[]).
inorder(t(Lt,X,Rt),Xs) :-
    inorder(Lt,Ls), inorder(Rt,Rs), append(Ls,[X|Rs],Xs).

%% The optimized version of predicate inorder/2.
%% Without appends, linear time complexity.
inorder_opt(T,Is) :- inorder_app(T,[],Is).

%% inorder_app(Tree,List,Is) :- the inorder traversal
%%           of Tree appended before List results Is.
inorder_app(o,Xs,Xs).
inorder_app(t(Lt,X,Rt),Xs,Is) :-
```

```
    inorder_app(Rt,Xs,Ys), inorder_app(Lt,[X|Ys],Is).
```

Solution 16.5. Basic operations on binary search trees

```
%% The usual operations of proper binary search trees
%% (creating an empty tree, inserting a given piece of data,
%% checking for it, deleting it:
%% the search trees must not contain duplications).

%% empty_tree(T) :- T is an empty tree.
empty_tree(o).

%% PreCond: T is proper binary search tree.
%% tree_ins(T,X,TX) :- TX proper binary search tree is
%%     received by the sorted insert of X into T.
tree_ins(o,X,t(o,X,o)).
tree_ins(t(Lt,Root,Rt),X,TX) :-
    ( X @< Root -> TX = t(LXt,Root,Rt), tree_ins(Lt,X,LXt)
    ; X @> Root -> TX = t(Lt,Root,RXt), tree_ins(Rt,X,RXt)
    ; TX = t(Lt,Root,Rt)
    ).

%% PreCond: T is proper binary search tree.
%% tree_has(T,X) :- T contains item X.
tree_has(t(Lt,Root,Rt),X) :-
    ( X @< Root -> tree_has(Lt,X)
    ; X @> Root -> tree_has(Rt,X)
    ; true
    ).
%% Queries like tree_has(o,X) automatically fail.

%% PreCond: TX is proper binary search tree.
%% tree_del(TX,X,T) :- T proper binary search tree is
%%     received by the sorted delete of X from TX.
tree_del(t(Lt,Root,Rt),X,T) :-
    ( X @< Root -> T = t(Ldt,Root,Rt), tree_del(Lt,X,Ldt)
    ; X @> Root -> T = t(Lt,Root,Rdt), tree_del(Rt,X,Rdt)
    ; Rt == o -> T = Lt
    ; Lt == o -> T = Rt
    ; T = t(Lt,Min,Rmt), Rt = t(L,Y,R),
      out_min(L,Y,R,Min,Rmt)
    ).

%% PreCond: t(L,X,R) is a proper binary search tree.
%% out_min(L,X,R,Min,Tm) :-
%%     the leftmost element of the proper binary tree
%%     t(L,X,R) is Min, the remaining tree is Tm.
out_min(o,Min,Rt,Min,Rt).
out_min(t(L,X,R),Y,Rt,Min,t(Lm,Y,Rt)) :-
    out_min(L,X,R,Min,Lm).
```

Solution 16.6. Sorting lists

```
%% PreCond: Xs is a proper list.
%% insertsort(Xs,Ys) :- proper list Ys contains
%%     the items of Xs increasingly sorted according to the
%%     standard order of Prolog terms.
insertsort(Xs,Ys) :- sorted_inserts(Xs,[],Ys).

%% PreCond: Xs is a proper list.
%% sorted_inserts(Xs,As,Ys) :- sorted insert of
%%     the items of proper list Xs into the sorted
%%     proper list As results sorted proper list Ys.
sorted_inserts([],Ys,Ys).
sorted_inserts([X|Xs],As,Ys) :-
```

```
    insert_sorted(As,X,Bs), sorted_inserts(Xs,Bs,Ys).

%% PreCond: As is an increasingly sorted proper list.
%% insert_sorted(As,X,Bs) :- sorted insert of X
%%               into As results the increasingly
%%               sorted proper list Ys.
insert_sorted([],X,[X]).
insert_sorted([Y|Ys],X,Zs) :-
    ( X @=< Y -> Zs = [X,Y|Ys]
    ; Zs = [Y|Us], insert_sorted(Ys,X,Us)
    ).

%% PreCond: Xs is a proper list.
%% mergesort(Xs,Ys) :- proper list Ys contains
%%     the items of Xs increasingly sorted
%%     according to the standard order of Prolog terms.
mergesort([],[]).
mergesort([X|Xs],Ys) :- mergesort(Xs,X,Ys).

%% PreCond: Xs is a proper list.
%% mergesort(Xs,X,Ys) :- proper list Ys contains
%%     the items of [X|Xs] increasingly sorted
%%     according to the standard order of Prolog terms.
mergesort([],X,[X]).
mergesort([Y|Ys],X,Zs) :-
    divide(Ys,As,Bs),
    mergesort(As,X,Es), mergesort(Bs,Y,Fs),
    sorted_merge(Es,Fs,Zs).

%% PreCond: Xs,Ys: increasingly sorted proper lists.
%% sorted_merge(Xs,Ys,Zs) :- proper list Zs is
%%     the result of the sorted merge of Xs and Ys.
sorted_merge([],Ys,Ys).
sorted_merge([X|Xs],[],Ms) :- !, Ms = [X|Xs].
sorted_merge([X|Xs],[Y|Ys],Ms) :-
    ( X @< Y -> Ms = [X|Zs], sorted_merge(Xs,[Y|Ys],Zs)
    ; X @> Y -> Ms = [Y|Zs], sorted_merge([X|Xs],Ys,Zs)
    ; Ms = [X,Y|Zs], sorted_merge(Xs,Ys,Zs)
    ).

%% Note: if we use sorted_union/3 instead of sorted_merge/3,
%% we receive a strictly increasing list, duplicates removed:

unionsort([],[]).
unionsort([X|Xs],Ys) :- unionsort(Xs,X,Ys).

unionsort([],X,[X]).
unionsort([Y|Ys],X,Zs) :-
    divide(Ys,As,Bs),       % See Exercise 3.
    unionsort(As,X,Es), unionsort(Bs,Y,Fs),
    sorted_union(Es,Fs,Zs). % See Exercise 3.
```

Solution 16.7. Checking, whether a term is ground or not

```
%% ground_(Term) :- Term is a ground.
ground_(Term) :-
    ( atomic(Term) -> true
    ; compound(Term),
      functor(Term,_F,N), ground_args(N,Term)
    ).

ground_args(N,Term) :- N>1,
    arg(N,Term,A), ground_(A),
    N1 is N-1, ground_args(N1,Term).
ground_args(1,Term) :-
```

```
    arg(1,Term,A), ground_(A).
```

Solution 16.8. Implementing the predefined predicate `findall/3`

```
%% find_all(X,Goal,Xs) :- findall(X,Goal,Xs).

:- module( findall, [ find_all/3 ] ).
:- meta_predicate find_all(?,:,?).

:- dynamic((solution/2, counter/1)).

set_counter(C) :-
    retractall(counter(_)),
    asserta(counter(C)).

:- set_counter(1).

find_all(X,Goal,Xs) :-
    counter(I),
    %% to handle embedded calls to find_all/3:
    I1 is I+1, set_counter(I1),
    catch(
          (
            %% clear things, if an old goal crashed:
            retractall(solution(I,_)),
            %% assert solutions at the Ith level:
            Goal, assertz(solution(I,X)), fail
          ;
            collect_solutions(I,Xs),
            set_counter(I)
          ),
          Error,
          ( set_counter(I),
            throw(Error)
          )
         ).

collect_solutions(I,Ys) :-
    ( retract(solution(I,X)) ->
          Ys = [X|Xs], collect_solutions(I,Xs)
    ; Ys = []
    ).
```

Solution 16.9. Collecting solutions without duplications

```
.
%% Compare the standard Prolog predicate setof/3
%% to our predicate collect/3.
```

Predicate `setof/3` is less effective than `collect/3`, but it is more complex and it has more expressive power: it may produce more sets of solutions through backtracking depending on the variable quantifications of the goal parameterizing `setof/3`. However, this extra expressive power is useless in many practical applications. Also, `collect/3` never fails, and produces exactly one set of solutions, but `setof/3` fails iff the parameterizing goal has no (more) solutions.

Solution 16.10. Queue handling and Prolog modules

```
%% Implement the standard operations of queues in a module.
%% empty(Q): Q is empty queue (create and check).
%% add(Q,X,QX): add X to the end of Q.
%% rem(XQ,X,Q): remove the first item of queue XQ.
```

```
:- module( queue3,
   [ empty0/1, add0/3, rem0/3,
     empty/1, add/3, rem/3,
     emptyq/1, addq/3, remq/3 ] ).

%% proper list representation:
%% A queue has the form: [X1,X2,...,XN]

empty0([]).

%% add0/3 has linear computational complexity:

add0([],X,[X]).
add0([Y|Ys],X,[Y|Zs]) :- add0(Ys,X,Zs).

rem0([X|Xs],X,Xs).

%% double-stack representation:
%% d([X1,...,XM],[YN,...,Y1]) represents
%% the abstract queue <X1,...,XM,Y1,...,YN>

empty(d([],[])).

add(d(Xs,Ys),E,d(Xs,[E|Ys])).

rem(d(Xs,Ys),E,ResultQueue) :-
    ( Xs = [Z|Zs] -> E = Z, ResultQueue = d(Zs,Ys)
    ; % Xs == [], Ys \== [],
      reverse(Ys,[E|Us]), ResultQueue = d(Us,[])
    ).

:- use_module( library(lists), [reverse/2] ).

%% Queue represented with a pair of partial lists
%% where the second component is the logical variable
%% at the end of the first partial list:
%% a nonempty queue has this form: [P1,P2,...,PN|Z]-Z
%% and an empty queue has this form: Z-Z
%% where var(Z) must be true.

emptyq(Q-Q) :- var(Q).

addq(Q1-[ITEM|Y],ITEM,Q1-Y).

%% A call: addq([P1,P2,...,PN|Z]-Z,ITEM,R).
%% After the call: Q1 = [P1,P2,...,PN|Z],
%%                 Z = [ITEM|Y], R = Q1-Y.
%% Therefore: R = [P1,P2,...,PN,ITEM|Y]-Y.

remq([H|T]-Z,H,T-Z) :-
     [H|T]\==Z.  % the input queue was nonempty

%% Using this last representation
%% each single operation needs a constant number of LI
%% (Logical Inferences = predicate invocations).
```

Solution 16.11. Shortest path

```
%% Graph-search with breadth-first-search strategy.

%% breadth_first_search(Start,Goal,SolPath) :-
%%     SolPath is a proper list of nodes representing a path
%%     of the minimal length (i.e.optimal) from Start to Goal.
```

```
breadth_first_search(Start,Goal,SolPath) :-
    ground(Start), ground(Goal),
    ( Start == Goal -> SolPath = [Start]
    ; empty(InitQueue), add(InitQueue,[Start],Queue),
      bfs(Queue,[Start],Goal,SolPath)
    ).

:- use_module( solution10_lp, [ empty/1, add/3, rem/3 ] ).

%% Queue is a queue of lists of the form [Node|Ancestors]
%% where Node\=Goal, and to Node we have found an optimal
%% path but Node has not been expanded. Ancestors consists
%% of the ancestors of Node on this optimal path starting
%% with its parent. Visited contains the visited nodes.

bfs(Queue,Visited,Goal,SolPath) :-
    rem(Queue,[Node|Ancestors],RemainderQueue),
    children(Node,Children,Visited),
    ( has(Children,Goal) ->
         reverse([Goal,Node|Ancestors],SolPath)
    ; process_children(Children,[Node|Ancestors],
                       RemainderQueue,ResultQueue),
      append(Children,Visited,NewVisited),
      bfs(ResultQueue,NewVisited,Goal,SolPath)
    ).

:- use_module( library(lists), [ reverse/2 ] ).

%% expansion: Children is the list of
%%     the unvisited children of Node.

children(Node,Children,Visited) :-
    findall(Child,child(Node,Child,Visited),Children).

%% Child is an unvisited child of Node.

child(Node,Child,Visited) :-
    arc(Node,Child), \+ has(Visited,Child).

has([X|Xs],Y) :-
    ( X == Y -> true
    ; has(Xs,Y)
    ).

%% Add the children with their ancestors to RemainderQueue.

process_children( [FirstChild|Children], Ancestors,
                  RemainderQueue, ResultQueue ) :-
    add(RemainderQueue,[FirstChild|Ancestors],TempQueue),
    process_children(Children,Ancestors,TempQueue,
                                        ResultQueue).
process_children([],_Ancestors,ResultQueue,ResultQueue).


/*  Test data   */

%% Acyclic component
    arc(a,b).   arc(a,c).   arc(a,d).  arc(a,e).
    arc(b,f).   arc(b,i).
    arc(c,f).   arc(c,g).
    arc(d,j).
    arc(e,k).
    arc(f,h).   arc(f,i).
    arc(g,h).
    arc(j,g).
```

```
    arc(k,j).

%% Cyclic component
    arc(x,y).   arc(y,z).   arc(z,x).
    arc(y,u).   arc(z,t).

/*
     a------>b-------->i
    /|\       \      /
   | | \       \    /                          ----------
   | |  \       \  /                          /          \
   | V   V       V/                          V            \
   | d   c------>f---->h                      x----->y----->z
   |  \   \          /                               |      |
   |   \   \        /                                |      |
   |    \   \      /                                 V      V
   |     \   \    /                                  u      t
   |      \   \  /
   V       V   V/
   e-->k-->j-->g
*/

%% | ?- breadth_first_search(a,g,Sol).
%% Sol = [a,c,g]
%% | ?- breadth_first_search(a,i,Sol).
%% Sol = [a,b,i]
%% | ?- breadth_first_search(x,t,Sol).
%% Sol = [x,y,z,t]
```

Solution 16.12. Quicksort and d-lists

```
%% PreCond: Xs is a proper list.
%% quicksort(Xs,Ys) :- proper list Ys contains
%%     the items of Xs increasingly sorted according to the
%%     standard order of Prolog terms.
%% Note:
%% Our quicksort selects the first item X of a nonempty list,
%% and separates the remainder of the list into two lists.
%% The first list contains the items smaller than X.
%% The second list contains the items greater than X.
%% (The items equal to X can go into either of them.)
%% Then the two lists are sorted recursively.
%% The resulting list is generated from the two sorted lists
%% with X as a middle element: these are concatenated together.
quicksort0([],[]).
quicksort0([X|Xs],Ys) :-
    separate(Xs,X,Smallers,Greaters),
    quicksort0(Smallers,Littles),
    quicksort0(Greaters,Bigs),
    append(Littles,[X|Bigs],Ys).
separate([],_X,[],[]).
separate([Y|Ys],X,Ss,Gs) :-
    ( Y @< X -> Ss = [Y|Ls], separate(Ys,X,Ls,Gs)
    ; Gs = [Y|Bs], separate(Ys,X,Ss,Bs)
    ).
%% One may use d-lists to get rid of the costs of append/3:
quicksort(Xs,Ys) :- quicksort(Xs,Ys,[]).
%% quicksort(Xs,Ys,Zs) :-
%%     d-list Ys-Zs represents the sorted Xs
%%     where list Zs is a suffix of list Ys.
%% Note: d-list [Y1,...,Yn|Zs]-Zs represents
%%       sequence <Y1,...,Yn>, and
%%       d-list Zs-Zs represents sequence <>.
quicksort([],Zs,Zs).
quicksort([X|Xs],Ys,Zs) :-
```

```
separate(Xs,X,Smallers,Greaters),
quicksort(Smallers,Ys,[X|Bigs]),
quicksort(Greaters,Bigs,Zs).
```

Final remark: Remember to TEST, what happens, if you force BACKTRACKING into your Prolog program?

17 Aspect-oriented programming

The main goal of this chapter is to present aspect-oriented programming as a new programming paradigm. After presenting the main reasons which lead to the creation and spreading of this new approach, its novelty will be described. Afterward some programming languages will be discussed which support this method, amongst these the AspectJ language will be introduced in more details. Finally, we look at some software development approaches related to aspect-oriented programming.

In the 70's the so called *software crisis* fundamentally changed the approach and methods of computer programming, propagating the structured, modular and object-oriented programming. Similarly, nowadays it is more and more noticeable that the developers of the programs are not able to adapt in speed and quality on the one hand, to Moore's law,[1] on the other hand, to the increasing rate of spreading of IT equipment in the public.

Object-oriented programming in that time was perhaps the best possible answer to the demands for clear and in that way, safer programming and for code reusability. These reason its widespread and de facto monopoly. This all was backed up by the standardization process introduced in modeling by UML.

As a general method and a programming language oriented approach, the OOP (and the UML) could not solve two problems. On the one hand, as it only helps general software modeling, in its original form it is only marginally suitable to solve the abstraction of application areas (such as business logic, 3D graphical modeling, controlling of telecommunications protocols, bank account management) and of software development aspects (such as logging, managing security services, network communication) or to support their standardization.

On the other hand, the UML based round-trip engineering solutions are only partially suitable for integrating legacy systems with new programs and for designing and generating complex software systems written in different languages and on various platforms.

For these problems numerous new methodologies try to find a solution. Such a methodology is, for example, the aspect-oriented programming [Kicz97], the multi-dimensional separation of concerns, the intentional programming ([IP-faq03] and [Sim99]), or the generative programming [CE00]. Amongst all of these methodologies the most mature and widely used is the aspect-oriented programming.

[1] Gordon Moore, the co-founder of Intel discovered in the 70's the regularity that the computational capacity of IT equipment is doubled in every 18 months. His thesis is confirmed, and this law is expected a few more decades to be valid.

Aspect-oriented programming is a programming paradigm that was born in the middle of the nineties. Nowadays, it is in the center of programming language research, and for sure, in the near future it will spread widely. About related methodologies, languages and features (even those discussed in this chapter) to the aspect-oriented programming, the research following [AOSD12] provides a detailed reference list.

The aspects are such functionally coherent program parts which can be found scattered[2] throughout the various parts of the source code in a program without the possibilities of AOP. In this way, the aspects implement a new kind of modularization feature which can make the separation of the various concerns [Dij76] of the software more perfect.

There are numerous aspect-oriented methods and constructs nowadays. The best known amongst these is the AspectJ for Java which will be discussed later in detail.

17.1 Overview of AOP

Software design methods and the programming languages mutually support each other. The modeling and designing methodologies brake down systems gradually into smaller and smaller units. The programming languages apply solutions in the opposite direction: they allow the developers to create abstractions representing subunits of the system, and by combining these in various ways to assemble the ready software product.

A design method and a programming language can efficiently work together, if the language offers such abstraction and composition services which support in an easily usable and natural way those elemental units to which the design method decomposes the systems.

In this regard, most of the object-oriented the procedural and the functional programming languages (described these as traditional programming languages for the sake of simplicity) function in a similar way. Those system design methods which were developed together with the traditional programming languages decompose the program based on behavior or functionality. This approach is usually called as functional decomposition. Of course, the kind of decompositions can greatly differ according to the languages and paradigms, but the language elements (function, procedure, module, object) are always the *functional* units of the big systems.

The last statement may seem too obvious, but becomes significant, since the aspect-oriented programming discussed in this chapter does this “orthogonally” to the above mentioned functional decomposition.

[2] The technical literature uses the notions *scattered code* and *tangling code*. The first means the scattering of the functionally coherent code parts and inadequate modularization, the last is a consequence of this: “tangling” refers to the increased complexity of the program product.

17.1.1 Aspects and components

Every program has parts which exist independently of the currently analyzed functionality in every subprogram, object or module resulting from the functional decomposition. Such elements are often program codes for synchronization, error handling, network communication or logging.

These details appear on many locations in nearly or exactly the same way, so they increase the size of the code unnecessarily, but during the design based on the traditional functional decomposition they are usually not grouped together into a separate program unit, because they do not represent separate elements in the functional logical structure of the system. If something changes in the above mentioned functionality, this leads to program modification demands on many locations at the same time which decreases the maintainability of the program.

For the proper handling of this situation aspect-oriented programming was introduced which could be defined the following way:

- *Components* are those program units which are the natural results of the functional decomposition of the system. (Object, method, procedure etc.)
- *Aspects* are those program parts which are not a natural result of the functional decomposition, but influence the behavior and efficiency of the system in some other systematic manner.
- After this, the goal of the aspect-oriented programming can be defined the following way: the components and aspects within the system should be separated with such methods and constructs which enable proper abstraction and the efficient composition of the system.

According to the above definition the AOP languages differ from the traditional programming languages mostly in two things. On the one hand, they support the decomposition of the programs by the abstraction of the components and aspects (so for example everything about logging could be described as a separate program unit), on the other hand, they also enable the natural and easy merging of the components and the aspect.[3] The latter is called by the technical literature as weaving which could be done by a precompiler or a compiler (like in AspectJ), or even in runtime (like in JMangler or PROSE).

17.1.2 Aspect description languages

In order for the aspect-oriented solutions to be able to live together with the existing languages in a natural way, such languages are usually defined which are

[3] Also the traditional programming languages contain isolated aspect-like elements. Amongst these the most obvious analogy could be the aspect-oriented interpretation of the exception handling. For example, in Java the behavior of the various exceptions can be described in separate modular units (*Exception* classes), the functionality and the exception handlers can be combined by explicit statements (**try**, **catch**). Of course the AOP is capable of more complex things than this (since the programmer here has to explicitly give instructions for the decomposition yet).

built into traditional languages extending them with the capability to define aspects and connect these aspects with the traditional parts of the program. These language extensions are called aspect description languages. Aspect description languages can be either specific, or for general use. Amongst the former can be found, for example, synchronization languages, languages aiding distributed programming or enabling monitoring (see e.g. [Lop97] and [GK99]), but this might include also the deployment descriptors from the nowadays so popular Enterprise JavaBeans technology which enables to specify the transaction aspects of the EJB business methods [Sun03]. General purpose aspect description languages are not specialized for a given kind of aspect description, but enable to specify and compose aspects implementing arbitrary functionality. Such a language is, for example, the AspectJ.

Considering the above, the requirements for the aspect-oriented implementation of an application are the following:

- – There must be a traditional language suitable for component description.
  - – There must also be one or more aspect description language(s).
- To combine these languages, appropriate weaver programs must exist.
- – The components must be written in the traditional language, focusing on their functional role.
  - – The aspects must be written using the appropriate aspect description language(s).

Aspect description languages differ from traditional languages usually in a visible way, since according to their nature they often contain declarative and imperative parts equally. The role of the imperative parts is obvious: these contain the operations which will form the implementation of each aspects. The declarative parts are usually made of rules which control the weaver program how and with which part of which components written in the traditional language have the behavior of the aspects to be connected with. This will be discussed in the next section in more detail.

17.1.3 Aspect weavers

The task of the aspect weavers is to assemble the whole system using the program parts written in the traditional and in the aspect description languages. The operation of the weavers is controlled by so called *join points*. The join points specify – based on the semantics of the traditional languages – where the aspects can join into the functioning of the components.

Join points can be, for example, the data streams between components or the entry points of the methods. These two examples also show that join points are not always explicitly defined by syntactical language elements, but also by

knowing the semantics (visibility and scope rules of variables, polymorphism and dynamic linking, etc.) of the language.

Due to the above the aspect weavers function in such a way that they first explore the (possible) join points of the component program, then they match these with the behavior of the aspects.

17.2 Introduction to AspectJ

The AspectJ [Kicz01] is a natural extension to the Java language: every Java program is also an AspectJ program, the programs produced by AspectJ can be executed by every JVM, the Java development tools can be naturally extended to AspectJ development tools, and the syntax of AspectJ is extending Java in a way that the Java programmers can consider it as a natural part of the language.

AspectJ was originally developed by XEROX PARC (Palo Alto Research Center) led by Gregor Kiczales. Today the development of AspectJ is coordinated by the Eclipse project [Ecl03].

To give a complete description of them is not our goal here, we would only like to demonstrate the approach, the particular aspect-oriented perception of the AspectJ.

17.2.1 Elements and main features of AspectJ

The main elements of the language: *dynamic join points*, *pointcuts*, *advices*, and *introductions* which could be combined into *aspects*.

- The *dynamic join points* refer to such events of the program execution, like a method call, receiving a method call, execution of the method, getting the attribute values, throwing exceptions, initialization of the classes and objects etc.
- The *pointcuts* specify sets of dynamic join points. From these pointcuts newer ones may be formed by the use of various set operations.
- The *advices* are such method-like operations which describe the operations assigned to a pointcut (executed before and/or after the event).
- The *introductions* enable to introduce required members (auxiliary variables and methods) for the implementation of the aspects.
- The *aspects* are modular units consisting of the three elements above.

In order to be able to clearly decide in what order the advices (even from multiple aspects) matched to a join point should be executed during run time, between and within the aspects there are well defined precedence relations. Beside this an inheritance relation can also be defined between the aspects.

17.2.2 A short AspectJ example

The next example shows how logging could be implemented using the constructions of the AspectJ.

The *SimpleErrorLogging* aspect below specifies that throwing an *Error* exception in any public method of all the classes of the package *hu.elte.inf* should cause its logging.

```
aspect SimpleErrorLogging {
    Log log = new Log();

    pointcut publicCalls():
        receptions(public * hu.elte.inf.*.*(..));

    after() throwing (Error e): publicCalls() {
        log.log(e);
    }
}
```

The declaration and initialization of the *log* variable is an introduction which is a traditional member variable declaration in its form. The new variable is not a member of an object or of a class, but of the *SimpleErrorLogging* aspect. AspectJ also supports introductions, for instance, or class level members for one or multiple classes.

The *publicCalls* pointcut specifies a set of method calls: this is signaled by the *receptions* keyword. After this keyword follow the descriptions of the desired methods where joker characters can also be used. The specified methods in this pointcut are public, and from the package *hu.elte.inf*. The class from this package is not specified: this is indicated by the second `*` joker character. The third `*` joker character allows any method names, the first and the `(..)` symbols mean any return values and any formal parameter lists.

The last element within the aspect is the advice which implements the behavior (logging), and specifies when this behavior should be executed for the method calls defined by the above mentioned *publicCalls* pointcut (after throwing an *Error* exception). The aspect weaver will match the logging to those dynamic join points where the given exception is thrown in a given method.

17.2.3 Development tools and related languages

Nowadays there are AspectJ extensions for many development tools (such as Eclipse, JBuilder, Forte 4J, SunOne Studio, Emacs) and support for other tools will keep coming. There is also a "native" development tool for AspectJ called AspectJ Browser. This allows to browse the language elements of the program in the usual way. For Java subprograms the joining aspects can be shown, and vice versa.

There are AspectJ-like extensions also for other languages. Such languages are, for example, the C, C++, C#, Perl, Python, Ruby or Smalltalk. These extensions have not reached the level of development, support and penetration, as AspectJ does.

17.3 Paradigms related to AOP and their implementations

Following are some related paradigms to AOP. There are some amongst them which could be seen as the predecessor of AOP (adaptive programming), or as a special case of it (multi-dimensional separation of concerns, generative programming), and there are others born with similar reasons, but based on different principles.

17.3.1 Multi-dimensional separation of concerns (MDSC)

This methodology can be seen as a generalization of the aspect-oriented programming, it was developed by IBM led by Harold Ossher and Peri Tarr. The MDSC [MDSC02] is such a paradigm which tries to break the tyranny of the dominant decomposition. This means that during the design of the system the subsystem should, even must be constructed not only in a single way, but in multiple ways according to multiple reference systems (dimensions). The former mentioned functional decomposition breaks down the system into components only in a single way (in one dimension of the references). Besides this to get clear, easy modifiable code with little complexity, also decomposition according to other dimensions is necessary. For example, by applying the object-oriented methodology the decomposition results in classes having operations describing various activities. Besides it would be needed that the same system should be decomposed by other methods "orthogonally" to the first. This will be shown in the following example [OT99].

Assume that we would like to develop a program, handling arithmetic expressions which can check the syntactical correctness of expressions and evaluate them, displaying the result. The model resulting from the object-oriented approach will contain classes in various relations (inheritance, aggregation, association) with each other, for example Expression, Literal, Variable, Operator, Unary Operator, Binary Operator etc. These classes will have the specialized versions of the operations Check, Evaluate and Print for the given class. A possible alternative decomposition would break down the system towards exactly these three tasks. Other aspects of the system (synchronization, logging, etc.) can be additional dimensions of the decomposition.

According to the principle of the multi-dimensional separation of concerns, to aid software development the decompositions based on different dimensions should be deemed to be equal, and during the lifecycle of the software the components created by them (the so called concerns) must be independently

manageable and separable. This goal can be reached with proper programming language and tool support.

A possible implementation of the above mentioned principles is the usage of hyperspaces. This is the basis of the HyperJ language which is an MDSC based extension of the Java language.

Subject-oriented programming (SOP)

The methodology of the *subject-oriented programming* [SOP03] was also developed by IBM (W. Harrison, H. Ossher). This method was the direct predecessor of the MDSC. The central notion of the methodology is the subject which is a collection of classes and class parts. Code parts needed for the solution of a subtask can be defined in form of a subject. The subject can be assembled into an application according to various composition rules. After all a subject could be seen as an aspect.

In a system built according to the object-oriented principles the objects model the objects from the real world. As a result, an object can participate in the solution of multiple problems (or program functions). A certain part of the data stored within the objects and certain operations are needed to solve one task, and other data and operations for another task. This can also be formulated that the same object can be examined according to multiple functions and angles that is it can play a role in defining multiple subjects. The subject is then a partial object model which contains the relevant parts (data members, operations) and connections (inheritance, aggregation, association) of the classes needed to implement a given functionality. (A similar notion is the "mixin" which represents a class part implementing a well defined, usually smaller functionality. The mixins are usually composed with other mixins and classes to produce objects with multilayered functionality.)

Subject-oriented programming is language independent, yet it was formed to an object-oriented approach. There is an implementation for the C++ language, and for the IBM VisualAge for C++ development tool.

17.3.2 Adaptive programming (AP)

This paradigm is a predecessor of AOP, and can be seen as a special case of it. It was developed on the Northeastern University of Boston led by Karl Lieberherr [AP80]. The principle of the *adaptive programming* is that the code of an activity should be made independently working form the structure, on which it would be applied. This is how the functionality and the structure (as the two important aspects of the system) can be separated.

This approach corresponds to *Demeter ' s law* which is a general object-oriented design principle (see 10.8.4), stating that the operations should use only minimal information about the structure of the system. For example, in an object-oriented program the class hierarchy is not burnt in into the code of the

methods, but is written so that it *adapts* to the structure. The methodology of the adaptive programming suggests that the methods doing the actual computations should be implemented in the traditional way (for example, in the given object-oriented language), but the code searching for objects for the above methods to execute on, should be defined using traversal strategies. The traversal strategies enable the searching for nodes with certain properties of the graph describing the connections between objects (inheritance, aggregation, association).

The principles of the adaptive programming are implemented, for example, by the Demeter/C++, or in Java by DemeterJ or its successor, by DJ.

17.3.3 Composition filters (CF)

Another methodology trying to overcome the deficiency of the object-oriented paradigm is the use of *composition filters*. They intervene with the delivery of messages between objects, extending the system so with a newer functionality, newer aspects of the operations. With these filters certain aspects of the behavior of the program can be defined in a modular way. The filters modify the incoming and outgoing messages of the objects that is, using aspect-oriented terminology, their join points are the messages sent and received. Beside defining aspects the composition filters can be used to implement other techniques, such as delegation and dynamic linking.

The composition filters were developed by Mehmet Aksit from the University of Twente [CF01]. There are already multiple implementations, such as the Sina language, or the extensions for the C++ language and for CORBA. Other interesting tries are for the Java language ConcernJ and ComposeJ.

Please note that the Enterprise JavaBeans technology [Sun03] is in close relation with the composition filters [CE00]. EJB-s can be used to define server side components which can rely on the services of the J2EE framework through standard interfaces. The code of the bean consists of the implementation of the business logic in the Java language, and a so called assembly descriptor which describes various other aspects (persistence, transaction handling, concurrency, security, shared communication) of the component. At the installation of the component, according to this assembly descriptor, the framework will generate wrapping objects (substantially composition filters) which will implement the above mentioned aspects of the component.

17.3.4 Generative programming (GP)

Generative programming is a quite new approach, its standard book was published with the same title in 2000 [CE00]. The essence of the paradigm is that it tries to automate software development on a big scale. That is why generative programming is a collection of methods and tools which aid in the design, building, configuration and assembly of the components of the system.

aspect-oriented programming is only one of these methods. Generic programming, metaprogramming, and intentional programming etc. can also be named here. The following few sentences summarize what generative programming can offer for software developers in the field of system analysis and design, and programming (implementation).

Generic programming is an implementation technique, on which the generative programming builds significantly. This technology supports the abstract composition of algorithms and data structures (enabling so the most abstraction from uninteresting details at building system components), besides, it ensures the configuration possibilities and interaction management of the so created abstract components. A beautiful example for generic programming is C++ Standard Template Library which uses besides parametrization with types and polymorphism (see Chapter 11.) also other techniques – iterators, adapters, function objects, traits.

Metaprogramming, which is another important feature of generative programming, means that programs are manipulated (configured, adapted), instrumented (for testing or profiling) or generated pragmatically. The last includes the usage of macros and also the aspect-oriented techniques. The compilers and precompilers are such metaprograms. Reflective programs which examine (introspection) or modify (intercession) themselves, are also metaprograms. (Smalltalk and the CLOS languages, for example, widely support reflective programming, the Java language only offers introspection.) Finally, another interesting example of metaprogramming is the C++ template metaprogramming based on the C++ *template* which enables to execute certain parts of the C++ program in compilation time [Ale01].

The methodology of the generative programming suggests that the knowledge of the application domains should be described in flexible parametrizable and combinable standard libraries ("active libraries"), and the system developers (designers) should have the possibility to use the best modeling languages for the given field (in contrast to the too general UML), and to use code generators heavily utilizing the services of these active libraries. In this way the state could be best approached, when the program code is generated (this is where the name of the paradigm comes) from the system plan in 100%.

17.3.5 Intentional programming (IP)

Intentional programming was started as a Microsoft Research project in the beginning of the 90'-es led by Charles Simonyi. The goal of this approach is on the one hand to vanish the sharp boundaries between the program plan and the code, on the other hand to be able to describe the abstractions of any kinds of fields and combine these abstractions flexibly and efficiently. That is why intentional programming is also seen as a specialization of the generative programming.

The name ”intentional programming” comes from cognitive psychology, and is intended to formulate that thanks to the ideal symbiosis of the abstraction and the representation, the original intention of the program creator will vanish less, as in the case of the traditional methods.

After Microsoft had stopped the IP project, not much later Charles Simonyi and Gregor Kiczales founded Intentional Software Corporation which tries to implement the idea of the IP in form of a complete development tool.[4] The participation of Kiczales in the project signals that this new approach aims primarily for the users of AOP, as IP is capable of describing aspect-orientation.

17.3.6 Further promising initiatives

The following are some initiatives which were created for the goals similar to the above.

Eclipse Universal Modeler

The goal of the IBM Research is to extend the Eclipse universal development tool to support arbitrary abstraction mechanisms and notation systems [Ecl03].

This tool uses standard (OMG that is Object Management Group and W3C that is WorldWide Web Consortium) protocols and basic infrastructure (XMI that is XML Metadata Interchange, MOF that is Meta-Object Factory, SVG that is Scalable Vector Graphics) to describe the metadata and to represent the diagrams.

Jackpot

This project also aims, like the EUM, for the development of an universal modeling tool, based on the NetBeans tool. The leader of this project is James Gosling at Sun Microsystems, one of the creators of Java, who also participated in the development of Emacs.

The development of UML

UML is also developing in the direction that it could generate a more complete program code from the resulting model of the design. Such oriented initiative is the UML behavior semantics, the plan for a completely UML based software platform. To quote Ivar Jacobson (the author of Objectory, one of the three basic object-oriented modeling notion systems): ”in a few years everyone will program in UML, without the present programming languages. This closely resembles the goals of the IP.”

[4] A significant part of the development of the IP was done in Hungary in cooperation with NETvisor Ltd. and ELTE Faculty of Informatics.

Also the result of Ivar Jacobson is the WayPointer system which uses intelligent agents[5] in a new way to help system designers and developers on the various fields of UML modeling [GT99], using Rational Rose (a UML modeler, designer and code generator tool) and RUP (Rational Unified Process, a methodology base on the UML, covering the full lifecycle of the software development). Using this help the designers could produce such UML models which enable to utilize the code generation capabilities to the fullest, optimal extent.

17.4 Summary

In this chapter we tried to present aspect-oriented programming as a new programming paradigm. We described the novelty of this new approach. After presenting this some programming languages were discussed which support this method, amongst these the AspectJ language was introduced in more details. Finally, some software development approaches related to aspect-oriented programming were described.

[5] The agents – by a possible definition – are such independent program components which react to the state and behavior of their environment, and are able to behave intelligently in a set manner by their controlling rules, intentions and emotions. In this concrete situation such agents are meant which are controlled by the knowledge base and rule system of the valid and complete usage of the possibilities of the UML, and make actual suggestions to the system designers, trying to ”figure out” the next or the missing steps.

18 Appendix

18.1 Short descriptions of programming languages

In this section we will try to give a short description of today's most widely used programming languages. We do not attempt to be comprehensive, our goal is only to show the diversity of the programming languages and to emphasize the approach not to implement everything on a single, currently trendy and widespread language, as the possibility is there to choose the ideal solution for any given problem and environment.

18.1.1 Ada

The programming language Ada was designed by an international team, the CII-Honeywell Bull on the order of the DoD (Department of Defense, USA). The leader of the group was the Frenchman Jean Ichbiah, who also designed the LIS programming language. The name of the language came from Lady Ada Augusta Byron, who is said to be the world's first programmer. Pascal was chosen as the starting language for the design, but some aspects from other languages – such as Modula-2, ALGOL 68, SIMULA 67, Alphard and CLU – were also taken.

The criteria of the design were the following:

- Reliability;
- Expandability;
- Consistency;
- Strongly typed.

The ANSI acknowledged the Ada standard in 1983, and it became an ISO standard in 1987. The revision of this standard resulted in the Ada 95 [Nyek98].

The Ada programming language – especially in designing larger software systems – proved to be very successful. Although it was meant for developing military systems, it was soon spread also to other fields of usage (commerce, telecommunication). The broader field of usage resulted in new requests and requirements for this programming language.

In 1988 the revision of the standard was started, following these recommendations of the Ada Board:

- Support of object-oriented programming;
- Possible interface to other programming languages;
- Hierarchical library structure;
- More advanced task scheduling, easier and reliable usage of shared objects;
- Extended parameter possibilities for templates.

By forming the new standard, backward compatibility with Ada 83 was very important (which is succeeded with some rare exceptions), and to preserve its great achievements (safety, reliability, etc.). It was important for this extension to be simple, so six new keywords were introduced in all.

So in 1995, Ada 95 was born, the worthy successor of Ada 83, which of course also became an ISO standard. Similar extensions and revisions were made and introduced in 2007 and in 2012. The actual standard Ada 2012 enhanced safe, secure and reliable software engineering to the next level with following features and benefits:

- Contract-based programming, where pre- and postconditions define the expectations and obligations of a subprogram, type invariants specify boundary constraints for objects of an encapsulated (private) type and subtype predicates capture general constraints on data objects;
- Concurrency and multicore support, where task affinities and dispatching domains allow tasks to be mapped to specific CPUs or cores and Ravenscar for multiprocessor systems adapts a safe and widely used tasking profile to modern architectures;
- Increased Expressiveness, where expression functions offer a convenient way to express simple functions, conditional expressions provide a compact notation for a common idiom and quantified expressions for universal and existential forms specify predicates over arrays and containers;
- Container Enhancements, where bounded containers use stack allocation and do not incur the overhead of dynamic memory management, task-safe queues and priority queues provide efficient implementations of synchronized structures, holder containers create singleton structures for objects of an unconstrained type, and iterators provide familiar idioms with uniform syntax to search and manipulate arrays and containers.

The central homepage of the language:
`http://www.ada2012.org/`

18.1.2 ALGOL 60

ALGOL (ALGOrithmic Language) is one of the languages which were designed to support scientific computations.

ALGOL-60 introduced the feature of block structures, the structured control transfer statements and the language implementation of recursion. This was the first language which offered the possibility for the users to introduce their own data types, and where the keywords were also reserved words.

This was the first language of which syntax was formally defined with the notation since known as BNF by John Backus and Peter Naur.

The central homepage of the language:
`http://www.engin.umd.umich.edu/CIS/course.des/cis400/algol/algol.html`

18.1.3 ALGOL 68

ALGOL 68 is an improvement of ALGOL 60. The designers of ALGOL 68 applied a new kind of grammar: the language was defined by a context-independent, two-level (so called Wijngaarden) grammar which was also suitable to describe not only the syntax, but also part of the semantics. So for example the rules of declarations and automatic type conversions were also defined by the help of the grammar. A new terminology was introduced for the notations, maybe this was also a reason that this language could not spread widely, today it is almost extinct. Meanwhile numerous features of it influenced the designers of later languages (for example Bjarne Stroustrup, the designer of C++).

The central homepage of the language:
`http://www.algol68.org/`

18.1.4 BASIC

BASIC (Beginner's All Purpose Symbolic Instruction Code) was developed in 1964 in Dartmouth led by John G. Kemény and Tom Kurtz, with the goal "to make the computer for every student accessible".[1] The biggest achievement of the language was to allow computer usage for everyone – not only for mathematicians and engineers – leading for the wide acceptance of computers in everyday life.

The central homepage of the language:
`http://www.fys.ruu.nl/~bergmann/basic.html`

18.1.5 BETA

The language BETA was designed by Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen and Kristen Nygaard, based on SIMULA 67.

This modern object-oriented language introduces an important abstraction: the "pattern". In BETA everything is a "pattern": the class, the procedure, the function, the parallel tasks and also the exceptions. There is pattern inheritance, so it can be applied in a broader range than simple inheritance in other languages (see [MMPN93]).

The central homepage of the language:
`http://www.daimi.au.dk/~beta/`

18.1.6 C

The language C was developed by Dennis Ritchie at the AT&TBell Labs in the beginning of the 70's [KR89]. The main reason for creating this language was that the UNIX operating system could be written in such a programming language, which is high level enough to greatly support the portability of the programs written in it, but also low level enough for developing operating systems

[1] Quote from John G. Kemény, presented by George Marx, see [Mar93]

and applications, which could fully utilize all the possibilities of the hardware resources.

According to the above criteria the C language supports all features, which were typical to the most modern high level – that is structured, procedural and modular – programming languages that time. At the same time low level resource access is also offered for the C developers. To be concrete, the flexible management of memory locations (pointers), the quite loose type system and the nearly totally omitted runtime checks assure the key to efficiency.

These also harbor the most dangers of the application of this language, since using C without proper understanding and omitting rigorous testing can easily result in unreliable and buggy program code.

Opposed to the common languages with "talkative" syntax in the 60's and 70's, C has an unusually dense syntax (for example { and } instead of the **begin** and **end** keywords). This feature – despite rendering the C source code to be hard readable – was quickly adopted by the programmers, and even caused numerous later programming languages to introduce C-like language elements.

C had a major role in spreading UNIX, and afterwards many other operating systems, system programs and applications were written in this language.

The central homepage of the language:
`http://cm.bell-labs.com/cm/cs/cbook/`

18.1.7 C++

The programming language C++ was developed by Bjarne Stroustrup at the AT&TBell Labs in the beginning of the 80's [Str00], as a further development of the language C.

The main reason for the creation of this language was the more and more widespread object-oriented paradigm in that time. According to this C++ extended the C language with some important features:

- Data abstraction;
- Object-oriented programming with multiple inheritance;
- Operator overloading;
- Templates;
- Exception handling;
- Data streams;
- Standard Library.

Besides all of these C++ kept the simplicity and efficiency of C by being backward compatible with it, for the design of the new constructs the same main aspects were used – the goal is to apply the highest level of constructs by fully utilizing the hardware resources. According to this every syntactical correct C program is also a syntactical correct C++ program.

Because the C++ language introduced the object-oriented concept not for its own sake, but by extending C, a language known and recognized by many,

this greatly contributed to the fact that the object-oriented approach became so dominant in the 90's.

The significance of C++ is also indicated by being the ancestor of the most widely used programming language in the beginning of the 21st century, Java (and its wraith, C#), which was developed in a very similar way.

The central homepage of the language:
`http://www.research.att.com/~bs/C++.html`

18.1.8 C#

C# is an all purpose imperative programming language developed by Microsoft [Sch02].

The language tries to combine the expressive power of C++ with the easy usage of Visual Basic. The syntactical bases come from C++, the simplicity from Visual Basic. C# is compared by many with Java, not completely without reason. The language supports object-oriented program development, and includes most of the capabilities of the C++ programming language, introducing some novel features. One of the major shortcomings of C# is that it does not support templates.

The central homepage of the language:
`http://msdn.microsoft.com/vcsharp/`

18.1.9 Clean

The Concurrent Clean (Plasmeijer, Nijmegen, 1987, [PE01] and [Pla99]) was developed from an experimental graph rewriting language (the LEAN) as a clean functional language using basically a lazy evaluation. Its current version (2.4) is very close to Haskell [HHZ02], but includes more language features.

The central homepage of the language:
`http://www.cs.kun.nl/~clean/`

18.1.10 CLU

The CLU language was designed and developed in the 1970's on the MIT led by Barbara Liskov, primarily for supporting the teaching of programming methodology. This was the first implemented programming language which offered language constructs for data abstraction by supporting the implementation of compilation units which make only the type specification accessible and hide other implementation relevant information. These compilation units are called *clusters*, that is where the name of the language comes from (see [LG96]). CLU introduced significant innovations in the areas such as exception handling, iterators and templates, and though the language is not widespread, designers of numerous programming languages took ideas from it.

The central homepage of the language:
`http://www.pmg.lcs.mit.edu/CLU.html`

18.1.11 COBOL

In 1959 the opinion was formulated on the Pennsylvanian University that a machine independent, data processing-oriented programming language was needed that is also suitable for handling registers. A language was requested in which only the task should be formulated (!), the computation should be done by the machine.

A joint taskforce referred to as the "Short Range Committee" was established, whose goal was to evaluate all the existing languages and create the "minimal program" of the new language. The name COBOL (COmmon Business-Oriented Language) was agreed on, and in 1968 the specification titled as USA Standard COBOL was published, then in 1974 a slightly modified version appeared which is still the most widespread COBOL version. The standard COBOL-85 introduced object-oriented constructs into the language. Most of the already existing COBOL programs are made according to the COBOL-74 standard, the majority of the compilers also support this version (see [Bak74]).

The developers of the language reached their goals. Concentrating on the task instead of the algorithm resulted in a particular, but for given tasks very applicable programming language. In many aspects, such as for types the portability of COBOL programs is much better than in most of the other languages – even for those developed a decade later –; the hardware independence of its language constructs is clearly shown by not requiring the programmer – as opposed to FORTRAN [LV77] – to know the machine code the compiler will produce from the program.

The central homepage of the language:
`http://www.cobolreport.com/`

18.1.12 Delphi

By the appereance of graphical applications the demand was raised to have such development environments on the software market which support the quick implementation of GUIs. The Borland company introduced the first version of Delphi in 1995 for this kind of support [Lis00]. Benefits include the form-centered IDE, quick compiler, simple database management, wide support of the Windows environment, high level access of operating system capabilities, and last but not least component based application development.

The programming language of Delphi is Object Pascal, basically an enhanced version of the Turbo Pascal 7.0 language. The creators of Delphi focused on the following points during language design:

- Simple and quick development of Windows applicaitons;

- Support of object-oriented properties;
- The possibility of client/server database management.

Later the Linux based version of Delphi, Kylix was also introduced.

The central homepage of the language:
`http://www.borland.com/delphi/`

18.1.13 Eiffel

The programming language Eiffel was designed in the 1980's led by Bertrand Meyer [Mey91]. Eiffel is an object-oriented language recommended in production environments for quality software design and implementation. The aspects of the creation of the language were: support for reusable, expandable, correct, safe, portable and effective programming. All of these were attempted to be solved in a simple, elegant and easy to handle way.

The ascendants of the Eiffel language are: SIMULA 67, ALGOL W, Alphard, CLU, Ada and the Z specification language. Besides a number of new aspects were considered in the field of inheritance, type and exception handling, assertions, and the constructs taken from the above languages were unified.

The Eiffel language was designed to be practically inseparable from its environment provided around it. Its usage is unthinkable without the comprehensive system of library classes. It is tightly integrated with the LACE language (Language for Assembling Classes in Eiffel) which describes the restrictions for the compilation and execution in a makefile-like way. The Eiffel development environment offers more tools to make the effective usage of the language easy.

The central homepage of the language:
`http://www.eiffel.com/`

18.1.14 FORTRAN

FORTRAN is the first language used for numerical computation and still the most widespread nowadays. Its name is originated from that: FORTRAN = FORmula TRANslator. The design of the first version started in 1954 by John Backus at IBM, the compiler was finished in 1957. In 1964 the first standard of the FORTRAN language was introduced by the ASA (American Standards Association) followed by many versions. The definition of FORTRAN 90 is worth being mentioned which, among others, introduced the possibility of recursion. For the standardization of the extensions supporting parallelism the HPF (High Performance Fortran) language was born. The modern FORTRAN 2003/2008 standards extend the language with exception handling and object-oriented possibilities and improves the cooperation with the C language (see [Fort03]).

As FORTRAN was developed for scientific purposes, its text processing capabilities are strongly limited. Thanks to its newer versions it is still a living language: some physical, statistical, sociological, weather forecast etc. computations are still done in FORTRAN today.

The central homepage of the language:
`http://www.fortran.com/`

18.1.15 Haskell

The creation of Haskell was decided in 1987 on a conference on functional programming (FPCA '87). The language was named after the mathematician Haskell Brooks Curry. Its newest definition is the Haskell'2010 standard.

Amongst the members of the designer group researchers from universities and institutes from all over the world can be found (J. Hughes, S. Peyton Jones, P. Hudak, K. Hammond, E. Meijer, J. Peterson, P. Wadler and others). For Haskell the following basic requirements were formulated:

- It should be suitable for education and research and for developing large scale application programs;
- The syntax and semantics should be formally defined;
- It should be freely accessible;
- The widely accepted basic principles should be complied with;
- It should serve as a guiding direction for the modern functional languages differing in greater or lesser extent from each other.

The central homepage of the language:
`http://www.haskell.org/`

18.1.16 Java

Java [Nyek08] started as an object-oriented language for programming tiny embedded digital TV controllers. This project actually died, but the language – thanks to the foreseeing of some developers – survived.

At the time of the birth of WWW Netscape took the fancy of the new language (not least because it enabled to fill static webpages with life) and started to include it into their browser. Seeing the intention of Netscape, Microsoft also built in Java support into their browser, which also helped the spreading of the Java language as a side effect.

The main characteristics of the language are the following:

- Object-orientation. In Java actually everything is an object (class). There are no global variables outside every class, like in C++.
- Inheritance. Every class has a well defined location within an inheritance tree rooted by the *Object* class. There is no multiple inheritance between classes, but there are **interface**s.
- Parallelism. Java supports the parallel execution of the program on multiple threads. These threads can be prioritized and grouped together.

- Easy learning. The language borrowed many constructs from other languages (mainly from C/C++), so the understanding of its syntax is not hard.
- Safety. The language tries to prevent the coding of incorrect programs with its structure. There is for example no pointer arithmetic (references are used instead).
- Exception handling. Thanks to its well developed exception handling the language enables the efficient handling of errors and exceptions helping so the software development process and safety.
- Automatic memory handling. The program can allocate memory, but cannot directly free it – this is the job of the built in GC (Garbage Collector). In this way referencing already freed objects can be avoided which also increases safety.
- Virtual machine. The Java programs are executed by a so called virtual machine, this makes the language totally portable.
- Standard class libraries. The Java programmers can rely on many built in services and functions (such as GUI, I/O, etc.) which makes the development much easier.

The central homepage of the language:
`http://java.sun.com/`

18.1.17 LISP

LISP is the first functional language. It was developed in the end of the 1950's by John McCarthy and his team on MIT. After the hardware boom in the 1970's it quickly gained enormous popularity, especially in artificial intelligence applications.

There are many variations; the near-standard Common Lisp (see [Ste90]) became widespread quite late. There is an object-oriented extension to Common Lisp called CLOS. This step was quite important, as frameworks – a precursor of object-oriented programming – have long been used successfully in artificial intelligence applications.

It is important to note, that LISP is an interpreted language.

The central homepage of the language:
`http://www.lisp.org/`

18.1.18 Maple

An old dream of people dealing with applied mathematics is the intelligent calculator which cannot only handle numbers, but also formulas. One of the tools for symbolic computation is Maple V. This language is one of the best usable formula manipulation systems.

The possibilities of Maple V spread from the functions as an intelligent calculator until the professional programming of the system, from the usage of basic calculus until the definition of numeric differential equations solving functions. It helps to handle different algebraic structures, function approximation and linear algebraic computations. The display of the results is supported graphically making it more visible and easier to understand.

The central homepage of the language:
`http://www.maplesoft.com/`

18.1.19 Modula-2

The language Modula-2 was developed by Niklaus Wirth, the designer of the ALGOL W, Pascal and Oberon in the 1970's in Zürich.

This language is easy to learn, meanwhile it is sophisticated enough for the development of large systems and also of realtime embedded systems.

The central homepage of the language:
`http://www.modula2.org/`

18.1.20 Modula-3

Modula-3 was developed in the end of the 1980's by DEC (Digital Equipment Corporation) and Olivetti in a joint venture based on Modula-2. The language supports module based software design, single inheritance, exception handling and parallelism. Modules can be parameterized with interfaces, so templates can be created.

The central homepage of the language:
`http://modula3.org/`

18.1.21 Objective-C

The programming language Objective-C is an object-oriented extension of the language C, just like C++. It was developed in the middle of the 1980's with the goal to extend the C language for objects but with the least amount of new language elements, and to support dynamic software development by enabling the programmer to disable some error checkings of the compiler at his/her own risk, resulting a more efficient code this way. Objective-C follows the object-oriented school of Smalltalk as opposed to C++, which follows those of SIMULA 67. It is a superset of the C programming language and provides object-oriented capabilities and a dynamic runtime. Objective-C inherits the syntax, primitive types, and flow control statements of C and adds syntax for defining classes and methods. It also adds language-level support for object graph management and object literals while providing dynamic typing and binding, deferring many responsibilities until runtime.

Like C++, Objective-C is also one of the ancestors of Java that is illustrated by the notion of *protocol* which was adopted by Java as **interface** to solve the problem of multiple inheritance.

The central homepage of the language:
`https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/`

18.1.22 Pascal

The language Pascal was developed by Niklaus Wirth in 1970 based on ALGOL. The language is easy to learn and supports well structured programming. It was spread worldwide pretty quickly, it is an efficient tool for programming education.

The central homepage of the language:
`http://www.merlyn.demon.co.uk/pascal.htm`

18.1.23 Perl

The first version of the language Perl [SP01] was created by Larry Wall in 1991 supported by NASA. The language was intended to be mainly used for text processing. Its efficiency in writing the code and executing the program was considered more important than its beauty. The name of the language suggests also this idea meaning "Practical Extraction and Report Language". But at first it was also called by Larry Wall as "Pathologically Eclectic Rubbish Lister".

A big advantage of the language is its platform independence: it exists for all variants of UNIX, Linux, VMS, OS/2 and Microsoft products.

Perl is an interpreted language compiled at loadtime. Originally it was created by Larry Wall for supporting system administration tasks, and to overcome the limitations of the existing tools. So the language is based on the following existing tools: C, sed, awk and sh. Knowledge of the LISP language can also help a lot to understand the list handling.

In Perl only the hardware limitations of the computer apply: a whole file can be read into a string variable (if there is enough memory), recursion depth is also unlimited (if having enough patient and memory). The access of associative arrays is accelerated by hashtables (which results in pretty efficient code). There is a very quick and flexible pattern searching algorithm for finding and replacing text. Binary data are also supported which can be composed into complex data structures. To support administrative task, database files can be associated to arrays whose structure can be defined by any skilled programmers. From the version nr. 5 on language constructs for modular programming and object-orientation are also supported.

The central homepage of the language:
`http://www.cpan.org/`

18.1.24 PHP

The official name was "PHP: Hypertext Preprocessor", however this suffix has long been outgrown. Today PHP is the most widespread content generator for HTML pages, the number of sites using PHP is over multiple millions. The reason for its popularity is due to the fact that this language (as reflected by its name) was from the beginning designed to be embedded into HTML pages, and the development environments are also designed to connect to web servers and run the programs through them, the result are shown as web pages.

As a result of its wide area of usage there is a great deal of extensions from database handling through image conversions up to GUI programming.

PHP is based on the Java, C and Perl languages. The similarity with the Perl language in its syntax and forming is very conspicuous (it could be characterized as a slightly modified Perl – there is also a similar Perl extension called Embedded Perl).

The central homepage of the language:
`http://www.php.net/`

18.1.25 PL/I

In the beginning of the 1960's PL/I was designed by combining the beneficial properties of ALGOL, FORTRAN and COBOL in the hope to become the *Programming Language No. 1.* The language has the block structure of ALGOL 60 and its control statements, the ***`FORMAT`*** input-output from FORTRAN, and the portable types and file handling from COBOL. Besides, pointers for dynamic data structures were introduced, as also flexible string operations and an error handling mechanism. The compilation of procedures on their own is also supported (see [Koz92]. The sequence of expanding subsets of the language was also defined (these were the SP/k, where k=1..8) to support the teaching of programming.

The central homepage of the language:
`http://www-03.ibm.com/software/products/us/en/plicompfami`

18.1.26 Python

The language Phyton, an efficient object-oriented script language is worth being mentioned because of its dynamic type system is being developed since 1991 by Guido van Rossum and team.

The central homepage of the language:
`http://www.python.org/`

18.1.27 Ruby

Ruby's creator, Yukihiro "Matz" Matsumoto carefully blended parts of his favorite languages (Perl, Smalltalk, Eiffel, Ada, and Lisp) to form a new language

that balanced functional programming with imperative programming. The result is a scripting language that is more powerful than Perl, and more object-oriented than Python.

In Ruby, everything – even numbers and other primitive types – is an object. Every bit of information and code can be given their own properties and actions following a pure object-oriented approach.

Ruby is seen as a flexible language, since it allows its users to freely alter its parts. Essential parts of it can be removed or redefined at will, existing parts can be added upon. The operators are syntactic sugar for methods and can be redefined as well.

Ruby has a wealth of other features, such as Java-like exception handling, a true mark-and-sweep garbage collector for all Ruby objects, dynamic extension library support if the OS allows, OS independent threading and is highly portable by being supported on all major operating systems.

The central homepage of the language:
`http://www.ruby-lang.org/`

18.1.28 SIMULA 67

SIMULA 67 (SIMUlation LAnguage) was designed by Dahl, Myhrhaug and Nygaard (see [DMN70]). By enhancing ALGOL 60 a language was created for efficient simulation of complex interactive systems.

This was the first object-oriented language, introducing the notions of the class and inheritance – or in SIMULA terminology: prefixing –, polymorphism and dynamic binding.

The central role in the implementation of simulations is played by the standard *SIMSET* class managing two way cyclic lists, and by its descendant, the *SIMULATION* class.

The central homepage of the language:
`http://www.engin.umd.umich.edu/CIS/course.des/cis400/simula/simula.html`

18.1.29 Smalltalk

In the beginning of the 1970's in the Xerox Palo Alto Research Center laboratories a team lead by Alan Kay, Daniel H. Ingalls and Adele Goldberg was working on a system to enable people an efficient and easy working with the computer. So Smalltalk was born, the second object-oriented language after SIMULA 67, but the first being completely designed according to this paradigm. In Smalltalk everything is an object – even classes, code blocks and the compiler itself. The object independent part of the language is minimal, even traditional control structures (branching, loops) are implemented as messages sent to objects; almost the assignment is the only operation not being treated like this.

Smalltalk was the first language with an integrated development environment, which was the base for developing later windowing systems.

The central homepage of the language:
`http://www.smalltalk.org/`

18.1.30 SML

The first typed functional language was ML (Meta Language). Originally it was the meta language of the LCF (Logic for Computable Functions) system designed for theorem proving in Edinburgh. The ML language was designed by R. Milner in the middle of the 70's, the SML (Standard ML) was created after Hope (Burstall, 1980) by Milner, Tofte and Harper between 1983 and 1990. The most recent version of the SML standard was introduced in 1997 ([MTHM97], [Har01] and [Han00]). Some variants of ML are Caml (INRIA, 1984-1990, the base language of the Coq theorem prover) and Objective Caml (the enhanced version of Caml Light, INRIA, 1990-). These ML variants are not purely functional languages, they also contain imperative language features (such as modifiable variables).

The central homepage of the language:
`http://www.standardml.org/`

18.1.31 SQL

Relational database handling and with it SQL were spread in the 80's. SQL is standing for Structured Query Language. This naming also shows, how crucial it is for relational database handling to have efficient queries on database tables.

The language SQL is divided into two parts: DML (Data Manipulation Language) for querying data and inserting, updating and deleting rows; and DDL (Data Definition Language) to create, modify and delete tables, views, indexes and other related database objects.

The (ANSI) standard SQL language only contains basic declarative elements; it does not support the implementation of complex algorithms. That is why numerous extensions and integrations with other languages were born to merge the benefits of imperative and procedural languages with the capabilities of SQL. This type of solutions is called embedded SQL.

There are two main types of embedded SQL: in one of them the SQL statements are contained in so called stored procedures within the database management system (Oracle PL/SQL, Microsoft Transact-SQL); the other approach connects to the database and uses SQL statements through an external API from a general purpose language (Pro C, ODBC, JDBC).

Because of the spreading of object relational and object-oriented database systems SQL was enhanced and served as a basis for newer database handling languages. Despite of this SQL is still the most widespread language for querying and manipulation data stored in databases.

The central homepage of the language:
`http://www.sql.org/`

18.1.32 Tcl

Tcl (Tool command language) is a simple interpreted language based on a few but versatile elements. It combines some elements from LISP, C and the unix C-shell.

Being a high level language it enables rapid and comfortable program development. All of this does not go at the expense of efficiency since Tcl lives in tight symbioses with C. If the basic functionality of an application is written in C, the higher level parts and GUI in Tcl, it could also be used as a macro language providing a great level of flexibility.

The most important extension of Tcl is the Tk toolkit, which enables Tcl applications to have GUIs.

Features not supported by this language: abstract data types, operator overloading, OOP, correctness proving, parallelism, persistence.

The central homepage of the language:
`http://www.tcl.tk/`

18.2 Codetables

18.2.1 The ASCII character table

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
0	0x00	NUL Null	32	0x20	Space	64	0x40	@	96	0x60	‘
1	0x01	SOH Start of heading	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	STX Start of text	34	0x22	"	66	0x42	B	98	0x62	b
3	0x03	ETX End of text	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	EOT End of transmission	36	0x24	$	68	0x44	D	100	0x64	d
5	0x05	ENQ Enquiry	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	ACK Acknowledge	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	BEL Bell	39	0x27	’	71	0x47	G	103	0x67	g
8	0x08	BS Backspace	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	HT Horizontal tab	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0a	LF Line feed, new line	42	0x2a	*	74	0x4a	J	106	0x6a	j
11	0x0b	VT Vertical tab	43	0x2b	+	75	0x4b	K	107	0x6b	k
12	0x0c	FF Form feed, new page	44	0x2c	,	76	0x4c	L	108	0x6c	l
13	0x0d	CR Carriage return	45	0x2d	-	77	0x4d	M	109	0x6d	m
14	0x0e	SO Shift out	46	0x2e	.	78	0x4e	N	110	0x6e	n
15	0x0f	SI Shift in	47	0x2f	/	79	0x4f	O	111	0x6f	o
16	0x10	DLE Data link escape	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1 Device control 1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2 Device control 2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3 Device control 3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4 Device control 4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK Negative acknowledge	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN Synchronous idle	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB End of trans. block	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN Cancel	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM End of medium	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1a	SUB Substitute	58	0x3a	:	90	0x5a	Z	122	0x7a	z
27	0x1b	ESC Escape	59	0x3b	;	91	0x5b	[	123	0x7b	{
28	0x1c	FS File separator	60	0x3c	<	92	0x5c	\	124	0x7c	|
29	0x1d	GS Group separator	61	0x3d	=	93	0x5d	]	125	0x7d	}
30	0x1e	RS Record separator	62	0x3e	>	94	0x5e	^	126	0x7e	~
31	0x1f	US Unit separator	63	0x3f	?	95	0x5f	_	127	0x7f	DEL

Table 18.1: The ASCII character table

18.2.2 The ISO 8859-1 (Latin-1) printable character table

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
160	0xa0		176	0xb0	°	192	0xc0	À	208	0xd0	Ð	224	0xe0	à	240	0xf0	ð
161	0xa1	¡	177	0xb1	±	193	0xc1	Á	209	0xd1	Ñ	225	0xe1	á	241	0xf1	ñ
162	0xa2	¢	178	0xb2	²	194	0xc2	Â	210	0xd2	Ò	226	0xe2	â	242	0xf2	ò
163	0xa3	£	179	0xb3	³	195	0xc3	Ã	211	0xd3	Ó	227	0xe3	ã	243	0xf3	ó
164	0xa4	¤	180	0xb4	´	196	0xc4	Ä	212	0xd4	Ô	228	0xe4	ä	244	0xf4	ô
165	0xa5	¥	181	0xb5	µ	197	0xc5	Å	213	0xd5	Õ	229	0xe5	å	245	0xf5	õ
166	0xa6	¦	182	0xb6	¶	198	0xc6	Æ	214	0xd6	Ö	230	0xe6	æ	246	0xf6	ö
167	0xa7	§	183	0xb7	·	199	0xc7	Ç	215	0xd7	×	231	0xe7	ç	247	0xf7	÷
168	0xa8	¨	184	0xb8	¸	200	0xc8	È	216	0xd8	Ø	232	0xe8	è	248	0xf8	ø
169	0xa9	©	185	0xb9	¹	201	0xc9	É	217	0xd9	Ù	233	0xe9	é	249	0xf9	ù
170	0xaa	ª	186	0xba	º	202	0xca	Ê	218	0xda	Ú	234	0xea	ê	250	0xfa	ú
171	0xab	«	187	0xbb	»	203	0xcb	Ë	219	0xdb	Û	235	0xeb	ë	251	0xfb	û
172	0xac	¬	188	0xbc	¼	204	0xcc	Ì	220	0xdc	Ü	236	0xec	ì	252	0xfc	ü
173	0xad	-	189	0xbd	½	205	0xcd	Í	221	0xdd	Ý	237	0xed	í	253	0xfd	ý
174	0xae	®	190	0xbe	¾	206	0xce	Î	222	0xde	Þ	238	0xee	î	254	0xfe	þ
175	0xaf	¯	191	0xbf	¿	207	0xcf	Ï	223	0xdf	ß	239	0xef	ï	255	0xff	ÿ

Figure 18.1: The printable characters in ISO 8859-1 (Latin-1) table missing from the ASCII table

18.2.3 The ISO 8859-2 (Latin-2) printable character table

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
160	0xa0		176	0xb0	°	192	0xc0	Ŕ	208	0xd0	Đ	224	0xe0	ŕ	240	0xf0	đ
161	0xa1	Ą	177	0xb1	ą	193	0xc1	Á	209	0xd1	Ń	225	0xe1	á	241	0xf1	ń
162	0xa2	˘	178	0xb2	˛	194	0xc2	Â	210	0xd2	Ň	226	0xe2	â	242	0xf2	ň
163	0xa3	Ł	179	0xb3	ł	195	0xc3	Ă	211	0xd3	Ó	227	0xe3	ă	243	0xf3	ó
164	0xa4	¤	180	0xb4	´	196	0xc4	Ä	212	0xd4	Ô	228	0xe4	ä	244	0xf4	ô
165	0xa5	Ľ	181	0xb5	ľ	197	0xc5	Ĺ	213	0xd5	Ő	229	0xe5	Í	245	0xf5	ő
166	0xa6	Ś	182	0xb6	ś	198	0xc6	Ć	214	0xd6	Ö	230	0xe6	ć	246	0xf6	ö
167	0xa7	§	183	0xb7	ˇ	199	0xc7	Ç	215	0xd7	×	231	0xe7	ç	247	0xf7	÷
168	0xa8	¨	184	0xb8	¸	200	0xc8	Č	216	0xd8	Ř	232	0xe8	č	248	0xf8	ř
169	0xa9	Š	185	0xb9	š	201	0xc9	É	217	0xd9	Ů	233	0xe9	é	249	0xf9	ů
170	0xaa	Ş	186	0xba	ş	202	0xca	Ę	218	0xda	Ú	234	0xea	ę	250	0xfa	ú
171	0xab	Ť	187	0xbb	ť	203	0xcb	Ë	219	0xdb	Ű	235	0xeb	ë	251	0xfb	ű
172	0xac	Ź	188	0xbc	ź	204	0xcc	Ě	220	0xdc	Ü	236	0xec	ě	252	0xfc	ü
173	0xad	-	189	0xbd	˝	205	0xcd	Í	221	0xdd	Ý	237	0xed	í	253	0xfd	ý
174	0xae	Ž	190	0xbe	ž	206	0xce	Î	222	0xde	Ţ	238	0xee	î	254	0xfe	ţ
175	0xaf	Ż	191	0xbf	ż	207	0xcf	Ď	223	0xdf	ß	239	0xef	ď	255	0xff	˙

Figure 18.2: The printable characters in ISO 8859-2 (Latin-2) table missing from the ASCII table

18.2.4 The IBM Codepage 437

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
128	0x80	Ç	160	0xa0	á	192	0xc0	└	224	0xe0	α
129	0x81	ü	161	0xa1	í	193	0xc1	┴	225	0xe1	β
130	0x82	é	162	0xa2	ó	194	0xc2	┬	226	0xe2	Γ
131	0x83	â	163	0xa3	ú	195	0xc3	├	227	0xe3	π
132	0x84	ä	164	0xa4	ñ	196	0xc4	─	228	0xe4	Σ
133	0x85	à	165	0xa5	Ñ	197	0xc5	┼	229	0xe5	σ
134	0x86	å	166	0xa6	ª	198	0xc6	╞	230	0xe6	µ
135	0x87	ç	167	0xa7	º	199	0xc7	╟	231	0xe7	γ
136	0x88	ê	168	0xa8	¿	200	0xc8	╚	232	0xe8	Φ
137	0x89	ë	169	0xa9	⌐	201	0xc9	╔	233	0xe9	θ
138	0x8a	è	170	0xaa	¬	202	0xca	╩	234	0xea	Ω
139	0x8b	ï	171	0xab	½	203	0xcb	╦	235	0xeb	δ
140	0x8c	î	172	0xac	¼	204	0xcc	╠	236	0xec	∞
141	0x8d	ì	173	0xad	¡	205	0xcd	═	237	0xed	φ
142	0x8e	Ä	174	0xae	«	206	0xce	╬	238	0xee	ε
143	0x8f	Å	175	0xaf	»	207	0xcf	╧	239	0xef	∩
144	0x90	É	176	0xb0	▒	208	0xd0	╨	240	0xf0	≡
145	0x91	æ	177	0xb1	▒	209	0xd1	╤	241	0xf1	±
146	0x92	Æ	178	0xb2	▓	210	0xd2	╥	242	0xf2	≥
147	0x93	ô	179	0xb3	│	211	0xd3	╙	243	0xf3	≤
148	0x94	ö	180	0xb4	┤	212	0xd4	╘	244	0xf4	⌠
149	0x95	ò	181	0xb5	╡	213	0xd5	╒	245	0xf5	⌡
150	0x96	û	182	0xb6	╢	214	0xd6	╓	246	0xf6	÷
151	0x97	ù	183	0xb7	╖	215	0xd7	╫	247	0xf7	≈
152	0x98	ÿ	184	0xb8	╕	216	0xd8	╪	248	0xf8	°
153	0x99	Ö	185	0xb9	╣	217	0xd9	┘	249	0xf9	·
154	0x9a	Ü	186	0xba	║	218	0xda	┌	250	0xfa	•
155	0x9b	¢	187	0xbb	╗	219	0xdb	█	251	0xfb	√
156	0x9c	£	188	0xbc	╝	220	0xdc	▄	252	0xfc	ⁿ
157	0x9d	¥	189	0xbd	╜	221	0xdd	▌	253	0xfd	²
158	0x9e	Pt	190	0xbe	╛	222	0xde	▐	254	0xfe	■
159	0x9f	f	191	0xbf	┐	223	0xdf	▀	255	0xff	

Figure 18.3: The characters in IBM Codepage 437 from the position 128

18.2.5 The EBCDIC character table

Dec	Hex	Chr		Dec	Hex	Chr		Dec	Hex	Chr		Dec	Hex	Chr	
0	0x00	NUL	Null	64	0x40	␣		128	0x80			192	0xc0	{	
1	0x01	SOH	Start of heading	65	0x41			129	0x81	a		193	0xc1	A	
2	0x02	STX	Start of text	66	0x42			130	0x82	b		194	0xc2	B	
3	0x03	ETX	End of text	67	0x43			131	0x83	c		195	0xc3	C	
4	0x04	PF	Punch off	68	0x44			132	0x84	d		196	0xc4	D	
5	0x05	HT	Horizontal tab	69	0x45			133	0x85	e		197	0xc5	E	
6	0x06	LC	Lower case	70	0x46			134	0x86	f		198	0xc6	F	
7	0x07	DEL	Delete	71	0x47			135	0x87	g		199	0xc7	G	
8	0x08	GE		72	0x48			136	0x88	h		200	0xc8	H	
9	0x09	RLF		73	0x49			137	0x89	i		201	0xc9	I	
10	0x0a	SMM	Start of manual message	74	0x4a	¢		138	0x8a			202	0xca		
11	0x0b	VT	Vertical tab	75	0x4b	.		139	0x8b			203	0xcb		
12	0x0c	FF	Form feed	76	0x4c	<		140	0x8c			204	0xcc		
13	0x0d	CR	Carriage return	77	0x4d	(		141	0x8d			205	0xcd		
14	0x0e	SO	Shift out	78	0x4e	+		142	0x8e			206	0xce		
15	0x0f	SI	Shift in	79	0x4f	|		143	0x8f			207	0xcf		
16	0x10	DLE	Data link escape	80	0x50	&		144	0x90			208	0xd0	}	
17	0x11	DC1	Device control 1	81	0x51			145	0x91	j		209	0xd1	J	
18	0x12	DC2	Device control 2	82	0x52			146	0x92	k		210	0xd2	K	
19	0x13	TM	Tape mark	83	0x53			147	0x93	l		211	0xd3	L	
20	0x14	RES	Restore	84	0x54			148	0x94	m		212	0xd4	M	
21	0x15	NL	New line	85	0x55			149	0x95	n		213	0xd5	N	
22	0x16	BS	Backspace	86	0x56			150	0x96	o		214	0xd6	O	
23	0x17	IL	Idle	87	0x57			151	0x97	p		215	0xd7	P	
24	0x18	CAN	Cancel	88	0x58			152	0x98	q		216	0xd8	Q	
25	0x19	EOM	End of medium	89	0x59			153	0x99	r		217	0xd9	R	
26	0x1a	CC	Cursor control	90	0x5a	!		154	0x9a			218	0xda		
27	0x1b	CU1	Customer use 1	91	0x5b	$		155	0x9b			219	0xdb		
28	0x1c	IFS	Interchange file separator	92	0x5c	*		156	0x9c			220	0xdc		
29	0x1d	IGS	Interchange group separator	93	0x5d	)		157	0x9d			221	0xdd		
30	0x1e	IRS	Interchange record	94	0x5e	;		158	0x9e			222	0xde		
31	0x1f	IUS	Interchange unit separator	95	0x5f	¬		159	0x9f			223	0xdf		

Figure 18.4: The EBCDIC character table

Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr	Dec	Hex	Chr
32	0x20	DS Digit select	96	0x60	-	160	0xa0		224	0xe0	\
33	0x21	SOS Start of significance	97	0x61	/	161	0xa1	~	225	0xe1	
34	0x22	FS Field separator	98	0x62		162	0xa2	s	226	0xe2	S
35	0x23		99	0x63		163	0xa3	t	227	0xe3	T
36	0x24	BYP Bypass	100	0x64		164	0xa4	u	228	0xe4	U
37	0x25	LF Line feed	101	0x65		165	0xa5	v	229	0xe5	V
38	0x26	ETB End of transmission block	102	0x66		166	0xa6	w	230	0xe6	W
39	0x27	ESC Escape	103	0x67		167	0xa7	x	231	0xe7	X
40	0x28		104	0x68		168	0xa8	y	232	0xe8	Y
41	0x29		105	0x69		169	0xa9	z	233	0xe9	Z
42	0x2a	SM Set mode	106	0x6a		170	0xaa		234	0xea	
43	0x2b	CU2 Customer use 2	107	0x6b	,	171	0xab		235	0xeb	
44	0x2c		108	0x6c	%	172	0xac		236	0xec	
45	0x2d	ENQ Enquiry	109	0x6d	_	173	0xad		237	0xed	
46	0x2e	ACK Acknowledge	110	0x6e	>	174	0xae		238	0xee	
47	0x2f	BEL Bell	111	0x6f	?	175	0xaf		239	0xef	
48	0x30		112	0x70		176	0xb0		240	0xf0	0
49	0x31		113	0x71		177	0xb1		241	0xf1	1
50	0x32	SYN Synchronous idle	114	0x72		178	0xb2		242	0xf2	2
51	0x33		115	0x73		179	0xb3		243	0xf3	3
52	0x34	PN Punch on	116	0x74		180	0xb4		244	0xf4	4
53	0x35	RS Reader stop	117	0x75		181	0xb5		245	0xf5	5
54	0x36	UC Upper case	118	0x76		182	0xb6		246	0xf6	6
55	0x37	EOT End of transmission	119	0x77		183	0xb7		247	0xf7	7
56	0x38		120	0x78		184	0xb8		248	0xf8	8
57	0x39		121	0x79		185	0xb9		249	0xf9	9
58	0x3a		122	0x7a	:	186	0xba		250	0xfa	|
59	0x3b	CU3 Customer use 3	123	0x7b	#	187	0xbb		251	0xfb	
60	0x3c	DC4 Device control 4	124	0x7c	@	188	0xbc		252	0xfc	
61	0x3d	NAK Negative acknowledge	125	0x7d	’	189	0xbd		253	0xfd	
62	0x3e		126	0x7e	=	190	0xbe		254	0xfe	
63	0x3f	SUB Substitute	127	0x7f	"	191	0xbf		255	0xff	EO

Table 18.2: The EBCDIC character table (continued)

Bibliography

[Abe03] P. Abercrombie. *jContractor Crash Course*, 2003. http://jcontractor.sourceforge.net/doc/crashcourse.html

[Abr10] D. Abrahams. *Exception-Safety in Generic Components*, 2010. http://www.boost.org/community/exception_safety.html

[AC98] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, corrected second printing edition, 1998.

[Ack79] W. B. Ackermann et al. A Value-Oriented Algorithmic Language. Technical report, MIT Laboratory of Computer Science, June 1979. TR-218

[Ada12] *Ada Reference Manual, ISO/IEC 8652:2012(E)*, 2012. http://www.ada-auth.org/standards/12rm/html/RM-TTL.html

[Ada83] *Ada '83 Language Reference Manual*, 2003. http://archive.adaic.com/standards/83lrm/html/lrm-02-01.html

[Ada95] *Annotated Ada Reference Manual, ISO/IEC 8652:1995*, 1995. http://www.ada95.com

[Agh87] G. Agha. *Actors*. The MIT Press, 1987.

[Ald07] J. Aldrich. *Software Analysis*, 2007. http://www.cs.cmu.edu/~aldrich/courses/654-sp08/slides/9-hoare.pdf

[Ale01] A. Alexandrescu. *Modern C++ Design (Generic Programming and Design Patterns Applied)*. Addison-Wesley Professional, Reading, Mass., 2001.

[Amd67] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *spring joint computer conference, AFIPS'67*, Atlantic City, New Jersey, April 18-20 1967.

[Ame87] P. America. *POOL-T: A parallel object-oriented language*. MIT Press, Cambridge, Massachusetts, object-oriented concurrent programming edition, 1987.

[And91] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, 1991.

[Ang97] E. Angster. Az objektumorientált tervezés és programozás alapjai. Technical report, Gábor Dénes Főiskola, Budapest, 1997.

[AO97] K. R. Apt and E-R. Olderog. *Verification of Sequential and Concurrent Program*. Springer-Verlag, 1997.

[AOSD12] *Homepage of the Aspect-Oriented Software Development*, 2012. `http://www.aosd.net/`

[AP80] *Demeter: Aspect-Oriented Software Development*, 1980. `http://www.ccs.neu.edu/research/demeter/`

[APM07] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating Static Analysis Defect Warnings On Production Software. In 7^{th} *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 2007.

[Arv78] Arvind et al. *An Asynchronous Programming Language for a Large Multiprocessor Machine*. TR114a, Dept ISC, UC Irvine, Dec 1978.

[Aug99] L. Augustsson. Cayenne. Technical report, Göteburgs Universitet, Sweden, 1999. `http://www.math.chalmers.se/~augustss/cayenne/`

[AW00] P. Achten and M. Wierich. A Tutorial to the Clean Object I/O Library. Technical report, University of Nijmegen, Netherlands, 2000. `http://www.cs.kun.nl/~clean`

[BA06] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, 2^{nd} edition, 2006.

[Bak74] T. Bakos. *A COBOL programozási nyelv*. Műszaki Könyvkiadó, Budapest, 1974.

[Bar84] H. P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics*. North-Holland, Amsterdam, 1984.

[Bar92] G. Barrett. *Occam3 Reference Manual*. Bristol, 1992.

[Bar96] J. Barnes. *Programming in Ada95*. Springer-Verlag, New-York, 1996.

[Bar01] G. Barr. *Error - Error/Exception Handling in an OO-ish Way*, 2001. `http://search.cpan.org/author/UARUN/Error-0.15/Error.pm`

[Bar12] J. Barnes. *A brief introduction to Ada 2012*, 2012. `http://www.adacore.com/uploads/technical-papers/Ada2012_Rationale_Chp1_contracts_and_aspects.pdf`

[Ben06] J. Bender. *Mini-Bibliography on Modules for Functional Programming Languages*. ReadScheme.org, 2006.

[BJ66] D. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, 1966.

[BJK95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, 1995.

[Blo01] J. Bloch. *Effective Java Programming Language Guide*. Addison-Wesley Professional, Reading, Mass., 2001.

[Bor89] Borland. *Turbo Pascal 5.5 - Object-Oriented Programming Guide*, 1989.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM SIGPLAN Notices*, volume 33, pages 183–200. ACM, 1998.

[Bow02] J. Bowen. *Logic Programming*, 2002. `http://www.comlab.ox.ac.uk/archive/logic-prog.html`

[BP90] G. Blaschek and G. Pomberger. *Introduction to programming with Modula-2*. Springer-Verlag, Berlin, 1990.

[Bra99] M. Brandel. *1963: ASCII Debuts*, 1999. `http://www.bobbemer.com/brandela.htm`

[Bru02] K. B. Bruce. *Foundations of Object-Oriented Languages and Types*. MIT Press, Cambridge, Massachusetts, 2002.

[Bud91] T. Budd. *An Introduction to Object-Oriented Analysis*. Addison-Wesley, Reading, Mass., 1991.

[BW79] D. F. Brailsford and A. N. Walker. *Introductory Algol 68 programming*. Ellis Horwood series in Computer Science, Chichester, 1979.

[BW90] M. Barr and C. Wells. *Category Theory for Computer Science*. Prentice Hall, London, 1990.

[BW96] L. Böszörményi and C. Weich. *Programmieren mit Modula-3*. Springer-Verlag, Berlin, 1996.

[C4Ja] *Contract4J: Design by Contract for Java*, 2010. `http://www.polyglotprogramming.com/contract4j`

[C4Jb] *Contracts for Java (C4J)*, 2013. `http://c4j-team.github.io/C4J/`

[C90] *Programming Language C ISO/IEC 9899:1990*, 1990.

[Can00] M. Cantù. *Delphi 5 Mesteri szinten*. Kiskapu Kiadó, Budapest, 2000.

[Car12] M. Carlsson et al. SICStus Prolog 4.2.3 User's Manual. Technical report, Swedish Institute of Computer Science, 2012. `http://www.sics.se/isl/sicstuswww/site/documentation.html`

[Car13] M. Carlsson et al. Functional logic programming. Technical report, Wikipedia, 2013. `http://en.wikipedia.org/wiki/Functional_logic_programming`

[Cat01] B. Catambay. *Pascal standards. Pascal ISO 7185:1990, Extended Pascal ISO 10206:1990*, 2001. `http://www.pascal-central.com/standards.html`

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, Reading, Mass., 5th edition, 2000.

[CF01] *Composition Filters*, 2001. `http://trese.cs.utwente.nl/composition_filters/`

[CJS71] Jr. E. G. Coffman, M. J.Elphick, and A. Shoshani. System Deadlocks. *ACM Computing Surveys*, 2(3):67–78, 1971. doi:10.1145/356586.356588

[CLI12] *ECMA-335: Common Language Infrastructure (CLI)*, 2012. `http://www.ecma-international.org/publications/standards/Ecma-335.htm`

[CM84] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems*, 1984.

[CM03] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer, 2003.

[CMP95] C. Clack, C. Myers, and E. Poon. *Programmieren in Miranda*. Prentice Hall, München, 1995.

[Cob13a] *Cobra Programming Language Coding for Quality*, 2013. `http://cobra-language.com/docs/quality/`

[Cob13b] *Cobra Programming Language Contracts*, 2013. `http://cobra-language.com/trac/cobra/wiki/Contracts`

[Code13] *Code Contracts User Manual*, 2013. `http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf`

[Col82] A. Colmerauer. Prolog and Infinite Trees. In K. L. Clark and S. A. Tarnlund (eds.): *Logic Programming*, pages 231–251. Academic Press, London, 1982.

[Con63] M. E. Conway. Design of a Separable Transition-Diagram Compiler. *Communications of the ACM*, 7(6):396–408, July 1963.

[Cor09] T. H. Cormen et al. *Introduction to Algorithms*. MIT Press, 2009.

[CPP98] *C++ Standard ISO/IEC 14882:1998(E)*, 1998. `http://www.comnets.rwth-aachen.de/doc/c++std/`

[Cso96] Z. Csörnyei. Bevezetés a fordítóprogramok elméletébe, I-II. Egyetemi jegyzet, ELTE TTK ITCS, Budapest, 1996.

[Csref03] *C# Programmer's Reference*, 2003. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csref/html/vcoriCProgrammersReference.asp`

[CT09] F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly Media, 2009.

[CW85] L. Cardelli and P. Wegner. On Understanding of Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Cyb83] *CDC Cyber 200 FORTRAN Reference Manual*. Sunnyvale, California, 1983.

[Czy98a] R. Czyborra. *The ISO 8859 Alphabet Soup*, 1998. `http://czyborra.com/charsets/iso8859.html`

[Czy98b] R. Czyborra. *Unicode Transformation Formats: UTF-8 and Co.*, 1998. `http://czyborra.com/utf/`

[DD05] H. M. Deitel and P. J. Deitel. *Java How to Program, 6/e*, 2005. `http://www.deitel.com/articles/java_tutorials/20060106/Assertions.html` pp.664-665. Electronically reproduced by permission of Pearson Education, Inc.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, New York, 1972.

[DEDC96] P. Deransart, A. A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard (Reference Manual)*. Springer-Verlag, Berlin, 1996.

[Del99] *Borland Delphi 5 Quick Start*, 1999. `http://info.borland.com/techpubs/delphi/delphi5/`

[DG80] Donzeau-Gouge et al. *Formal Definition of the ADA Programming Language*. INRIA, Honeywell, Cii Honeywell Bull, corrected second printing edition, 1980.

[Dij71] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 2(1):115–138, June 1971.

[Dij74] E. W. Dijkstra. EWD-1000. Technical report, Center for American History, University of Texas, Austin, 1974.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, New York, 1976.

[Dij82] E. W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*, chapter The mathematics behind the Banker's Algorithm (EWD-623), pages 308–312. Springer-Verlag, 1982. ISBN 0-387-90652-5

[DL86] D. DeGroot and G. Lindstrom. *Logic Programming: Functions, Relations and Equations*. Prentice Hall, Englewood Cliffs, NY, 1986.

[Dlng13] *D Programming Language, Contract Programming*, 2013. `http://dlang.org/dbc.html`

[DM93] P. Deransart and J. Maluszyński. *A Grammatical View of Logic Programming*. MIT Press, Cambridge, Massachusetts, 1993.

[DMN70] O. J. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA - Common Base Language*. Norwegian Computing Center, Oslo, revised edition, October 1970.

[Eck00] B. Eckel. *Thinking in C++, vol. I.: Introduction to Standard C++*. Prentice Hall, New York, 2nd edition, 2000.

[Ecl03] *Homepage of the Eclipse project*, 2003. `http://www.eclipse.org/`

[ECM06] *ECMA-367 Eiffel: Analysis, Design and Programming Language*, 2006. `http://www.ecma-international.org/publications/standards/Ecma-367.htm`

[EH03] B. Eckel and A. Hejlsberg. *The Trouble with Checked Exceptions*, 2003. `http://www.artima.com/intv/handcuffs.html`

[Eif13] *Concurrent Eiffel with Simple Concurrent Object-Oriented Programming*, 2013. `http://docs.eiffel.com/book/solutions/concurrent-eiffel-scoop`

[Eiff] *Eiffel Documentation*, 2013. `http://docs.eiffel.com/`

[FFMS01] C. Fournet, F. Le Fessant, L. Maranget, and A. Schmitt. *The JoCaml language - Documentation and User's Manual*. INRIA Rocquencourt, Jan. 2001.

[FGN90] I. Fekete, T. Gregorics, and S. Nagy. *Bevezetés a mesterséges intelligenciába*. Műszaki Könyvkiadó, Budapest, 1990.

[Fla94] P. Flach. *Simply Logical. Intelligent Reasoning by Example*. John Wiley and Sons, 1994.

[Fly72] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, C(21):948, 1972.

[Fort03] *J3 - Fortran Standards*, 2003. `http://www.j3-fortran.org`

[Fot83] Á. Fóthi. Bevezetés a programozáshoz. Egyetemi jegyzet, ELTE TTK, Budapest, 1983.

[Gam95] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[GD68] F. Genuys and E. W. Dijkstra. *Programming Languages*, chapter Co-operating Sequential Processes, pages 43–112. Academic Press, New York, 1968.

[GG96] R. Gabriel and P. Graham. *The End of History and the Last Programming Language, in patterns of Software*. Oxford University Press, New York, 1996.

[GI90] F. Grund and W. Issel. *PL/I Programmierung*. Hüthig Verlag, Heidelberg, 1990.

[GK99] L. Gulyás and T. Kozsik. The Use of Aspect-Oriented Programming in Scientific Simulations. In Jaan Penjam (ed.): *Software Technology, Fenno-Ugric Symposium FUSST'99 Proceedings*, pages 17–28, Tallin, Estonia, August 19-21 1999.

[GM94] C. A. Gunter and J. C. Mitchell. *Theoretical Aspects of Object-Oriented Programming*. MIT Press, Cambridge, Massachusetts, 1994.

[GM10] M. Gabbrielli and S. Martini. *Programming Languages: Principles and Paradigms*. Springer-Verlag, 2010.

[Gom97] B. Gomes et al. A Language Manual for Sather 1.1. Tr 97-037, International Computer Science Institute, Oct 1997.

[GPB06] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, Reading, Mass., 2006.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, Mass., 1983.

[Gra04] Graham. *Hacker and Painters: Big Ideas from Computer Age*. O'Reilly Media, 2004.

[Gro10] J. van Groningen, T. van Noort, P. Achten, P. Koopman, and R. Plasmeijer. Exchanging sources between Clean and Haskell: a double-edged front end for the Clean compiler. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 49–60, New York, NY, USA, 2010. ACM.

[GT99] L. Gulyás and G. Tatai. *Ágensek és multi-ágens rendszerek*, pages 709–754. Aula Kiadó, Budapest, 1999.

[HAKP99] Z. Horváth, P. Achten, T. Kozsik, and R. Plasmeijer. Verification of the Temporal Properties of Dynamic Clean Processes. In *Proceedings of the 11th International workshop on the Implementation of Functional Languages, IFL'99*, pages 203–218, Lochem, The Netherlands, 1999.

[Han72] P. B. Hansen. Structured multiprogramming. *Communications of the ACM*, 15:574–578, 1972.

[Han94] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *The Journal of Logic Programming*, 1994.

[Han00] P. Hanák. *Deklaratív programozás*. Budapest, 2000. Oktatási segédlet, BME

[Har98] S. J. Hartley. *Concurrent Programming: The Java Programming Language*. Oxford University Press, Oxford, 1998.

[Har01] R. Harper. Programming in Standard ML. Working draft, Carnegie Mellon University, Spring Semester 2001. `http://www.cs.cmu.edu/~rwh/smlbook/`

[HBS73] C. Hewitt, P. Bishop, and R. Steiger. *A Universal Modular Actor Formalism for Artificial Intelligence*, 1973. `http://dli.iiit.ac.in/ijcai/IJCAI-73/PDF/027B.pdf`

[HC83] R. C. Holt and J. R. Cordy. The TURING language report. Technical report csrg-153, Computer Systems Research Institute, University of Toronto, December 1983.

[Hen86] P. V. Henterick. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, Massachusetts, 1986.

[HFP99] P. Hudak, J. H. Fasel, and J. Peterson. *A Gentle Introduction to Haskell 98*, 1999. `http://www.haskell.org`

[HHZ02] H. Hegedűs †, Z. Horváth, and V. Zsók. A Haskell és a Clean nyelv összehasonlító elemzése. In *Informatika a Felsőoktatásban*, pages 1075–1084. Kiadó, Debrecen, 2002.

[HK02] Z. Horváth and T. Kozsik. Certified Proven Property Carrying Code (CPPCC) - Safe Functional Mobile Code. In *ECOOP Workshops 1 and 18*, Malaga, 2002.

[Hoa73] C. A. R. Hoare. Computer Science (New lecture series). Technical report, Queen's University Belfast, 1973.

[Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, New York, 1985.

[Hoa04] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 2004.
ISBN 0-13-153271-5

[Hol84] R. C. Holt. TURING: An inside look at the genesis of a programming language. *Computer World*, 18(20), May 1984.

[Hor94] E. Horowitz. *Fundamentals of Programming Languages*. Computer Science Press, Rockville, Maryland, 2nd edition, 1994.

[How03] M. Howard et al. *Type-safe covariance: Competent compilers can catch all catcalls*, 2003.
`http://se.ethz.ch/~meyer/ongoing/covariance/recast.pdf`

[HP79] R. Herschel and F. Pieper. *PASCAL: systematische Darstellung von PASCAL und Concurrent PASCAL für den Anwender*. Oldenbourg Verlag, München-Wien, 1979.

[HT99] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

[Hud89] P. Hudak. Conception, Evolution and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, Sept 1989.

[Hut87] N. C. Hutchinson et al. The Emerald Programming Language Report. Technical report, Department of Computer Science, University of Washington, Seattle, Oct. 1987.

[HZSP03] Z. Horváth, V. Zsók, P. Serrarens, and R. Plasmeijer. Parallel Elementwise Processable Functions in Concurrent Clean. *Computers and Mathematics with Applications*, 2003.

[IEE07] IEEE. *Unit Testing Concurrent Software*, Atlanta GA, Nov 5-9 2007.

[IPfaq03] *Intentional Programming Frequently Asked Questions*, 2003.
`http://www.omniscium.com/?page=IntentionalFaq`

[ISO00] International Organization for Standardization. *Information technology - Programming languages - Prolog - Part 2: Modules*, 2000.
ISO/IEC 13211-2

[ISO95] International Organization for Standardization. *Information technology - Programming languages - Prolog - Part 1: General core*, 1995.

ISO/IEC 13211-1

[Jac75] M. A. Jackson. *Principles of Programming Design*. Academic Press, London, 1975.

[Jass02] *Programming With Assertions*, 2002.
`http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html`

[Java13] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java® Specification Java SE 7 Edition*, 2013.
`http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`

[JML13] *Java Modeling Language*, 2013.
`http://en.wikipedia.org/wiki/Java_Modeling_Language`

[JMLc06] G. T. Leavens and C. Yoonsik. *Design by Contract with JML*, 2006.
`ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`

[Jos99] N. M. Josuttis. *The C++ Standard Library. A Tutorial and Reference*. Addison-Wesley, Reading, Mass., 1999.

[JW74] K. Jensen and N. Wirth. *Pascal - User Manual and Report*. Springer-Verlag, New York, 1974.

[Kes96] M. H. G. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*. Phd thesis, Catholic University of Nijmegen, 1996.

[Kicz01] G. Kiczales et al. An Overview of AspectJ. In *ECOOP 2001, Budapest*, pages 327–353, Berlin, 2001. Springer-Verlag.

[Kicz97] G. Kiczales et al. Aspect-Oriented Programming. In *ECOOP 1997, Jyväskylä, Finland*, Berlin, 1997. Springer-Verlag.

[Kno93] K. T. Knoll. Risk Management In Fly-By-Wire Systems. *NASA STI/Recon*, 1993.

[Knu05] D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 2: Generating All Tuples and Permutations*. Addison-Wesley Professional, 2005.

[Knu87] D. E. Knuth. *The Art of Computer Programming*, volume 1-3. Addison-Wesley, Reading, Massachusetts, 1987.

[Kor02] J. Korpela. *A Tutorial on Character Code Issues*, 2002.
`http://www.cs.tut.fi/~jkorpela/chars.html`

[Kow79] B. Kowalski. *Logic for Problem Solving*. North-Holland, New York, Amsterdam, Oxford, 1979.

[Koz92] S. Kozics. Az ALGOL 60, a FORTRAN, a COBOL és a PL/I programozási nyelvek. Egyetemi jegyzet, ELTE TTK, Budapest, 1992.

[Koz93] S. Kozics. Az Ada programozási nyelv. Egyetemi jegyzet, ELTE TTK, Budapest, 1993.

[KR89] B. W. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1989. `http://cm.bell-labs.com/cm/cs/cbook/`

[KV06] L. Kozma and L. Varga. *A szoftvertechnológia elméleti kérdései*. Eötvös Kiadó, 2006. ISBN: 9634636489

[Lak96] J. Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, Reading, Mass., 1996.

[Lam08] R. Lämmel. Google's MapReduce programming model-Revisited. *Science of Computer Programming 70.1*, 2008.

[Lam82] G. Lamprecht. *Einführung in die Programmiersprache SIMULA*. Springer Fachmedien, Wiesbaden, 1982.

[Lam77] B. W. Lampson et al. Report on the programming language Euclid. *ACM SIGPLAN Notices*, 12(2), February 1977.

[Le11] Nhat Minh Le. Contracts for Java: A Practical Framework for Contract Programming. Technical report, Google Switzerland GmbH, Grenoble, January 2011.

[Lei09] C. E. Leiserson. The Cilk++ concurrency platform. In *In Proceedings of the 46th Annual Design Automation Conference (DAC '09)*, pages 522–527, New York, NY, USA, 2009. `http://doi.acm.org/10.1145/1629911.1630048`

[LG96] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, Massachusetts, 1996.

[Lis93] B. Liskov. A history of CLU. *ACM SIGPLAN Notices*, 28(3):133–147, 1993. `http://citeseer.nj.nec.com/liskov92history.html`

[Lis00] R. Lischner. *Delphi in a Nutshell*. O'Reilly and Associates, Sebastopol, CA, March 2000.

[Lis81] B. Liskov et al. CLU Reference Manual. *LNCS*, 114, 1981.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 2nd edition, 1987.

[Loo12] R. Loogen. Eden – Parallel Functional Programming in Haskell. In V. Zsók, Z. Horváth, and R. Plasmeijer (eds.): *4th Central European Functional Programming Summer School (CEFP), Budapest, Hungary, June 14–24, 2011, Revisited Selected Lectures*, volume 7241 of *LNCS*, pages 142–206. Springer-Verlag, 2012.

[Lop97] C. V. Lopez. *D: A Language Framework for Distributed Programming*. Phd thesis, Graduate School of College of Computer Science, Northeastern University, Boston, MA, 1997.

[Lou11] K. C. Louden. *Programming Languages: Principles and Practices*. Course Technology, 3rd edition, 2011.

[LSS87] J. Loeck, K. Sieber, and R. D. Stansifer. *The Foundations of Program Verification*. John Wiley and Sons, New York, 2nd edition, 1987.

[LT01] H. W. Loidl and P. T. Trinder. A Gentle Introduction to GPH. Technical report, Department of Computing and Electrical Engineering, Heriot-Watt University, July 2001.

[LV77] Gy. Lőcs and J. Vigassy. *A FORTRAN programozási nyelv*. Műszaki Könyvkiadó, Budapest, 1977.

[LYBB13] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley, 2013.
`http://docs.oracle.com/javase/specs/jvms/se7/html/index.html`

[Mar93] Gy. Marx. Kemény János. *Fizikai Szemle*, 5:169, 1993.

[Mar02] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, Englewood Cliffs, NY, 2002. ISBN 0-13-597444-5

[Mar08] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, Englewood Cliffs, NY, 2008. ISBN-10: 0-13-235088-2

[Mar12] S. Marlow. Parallel and Concurrent Programming in Haskell. In V. Zsók, Z. Horváth, and R. Plasmeijer (eds.): *4th Central European Functional Programming Summer School (CEFP), Budapest, Hungary, June 14–24, 2011, Revisited Selected Lectures*, volume 7241 of *LNCS*, pages 339–401. Springer-Verlag, 2012.

[McC85] J. McCarthy et al. *Lisp 1.5 Programmer's Manual*. MIT Press, Cambridge, Massachusetts, 2nd edition, 1985.

[MDSC02] *Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces*, 2002.
`http://www.research.ibm.com/hyperspace/`

[MEP01] M. de Mol, M. van Eekelen, and R. Plasmeijer. SPARKLE: A Functional Theorem Prover. In T. Arts and M. Mohnen (eds.): *Proceedings of the 13th International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers, Älvsjö, Sweden*, pages 55–71, Berlin, September 24-26 2001. Springer-Verlag.

ftp://ftp.cs.kun.nl/pub/Clean/papers/2002/molm2002-TheoremProvingFP.ps.gz

[Mey91] B. Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1991.

[Mey00] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 2nd edition, 2000.

[MHB99] K. Murat, U. Hölzle, and J. Bruno. *jContractor: A Reflective Java Library to Support Design By Contract*, 1999.
http://www.eecs.northwestern.edu/~robby/contract-reading-list/jContractor.pdf

[Mic03] Sun Microsystems. *The Java Language Specification*, 2003.
http://www.javasoft.com/docs/

[Mit98] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, Massachusetts, 1998.

[MMPN93] O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, Mass., 1993.

[Mor11] D. Morgan et al. *A quick reference for Cofoja (Contracts for Java)*, 2011.
http://code.google.com/p/cofoja/wiki/QuickReference

[Mos94] C. Moss. *Prolog++, The Power of Object-Oriented and Logic Programming*. Addison-Wesley, Reading, Mass., 5th edition, 1994.

[MP91] Z. Manna and A. Pnuelli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1991.

[MPI94] Message Passing Interface Forum. A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, Tennessee, 1994.

[MQB07] M. Musuvathi, S. Qadeer, and T. Ball. *CHESS: A Systematic Testing Tool for Concurrent Software*, 2007.

[MS98] K. Marriot and P. J. Stuckey. *Programming with Constraints*. MIT Press, Cambridge, Massachusetts, 1998.

[MS00] B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming*, Erfurt, Germany, October 2000.

[MSH11] J. W. McCormick, F. Singhoff, and J. Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada*. Cambridge University Press, 2011.
http://www.cambridge.org/gb/knowledge/isbn/item5659578/

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[NA01] R. S. Nikhil and Arvind. *Implicit Parallel Programming in pH*. Morgan Kaufman, San Francisco, California, May 2001.

[Nel91] G. Nelson. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NY, 1991.

[Nil82] N. J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, Berlin, 1982.

[Nyek08] J. Nyéky-Gaizler (ed.) et al. *Java 2 útikalauz programozóknak: 5.0*. ELTE TTK Hallgatói Alapítvány, Budapest, 2008.

[Nyek98] J. Nyéky-Gaizler (ed.) et al. *Az Ada95 programozási nyelv*. ELTE Eötvös Kiadó, Budapest, 1998.

[OGS08] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[O'K90] R. A. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, Massachusetts, 1990.

[OSV11] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A comprehensive step-by-step guide*. Artima Inc., 2nd edition, 2011. ISBN 0981531644

[OT99] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. Ibm research report 21452, IBM T. J. Watson Research Center, April 1999. `http://www.research.ibm.com/hyperspace/Papers/tr21452.ps`

[Oxy13a] *Oxygene Programming Language*, 2013. `http://wiki.oxygenelanguage.com/en/Language`

[Oxy13b] *Oxygene – Class Contracts*, 2013. `http://wiki.oxygenelanguage.com/en/Class_Contracts`

[Oxy13c] *The Oxygene Language*, 2013. `http://www.remobjects.com/oxygene/`

[Pat71] S. Patil. Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes. *technical report*, 1971.

[PBRO11] A. Prokopec, P. Bawgell, T. Rompf, and M. Odersky. *On a Generic Parallel Collection Framework*, June 2011.

[PCM83] R. H. Perrott, D. Crookes, and P. Milligan. The programming language ACTUS. *Software Practice and Experience*, 13(4):305–322, Apr 1983.

[PD82] S. Pemberton and M. C. Daniels. *Pascal: The Language and Its Implementation - The P4 System*. Ellis Horwood Limited, Chichester, England, 1982.

[PE01] R. Plasmeijer and M. van Eekelen. Concurrent Clean Language. Technical report, University of Nijmegen, 2001.

[Per79] R. H. Perrott. A Language for Array and Vector Processors. *ACM TOPLAS*, 1(2):177–195, Oct 1979.

[Pet81] G. L. Peterson. Myths about the mutual exclusion problems. *Information Processing Letters*, 12:115–116, 1981.

[PH99] J. J. Peyton, J. Hughes, et al. *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language*, February 1999.

[Pil98] M. Pil. Dynamic Types and Type Dependent Functions, Implementation of Functional Languages, 10th International Workshop, IFL'98, London. *LNCS*, 1595:169–185, September 1998.
`http://www-i2.informatik.rwth-aachen.de/hutch/distributedHaskell`

[Pla99] R. Plasmeijer et al. *Functional Programming in Clean*. Draft, July 1999.
`http://www.cs.kun.nl/~clean/`

[Pos83] E. Post. Real Programmer's Don't Use Pascal, A letter to the editor. *Datamation*, 29(7), July 1983.
`http://www.pbm.com/~lindahl/real.programmers.html`

[PP08] B. Phillip and R. Paige. Cameo: An Alternative Model of Concurrency for Eiffel. *Formal Aspects of Computing*, 4(21):363–391, 2008.
doi:10.1007/s00165-008-0096-1

[PV92] B. Pârv and A. Vancea. Fundamentele limbajelor de programare. Egyetemi jegyzet, BBTE, Kolozsvár, 1992.

[PvE93] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, Reading, Mass., 1993.

[PVM02] OAK Ridge National Laboratory. *PVM Parallel Virtual Machine*, 2002.
`http://www.epm.ornl.gov/pvm/pvm_home.html`

[PW91] L. J. Pinson and R. S. Wiener. *Objective C*. Addison-Wesley, Reading, Mass., 1991.

[PZ01] T. W. Pratt and M. V. Zelkowitz. *Programming Languages: Design and Implementation*. Prentice Hall, Upper Saddle River, NY, 4th edition, 2001.

[Ric07] M. G. Ricken. A framework for testing concurrent programs. Technical report, Rice University, 2007.

[Rob65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *ACM*, 12:23–41, January 1965.

[Rog01] W. P. Rogers. *Reveal the magic behind subtype polymorphism*, 2001.

http://www.javaworld.com/javaworld/jw-04-2001/jw-0413-polymorph.html

[Sam69] J. Sammet. *Programming Languages: History and fundamentals*. Prentice Hall, Englewood Cliffs, NJ, 1969.

[SB04] P. Szeredi and T. Benkő. Deklaratív programozás: Bevezetés a logikai programozásba. Technical report, Budapesti Műszaki Egyetem, Budapest, 2004.
http://dp.iit.bme.hu/documents.html

[Sch02] H. Schildt. *C# The Complete Reference*. McGraw-Hill Osborne Media, New York, 2002.

[Sco09] M. K. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, Burlington, 3rd edition, 2009.

[Sea02] S. J. Searle. *A Brief History of Character Codes in North America, Europe, and East Asia*, 2002.
http://tronweb.super-nova.co.jp/characcodehist.html

[Seb13] R. W. Sebesta. *Concepts of Programming Languages*. Pearson, Boston, 10th edition, 2013.

[Sed77] R. Sedgewick. Permutation generation methods. *ACM Comput. Surv.*, 9(2):137–164, 1977.

[Set96] R. Sethi. *Programming Lanuages: Concepts and Constructs*. Addison-Wesley, Reading, Mass., 2nd edition, 1996.

[SH99] P. R. Serrarens, K. Hammond, et al. Explicit Message Passing for Concurrent Clean, Implementation of Functional Languages, 10th International Workshop, IFL'98. *LNCS*, 1595:229–245, September 1999.

[Sha02] A. U. Shankar. *Object-Oriented Exception Handling in Perl*, 2002.
http://www.perl.com/pub/a/2002/11/14/exception.html

[Sie05] J. Siek et al. *Concepts for C++0x*, 2005.
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1758.pdf

[Sim99] Ch. Simonyi. The Future is Intentional. *IEEE Computer Magazine*, 32(5):56–57, May 1999.

[SL95] A. Stepanov and M. Lee. The Standard Template Library. Technical report, Hewlett-Packard Laboratories, Palo Alto, CA, February 1995.

[Sma91] Inc. Digitalk. *Smalltalk/V Tutorial*, 1991.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, advances in formal methods edition, 2000.

[SOP03] *Subject-Oriented Programming*, 2003.
http://www.research.ibm.com/sop/

[SP01] R. L. Schwartz and T. Phoenix. *Learning Perl*. O'Reilly and Associates, Sebastopol, CA, 3rd edition, July 2001.

[Spi92] J. M. Spivey. The Z Notation: A Reference Manual. Technical report, Programming Research Group, University of Oxford, 1992.

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, Massachusetts, 2nd edition, 1994.

[ST96] L. Szili and J. Tóth. *Matematika és Mathematica*. ELTE Eötvös Kiadó, Budapest, 1996.

[Ste90] Jr. G. L. Steele. *Common Lisp: The Language*. Digital Press, Burlington, 2nd edition, 1990.

[Str94] B. Stroustrup. *The design and evolution of C++*. Addison-Wesley, Reading, Mass., 1994.

[Str00] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Mass., 3rd and special edition, 2000.
http://www.aw.com/catalog/academic/product/1,4096,0201700735,00.html

[Str12] B. Stroustrup. *Bjarne Stroustrup's C++ Style and Technique FAQ, Why doesn't C++ provide a finally construct?*, 2012.
http://www.stroustrup.com/bs_faq2.html#finally

[Suf08] B. Sufrin. Communicating Scala Objects. In Peter H. Welch et al. (eds.): *The thirty-first Communicating Process Architectures Conference*, volume 66 of *Concurrent Systems Engineering Series*, pages 35–54. IOS Press, 2008.

[Sun03] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.1*, 2003.
http://java.sun.com/products/ejb/docs.html

[sun03a] *The original Java homepage of Sun Microsystems*, 2003.
http://java.sun.com/

[Sut09] H. Sutter. *A Pragmatic Look at Exception Specifications*, 2009.
http://www.gotw.ca/publications/mill22.htm

[SV01] S. Sike and L. Varga. Objektum elvű modellalkotás UML-ben. Technical report, ELTE TTK, Informatikai Tanszékcsoport, Budapest, 2001.

[SWM04] B. Schoeller, T. Widmer, and B. Meyer. Making specifications complete through models. In Judith A. Stafford Ralf H. Reussner and Clemens A. Szyperski (eds.): *Architecting Systems with Trustworthy Components*, volume 3938, pages 48–70. SPRINGER LNCS, 2004.
http://se.inf.ethz.ch/old/people/schoeller/pdfs/schoeller_widmer_meyer_models.pdf

[Tan76] A. S. Tanenbaum. A Tutorial on Algol 68. *ACM Computing Surveys*, 8(2):155–190, June 1976.

[Tea09] The Go Team. Hey! Ho! Let's Go! Technical report, Google, 2009. `http://google-opensource.blogspot.hu/2009/11/hey-ho-lets-go.html`

[THH99] P. Tandi, K. Harmat, Z. Horváth, M. Wierich, and R. Plasmeijer. Web Computing in Clean. In *Proceedings of 4th International Conference on Applied Informatics*, pages 87–93, Eger, Hungary, 1999.

[THLP98] P. W. Trinder, K. Hammond, H. W. Loidl, and J. S. J. Peyton. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.

[Tho90] S. Thomson. Lawful Functions and Program Verification in Miranda. *Science of Computer Programming*, 13(2-3):181–218, May 1990.

[Tho99] S. Thompson. *Haskell, The Craft of Functional Programming*. Addison-Wesley, Reading, Mass., 1999.

[Til96] D. Till. *Teach Yourself in Perl 5 in 21 Days*. SAMS Publishing, Indianapolis, IN, 1996.

[TN06] A. Tucker and R. Noonan. *Programming Languages*. McGraw-Hill Science/Engineering/Math, 2006.

[Tur86] D. Turner. An Overview of Miranda. *ACM SIGPLAN Notices*, 21(12):158–166, 1986.

[Tur90] D. A. Turner. *Miranda System Manual*. Research Software Ltd., Canterbury, England, 1990.

[UCS02] *UCS (ISO 10646, Unicode) character blocks*, 2002. `http://www.cs.tut.fi/~jkorpela/ucs.htm8`

[Unr94] E. Unruh. Prime number computation, 1994. ANSI X3J16-94-0075/ISO WG21-462

[Ven00] B. Venners. *Inside the Java Virtual Machine, Chapter 20: Thread Synchronization*. McGraw-Hill Companies, 2nd edition, 2000. `http://www.artima.com/insidejvm/blurb.html`

[WA85] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, London, 1985.

[War80] D. H. D. Warren. Logic programming and compiler writing. *Software Practice and Experience*, 10(2), 1980. Article first published online: 27 OCT 2006.

[Wat06] D. A. Watt. *Programming Language Design Concepts*. Wiley, Chichester, 2006.

[Wel79] J. Welsh et al. Pascal Plus - Another Language for Modular Multiprogramming. *Software Practice and Experience*, 9:947–957, 1979.

[Whe96] D. A. Wheeler. *Ada 95 The Lovelace Tutorial*. Addison-Wesley, Reading, Mass., 1996.

[Wir73] N. Wirth. *Systematic Programming*. Prentice Hall, Englewood Cliffs, NJ, 1973.

[Wir74] N. Wirth. *Pascal - User Manual and Report*. Springer-Verlag, New York, 1974.

[Wir83] N. Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, 2nd edition, 1983.

[WP09] D. Wampler and A. Payne. *Programming Scala: Scalability = Functional Programming + Objects*. O'Reilly Media, 1st edition, 2009.
ISBN 0596155956

[Wul74] W. A. Wulf. Alphard: Toward a Language to Support Structured Programs. Technical report, Carnegie Mellon University, Pittsburgh, PA, April 1974.

[Zer12] *ZeroTurnaround Developer Productivity Report*, 2012.
`http://zeroturnaround.com`

[ZHH06] V. Zsók, Z. Hernyák, and Z. Horváth. Designing Distributed Computational Skeletons in D-Clean and D-Box. In *Central European Functional Programming School*, volume 4164 of *Lecture Notes in Computer Science*, pages 223–256. Springer Berlin Heidelberg, 2006.

[Zus72] K. Zuse. *Der Plankalkül*. Ges. für Mathematik und Datenverarbeitung, Bonn, 1972.

Index