## COLLABORATORS

<table>
<thead>
<tr>
<th>ACTION</th>
<th>NAME</th>
<th>DATE</th>
<th>SIGNATURE</th>
</tr>
</thead>
</table>

## REVISION HISTORY

<table>
<thead>
<tr>
<th>NUMBER</th>
<th>DATE</th>
<th>DESCRIPTION</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tartalomjegyzék

1. 1 INTRODUCTION 1
   1.1 1.1 Modeling 1
   1.2 1.2 Basic concepts 2
   1.3 1.3 Classification of programming languages 2
   1.4 Questions 3

2. 2 BASIC ELEMENTS 4
   2.1 2.1 Character set 4
   2.2 2.2 Lexical units 5
      2.2.1 2.2.1 Multi-character symbols 5
      2.2.2 2.2.2 Symbolic names 5
      2.2.3 2.2.3 Labels 6
      2.2.4 2.2.4 Comments 6
      2.2.5 2.2.5 Literals (Constants) 6
   2.3 2.3 General rules for the composition of source text 6
   2.4 Questions 7

3. 3 LITERALS IN LANGUAGES 8
   3.1 Pascal 8
   3.2 C 8
   3.3 Ada 9

4. 4 DATA TYPES 10
   4.1 4.1 Simple types 11
   4.2 4.2 Composite types 11
   4.3 4.3 Pointer type 13
   4.4 Questions 13

5. 5 NAMED CONSTANT AND VARIABLE 14
   5.1 5.1 Named constant 14
   5.2 5.2 Variable 14
   5.3 Questions 16
6. TYPES AND DECLARATIONS IN LANGUAGES
   6.1 Turbo Pascal ................................................................. 17
   6.2 Ada ............................................................................. 18
   6.3 C ................................................................................. 19

7. EXPRESSIONS
   7.1 Constant expressions .......................................................... 22
   7.2 Questions ..................................................................... 23

8. EXPRESSIONS IN C .................................................................. 24

9. STATEMENTS
   9.1 9.1 Assignment statements .................................................. 28
   9.2 9.2 The empty statement ..................................................... 28
   9.3 9.3 The GOTO statement ..................................................... 29
   9.4 9.4 Selection statements ..................................................... 29
       9.4.1 9.4.1 Conditional statements ....................................... 29
       9.4.2 9.4.2 Case/switch statement ....................................... 30
   9.5 9.5 Loop statements ......................................................... 32
       9.5.1 9.5.1 Conditional loops ............................................. 32
       9.5.2 9.5.2 Count-controlled loops ..................................... 32
       9.5.3 9.5.3 Enumeration-controlled loops ......................... 34
       9.5.4 9.5.4 Infinite loops .................................................. 34
       9.5.5 9.5.5 Composite loops .............................................. 34
   9.6 Questions ..................................................................... 34

10. EXAMPLES OF LOOP STATEMENTS IN LANGUAGES .............. 35
    10.1 FORTRAN .................................................................... 35
    10.2 PL/I ........................................................................... 35
    10.3 Pascal ......................................................................... 36
    10.4 Ada ............................................................................. 36
    10.5 C ................................................................................. 37
    10.6 Control flow statements in C ......................................... 38

11. THE STRUCTURE OF PROGRAMS ................................................. 39
    11.1 11.1 Subprograms ............................................................ 40
    11.2 11.2 The Call Chain and Recursion ................................ 42
    11.3 11.3 Secondary Entry Points ............................................ 42
    11.4 11.4 Block ................................................................... 43
    11.5 11.5 Compilation Unit ...................................................... 43
    11.6 Questions ..................................................................... 43
12. 12 PARAMETER EVALUATION AND PARAMETER PASSING 44
   12.1 12.1 Parameter Passing ......................................................... 45
   12.2 Questions ................................................................. 46

13. 13 SCOPE 47
   13.1 Questions ................................................................. 48

14. 14 EXAMPLES OF SPECIFIC LANGUAGE FEATURES 49
   14.1 FORTRAN ................................................................. 49
   14.2 PL/I ................................................................. 51
   14.3 Pascal ................................................................. 53
   14.4 Ada ................................................................. 54
   14.5 C ................................................................. 56

15. 15 ABSTRACT DATA TYPES AND THE PACKAGE 58
   15.1 Questions ................................................................. 62

16. 16 ON ADA COMPILATION 63
   16.1 16.1 Pragmas ................................................................. 63
   16.2 16.2 Compilation Units ................................................................. 63
   16.3 Questions ................................................................. 68

17. 17 EXCEPTION HANDLING 69
   17.1 17.1 Exception Handling in PL/I ................................................................. 70
   17.2 17.2 Exception Handling in Ada ................................................................. 72
   17.3 Questions ................................................................. 74

18. 18 GENERIC PROGRAMMING 75
   18.1 Questions ................................................................. 77

19. 19 PARALLEL PROGRAMMING AND THE TASK 78
   19.1 19.1 Ada Tasks ................................................................. 78
   19.2 Questions ................................................................. 84

20. 20 INPUT/OUTPUT 85
   20.1 20.1 I/O Features of Languages ................................................................. 86
   20.2 Questions ................................................................. 87

21. 21 MEMORY MANAGEMENT IN IMPERATIVE LANGUAGES 88
   21.1 Questions ................................................................. 89

22. 22 OBJECT-ORIENTED PARADIGM 90
   22.1 Questions ................................................................. 93
23. 23 JAVA

23.1 Java Basics ......................................................... 95
23.2 Types ............................................................... 95
23.3 Literals .............................................................. 96
23.4 Names ............................................................... 97
23.5 Block ................................................................. 97
23.6 Variables ........................................................... 97
23.7 Expressions ......................................................... 98
23.8 Statements ......................................................... 99
23.9 Packages ............................................................ 99
23.10 Classes ............................................................. 101
23.11 Fields ............................................................... 102
23.12 Methods ............................................................ 103
23.13 Instance initializer ............................................... 106
23.14 Static initializer .................................................. 106
23.15 Constructors ....................................................... 107
23.16 Instantiation ....................................................... 109
23.17 Interfaces ........................................................... 109
23.18 Exception handling .............................................. 111
23.19 Parallel programming ........................................... 112
23.20 Questions .......................................................... 113

24. 24 THE FUNCTIONAL PARADIGM .............................. 114

24.1 Questions ........................................................... 114

25. 25 THE LOGIC PARADIGM AND PROLOG .................... 115

25.1 Questions ........................................................... 120

26. BIBLIOGRAPHY ....................................................... 121
FOREWORD

The present book analyzes the features, concepts, philosophy, and computational models of high level programming languages. Specifically, it will focus on the particular elements of languages with a significant impact (FORTRAN, COBOL, PL/I, Pascal, Ada, C, Java, C#, Prolog). Note however that this book is not language description per se! It will introduce only certain parts of the languages, often in a simplified, incomplete form. The aim is to give an overview of programming language features at the model level, and to provide a general and coherent conceptual framework in which the concrete implementations of various languages can be placed. Knowledge of a specific language can be learned from books, electronic documentation, and tutorials. We give special attention to C, Ada and Java languages because of their practical importance.

You cannot learn programming in theory. You must write and execute lots and lots of programs!

To understand the subject of the book you need the following preliminary knowledge:
- abstract data structures;
- data representation;
- basic algorithms;
- basic concepts of operating systems.

Formal notation used in the book

The following notations will be used for the description of syntactic rules:

**Terminal**: written form; uppercase characters are used if the signs are letters.

**Non-terminal**: lowercase category name; names that consist of more than one word employ underscore characters as word separators.

**Alternative**: |  

**Option**: []

**Iteration**: . . . , it always means the optional repetition of the preceding syntactic item.

Syntactic rules can be formalized by the combinations of above mentioned items. The left side of each rule contains a non-terminal item, while the right side holds an arbitrary sequence of items. The two sides are separated by a colon. Terminals and non-terminals are set in **Courier New**; this font has also been used to highlight source code. Formal descriptor characters that are part of the given language will be set bold during the formalization.
1. fejezet

1 INTRODUCTION

1.1  1.1 Modeling

The human species has been anxious to learn the workings of the real world for a long time. The world that we conceive as real exhibits lots of kinds of objects (persons, animals, institutions, computer programs). These will be referred to as entities. On the one hand, entities have attributes characteristic of them; on the other hand, they form intricate relations with other entities. Entities react to the effects of the surrounding entities, enter into relations, and exchange information, i.e. entities have behavior. Specific entities can be distinguished from each other on the basis of their different attribute values and their different behavior. At the same time, real world entities can be categorized or classified by their common attributes and behavior.

The real world is too complex to be grasped in its entirety, which is why the human way of thinking is based on abstraction and high-level models. The essence of abstraction is to highlight the common, essential attributes and behavior, while ignoring those that are unimportant or different. The resultant model manages groups or classes instead of individual entities.

Our thinking relies on models whenever we communicate, teach, learn, face problems to solve or attempt to understand this writing.

The ability to create models is an innate capacity. A child getting acquainted with the world is in fact learning how to narrow down the diverse problems into a manageable number of problem classes.

In general, three requirements apply to models:

1. **Requirement of mapping**: There must be an entity to be modeled. This is the “original entity”.
2. **Requirement of narrowing**: Not all the features of the original entity appear in the model, just a select few.
3. **Requirement of feasibility**: The model must be feasible, i.e. conclusions drawn in the model have to be true when applied to the original entity.

Requirement 1 does not necessarily imply the actual existence of the original entity. The original entity can be fictitious (e.g. a character in a novel), hypothetic (e.g. a bacterium on Mars), or in the design phase (e.g. a machine to be produced).

Because of the second requirement, the model is always poorer, but at the same time more manageable as well (the original entity is not always manageable).

The reason we create models is formulated in Requirement 3. Since the original entity is often unavailable, research may be performed only on the model.

The appearance of computers has made it possible to automate certain elements of human thinking. Information technology has obtained essential importance in modeling. The attributes of entities can be managed via data, whereas the behavior of entities is managed by programs, which together result in a further model. So we can talk about data models and functional (procedural) models. This differentiation works only in computational environment, because the model itself is indivisible. From this perspective, data abstraction and procedural abstraction are another dimension of abstraction in informatics.
1.2. 1.2 Basic concepts

Three levels of computer programming languages may be distinguished:
– machine languages;
– assembly languages;
– high-level languages.
A program written in a high-level language is called a source program or source text. Rules that prescribe the structure and “grammar” of the source text are called syntactic rules. Rules of content, interpretation and meaning are called semantic rules. A high-level programming language is determined by its syntactic and semantic rules, i.e. its syntax and semantics.

Every processor has its own language and can execute only those programs that are written in that language. In order for the processor to understand the program written in the high-level language (i.e. the source text), some method of translation must be in place. There are two techniques to achieve this aim: (1) the compiler, and (2) the interpreter.

The compiler is a special program which creates an object program in machine code from the source program written in high-level language. The compiler treats the source program as a single unit, and executes the following steps:
– lexical analysis;
– syntactic analysis;
– semantic analysis;
– code generation.

During lexical analysis, the compiler segments the source text into lexical units (see Section 2.2). The aim of the syntactic analysis is to check whether syntactic rules are adhered to. Object programs can be derived from syntactically correct source texts only. The object program is already in a low-level machine language, but it cannot be run yet; in order to make the program work, the linkage editor has to create an executable program. The executable program is then placed into the memory by the loader and is given control. The running program is controlled by the run time system.

In the most general sense, compilers translate from any language to any other language. If a high-level language allows the source program to contain non-language elements, a precompiler (preprocessor) should be used first to generate a standard source program in the given language from the source text. This program may then be processed by the compiler of the language. C is such a language.

Compilers and interpreters share the first three steps, but differ in the fourth one: the interpreter does not create an object program. Instead, it takes the statements (or other language elements) of the source text one after the other, interprets the statement, and executes it. We get the results immediately by having a machine code routine run.

Programming languages may rely on compilers, interpreters, or both techniques.

Every programming language has its own standard which is called the reference language. The aim of the reference language is to define the precise syntactic and semantic rules which govern writing programs in that language. Syntax is usually given with the help of a specific formalism, while semantics is described for the most part in natural language (for example, in English). Several implementations may exist alongside the reference language (sometimes against it); these are compilers or interpreters adjusted to a given platform (processor and operating system). Sometimes more than one implementation is available even for the same platform, which may cause trouble as the implementations are neither compatible with each other nor with the reference language. The issue of program portability (if a program written in one implementation is transferred to another implementation, it runs there and provides the same results) has not been resolved in the course of the past 50 years.

Nowadays most programmers use Integrated Development Environments (IDE) with graphical user interfaces to write programs. Such environments contain a text editor, compiler (maybe interpreter), linkage editor, loader, run time system and debugger.

1.3. 1.3 Classification of programming languages

I. Imperative (algorithmic) languages

When the programmer writes a program text in these languages, he or she codes an algorithm, and this algorithm makes the processor work. The program is a sequence of statements. The most important programming feature is the variable, which
provides direct access to the memory, and makes it possible to directly manipulate the values stored within. The algorithm changes the values of variables, so the program takes effect on the memory. Imperative languages are closely connected to the von Neumann architecture.

Imperative languages fall into one of the following sub-groups:
- Procedural languages
- Object-oriented languages

II. Declarative (non-algorithmic) languages

These languages are not connected as closely to the von Neumann architecture as imperative languages. The programmer has to present only the problem, as the mode of the solution is included in language implementations. The programmer cannot perform memory operations, or just in a limited way.

Declarative languages fall into one of the following sub-groups:
- Functional (applicative) languages
- Logic languages

III. Other languages

This category comprises languages which do not fall into any of the above mentioned groups. These languages do not have much in common, apart from the fact that they generally deny one or more imperative features.

1.4. Questions

1. What is the model?
2. What are the requirements about the model?
3. How the compiler works?
4. How can programming languages be classified?
2. fejezet

2 BASIC ELEMENTS

This chapter is going to introduce the basic concepts and elements of programming languages.

2.1 2.1 Character set

Characters are the atomic building blocks of every program source code. The character set defines the basic elements that programs written in a given language may contain, and out of which more complex language elements can be composed. For imperative programs, these language elements are the following (in order of growing complexity):

- lexical units;
- syntactical units;
- statements;
- program units;
- compilation units;
- program.

Every language defines its own character set. Although there may be significant differences between the character sets, most programming languages categorize characters into the following groups:

- letters;
- digits;
- special characters.

All languages treat the 26 uppercase characters (from A to Z) of the English alphabet as letters. Many of the languages consider _ , $, #, @ characters as letters, too, although this is often implementation-dependent. Languages differ in their way of categorizing the lowercase characters of the English alphabet. Some languages (e.g. FORTRAN, PL/I) do not consider lowercase characters letters, while others (e.g. Ada, C, Pascal) do. This latter group of languages is further subdivided into classes that distinguish between capital letters and lowercase letters (e.g. C), and classes that treat them equal (e.g. Pascal). Most languages do not consider national characters as letters, except for a few recent languages. These languages allow the programmer to write for example “Hungarian” source code.

Regarding digits, programming languages are of a uniform opinion: the decimal numbers of the interval [0..9] are considered digits.

Special characters include mathematical operators (e.g. +, -, *, /), delimiter characters (e.g. [], {}, :, ', "), punctuation marks (e.g. ?, !), and other special characters (e.g. %, ~). Space is also treated as a special character (see Section 2.3).

The character sets of the reference language and the implementations may differ. Every implementation is equipped with a specific code table (EBCDIC, ASCII, UNICODE), which determines, on the one hand, whether it is possible to handle one byte or multi-byte characters; and determines, on the other hand, the order of the characters. Few reference languages define this order.
2.2. **2.2 Lexical units**

Lexical units are elements of the source text that have been recognized as such and tokenized (brought to an in-between form) by the compiler. Lexical units are of the following types:

- multi-character symbols;
- symbolic names;
- labels;
- comments;
- literals.

2.2.1. **2.2.1 Multi-character symbols**

Character sequences of more than one character whose meaning is predefined by the language such that they cannot be used in any other sense. Very often, these are operators and delimiters in the given language. For example, C defines the following multi-character symbols: `++`, `--`, `&&`, `/*`, `*/`.

2.2.2. **2.2.2 Symbolic names**

Symbolic names are identifiers, keywords, and standard identifiers.

*Identifier*: A character sequence that starts with letter, and continues with a letter or a digit. Programmers use identifiers to name and subsequently refer to their own programming constructs anywhere in the text of the program. Reference languages usually do not constraint the lengths of the identifiers, but for practical reasons implementations implicitly do so.

The following character sequences are regular identifiers in C (`_` is recognized as a letter):

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td>apple_tree</td>
</tr>
<tr>
<td>student_identifier</td>
</tr>
<tr>
<td>FirstName</td>
</tr>
</tbody>
</table>

Note that the following are not valid identifiers:

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x+y</td>
</tr>
<tr>
<td>the character ‘+’ is not allowed;</td>
</tr>
<tr>
<td>l23abc</td>
</tr>
</tbody>
</table>

Identifiers must start with a letter.

*Keyword (reserved word)*: A character sequence (usually with the restrictions of an identifier) whose meaning is defined by the language such that this meaning cannot be changed by the programmer. Not every language (e.g., FORTRAN, PL/I) acknowledges this construct. Statements usually start with a typical keyword, and are often referred to by that keyword in programmer jargon (e.g., “IF statement”). The keywords, which are often ordinary English words or abbreviations, characterize programming languages to a very large extent. Keywords cannot be used as identifiers.

The following are keywords in C:

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>if, for, case, break</td>
</tr>
</tbody>
</table>

*Standard identifier*: A character sequence whose meaning is defined by the language, which meaning however can be changed and reinterpreted by the programmer. Names of implementation constructs (e.g., built-in functions) are of this kind. Standard identifiers can be used as intended, or as one of the programmer’s own identifiers. For example, nil is one of C’s standard identifiers.
2.2.3. 2.2.3 Labels

Imperative languages use labels to mark executable statements, so that these statements can be referred to from another point in the program. All executable statements can be labeled.

Technically, a label is special character sequence, which can be either an unsigned integer number, or an identifier. Languages define labels in the following ways:

- COBOL: N/A
- FORTRAN: an unsigned integer number of no more than 5 digits.
- Pascal: In standard Pascal, a label is an unsigned integer number of at most 4 digits. Certain implementations allow identifiers as labels, too.
- PL/I, C, Ada: identifier

Labels are usually positioned before the statement, and are separated by colon. Ada also positions labels before the statement, but places them between the « and » multi-character symbols.

2.2.4. 2.2.4 Comments

A comment is a programming tool which allows programmers to insert character sequences into the program text that fall outside the scope of the compiler, and instead serve the interests of the reader of the program. Comments usually provide explanation on how to use the program, and give information about the circumstances of how it was written, what algorithms and solutions have been used. Comments are ignored by the compiler during lexical analysis. Comments may contain any of the characters included in the character set, all characters are considered equivalent, they represent themselves, and character categories are not important.

Note that there are three ways to place a comment in the source code:

• By placing complete comment lines in the source code (e.g. FORTRAN, COBOL). In this case, the first character of the line (e.g. C) indicates to the compiler that the line is not the part of the code proper.
• By placing the comment at the end of each line. In this case, the first part of a line contains the code to compile, while the second part contains characters that are to be ignored. In Ada, for example, comments last from the ‘--’ sign till the end of the line.
• By placing comments of arbitrary length wherever whitespace characters are allowed, but only if the language treats the space character as a terminating sign/delimiter (see Section 2.3). In this case, line endings are ignored; comments must start and end with special characters, or multi-character symbols. Examples of such comments are the ones placed between the { and } characters in Pascal, or between the /* and */ multi-character symbols in PL/I and C.

Well-written programs are rich in explanatory comments, which are indicators of good programming style.

2.2.5. 2.2.5 Literals (Constants)

A literal is a programming tool that allows programmers to include fixed, explicit values in the source code. Literals have two components: a type and a value. Literals are always self-defining. The written form of the literal (as a special character sequence) determines both the type and the value. Programming languages define their own literal sets.

2.3. 2.3 General rules for the composition of source text

Similarly to other kinds of text, the source code of every program is composed of lines. In this section we examine what roles lines play in programming languages.

Programming languages with fixed form: In the early programming languages (FORTRAN, COBOL), lines played a fundamental role. There was only one statement per line, and accordingly end of line characters also indicated the ends of statements. If a
statement did not fit into a single line, the programmer had to indicate that (in order to neutralize the effect of the line terminator). However, placing more than one statement in one line was not allowed. The order of the program elements within the line was also controlled. Programmers had to conform to strict rules.

*Programming languages with free-form:* These languages do not define any correspondence between the line and the statement. The programmer is allowed to write any number of statements per line, and one statement may occupy any number of lines. Program elements can appear at arbitrary locations within the lines. Ends of lines do not mark the end of statements. In order to help the compiler find where statements end, these languages introduce the statement terminator, which is generally the semicolon. In other terms, a statement stands between two semicolons in the source text.

Imperative languages demand that lexical units should be separated with a keyword, a special separator character (brackets, colon, semicolon, comma, etc.), or whitespace. With the help of these delimiters, the compiler is able to recognize the lexical units during the lexical analysis. Whitespace characters are universal delimiters in most (especially recent) languages. In comments, and string and character literals space plays an ordinary role, where it stands for itself. Wherever a space is allowed as a delimiter, any number of spaces may occur. Whitespaces are also allowed to occur at both sides of other delimiters, which improves the readability of the source code in general. FORTRAN allows programmers to put any number of spaces anywhere in the source code, because compilation starts with the elimination of spaces.

### 2.4. Questions

1. How can you categorize characters?
2. What is an identifier?
3. What is the keyword?
4. What kinds of symbolic names exist?
5. What is a label?
6. What is a comment used for?
7. What is a literal?
8. What is the special role of the “space”?
9. What are lexical elements?
3. fejezet

3 LITERALS IN LANGUAGES

3.1. Pascal

- Integer literal:

  \[\{+|-\} \text{ digit } [\text{ digit }...]\]

- Real literals:
  - Decimal fraction:
    \[
    \text{integer.unsigned_integer}
    \]
    Digits are required on both sides of the decimal point.
  - Exponential:
    \[
    \{\text{decimal_fraction } | \text{ integer}\}E\text{integer}
    \]

- String literal:

  \[
  ',\text{[character...]}'
  \]

3.2. C

- Short integer literals:
  - Decimal integer literal:
    The same as the integer literal in Pascal.
  - Octal integer literal:
    Must start with 0. For example: 011.
  - Hexadecimal integer literal:
    Starts with 0X. For example: 0X11

- Unsigned integer literal:

  \[
  \text{short_integer}\{U|u}\]

- Real literals:
- Long real (double-precision real):
  The same as the real literal in Pascal.

- Short real (single-precision real):
  ```plaintext
  long_real{f|F}
  ```

- Extended real (triple-precision real):
  ```plaintext
  long_real {l|L}
  ```

- Character literal:
  Represents the internal code of the given character, can also be used in calculations. Certain implementations allow more than one character to be put between the apostrophes.
  ```plaintext
  '`
  character
  ```

- String literal:
  ```plaintext
  "[character...]"
  ```

### 3.3. Ada

- Numeric literals:
  - Integer literal:
    ```plaintext
    digit[[_]digit...] [{E|e}[+]digit[digit...]
    ```

  - Real literal:
    The same as the real literal in Pascal.

  - Based literal:
    ```plaintext
    base#integer[.integer_without_sign]# [{E|e}[+|]digit[digit...]
    ```

    The base stands for the radix of 2-16 numeral system in decimal form. The optional exponent part contains digits in the decimal system, whereas digits between the # signs are the digits of the base numeral system.

- Character literal:
  ```plaintext
  '`
  character
  ```

- String literal:
  ```plaintext
  "[character...]"
  ```
4. fejezet

4 DATA TYPES

Data types are the manifestation of data abstraction in programming languages. A data type is an abstract programming feature, which always appears as the component of another concrete programming object. The name of a data type is an identifier.

Some of the programming languages recognize types, while others don’t; these languages may be referred to as typed vs. untyped languages, respectively. Imperative languages are generally typed languages.

The following three component determine a data type:
- domain;
- operations;
- representation.

The domain of a data type comprises those elements that concrete programming objects of the given type may take as values. With certain types, domain elements may appear as literals in the source code.

Data types define the operations that can be applied to the elements of their domain.

Every data type has an associated inner mode of representation which determines the way values of the given type are mapped onto memory. In other terms, the representation defines the number of bytes and possible bit-combinations that domain elements may be mapped onto.

Every typed language has built-in (standard) types.

Certain languages allow the programmer to define their own types, which implies a higher level of data abstraction. Custom data types allow the programmer to model the properties of real world entities more accurately.

Defining custom data types is more strongly related to the field of abstract data structures.

In order to create a custom data type, the programmer has to define the domain, operations and representation of the type. New custom types are often derived from built-in or previously defined custom types. The representation component is usually derived from that of another type, since only few languages allow custom representations (e.g. Ada). Languages further differ in whether the programmer is allowed to define custom operations and operators for new types. Certain languages allow both, while some make it is possible to define operations with the help of subprograms. The domain component may either be derived from other domains, or the elements may be given explicitly.

Data types are autonomous and differ from each other, except for one rather special case: when a new type (subtype) is derived from another type (base type) by restricting the latter’s domain, the operations and the representation remain invariant. As a result, the base type and the subtype are not different types.

Data types fall into two groups:

Scalar or simple data types: Their domains contain atomic values, all values are unique, and cannot be divided into smaller parts directly with language tools. Values taken from domains of scalar types can be appear as literals in the source code.

Structured or composite data types: The elements of these domains have their own type. The elements represent value groups, therefore they are not atomic. Value components may be accessed directly. Structured data types usually correspond to abstract data structures.
4.1. 4.1 Simple types

Every language includes at least one (signed) integer type, frequently more than one. Integer types have a fix-point representation, but differ in the number of bytes required to represent values of that type. Representation issues also determine the domain of the types. A few languages recognize the unsigned integer type whose representation is direct.

Real types are also fundamental; their representation is floating point. Their domain again depends on the chosen representation, which however varies with the implementations.

Integer types and real types are often referred to as numeric types. Numeric and relational operations can be executed on values of numeric types.

A few languages provide a complex type, usually implemented as a pair of real values that represent the real and imaginary Cartesian coordinates.

The domain of the character type comprises single characters, while the domain of the string type includes character sequences. The representation of these types is usually one or two bytes per character depending on the code table, their operations are the string and relational operations.

Some of the languages recognize the logical or Boolean type. The domain of the Boolean type consists of the true and false values; its operations are logical and relational operations. The representation is logical.

The enumeration type is a custom simple type that must be defined by the programmer by enumerating the elements of the domain. The elements are identifiers. Relational operations apply to these elements.

Another special class of simple types is the discrete or ordinal type, recognized by a few programming languages. Integer, character, logical and enumerated types are considered discrete or ordinal types by these languages. The domain elements of a discrete type are internally organized as a list (an abstract data structure), which means that there is a first and a last element, that every element is preceded by another element (except the first item), and that every element has a successor (except the last item). The order of the elements is therefore straightforward. The 0, 1, 2, … ordinal numbers are in bijective assignment with the domain elements except for integer types, where each domain element is assigned to itself.

The following operations always make sense with ordinal types:
- Querying the ordinal number of a given value in the domain, and vice versa.
- Querying the value of the preceding (predecessor) and the following (successor) elements.

The ordinal type can be conceived of as the generalization of the integer type.

The interval type is a subtype of the ordinal type.

4.2. 4.2 Composite types

The two most important composite types in procedural languages are the array (recognized by every language), and the record (unknown to FORTRAN only).

The array type is the programming language manifestation of the array abstract data structure. The array type is a static and homogeneous composite type. Its domain elements are value groups that have the same number of elements which are all of the same type.

The array as a type is determined by:
- its number of dimensions;
- the type and domain of its index set;
- the type of its elements.

There are languages (e.g. C) that do not recognize multidimensional arrays, and instead manage them as one-dimensional arrays of one-dimensional arrays.

Multidimensional arrays can be represented in either a row-major or a column-major order. Although the representation is widely implementation-dependent, the row-major order is the more frequent approach.
The name of a programming object with array type can be used to refer to all the elements as one value group (the sequence of the elements is determined by the representation). The elements of the value group can be accessed with indices written after the name of the programming object. Some of the languages enclose indices in round brackets, other languages prescribe the use of square brackets. Certain languages (e.g. COBOL, PL/I) allow the programmer to access all the elements of a given dimension of an array in a single operation (e.g. to access one row of a two-dimensional array).

Languages must answer the following questions about the array type:

1. *Which element types does the array support?*
   - Every language allows every scalar type.
   - Modern languages allow composite types, too.

2. *Which index types does the array support?*
   - Every language allows the integer type.
   - Enumerations are also allowed in Pascal and Ada.

3. *When defining an array type, how can the index domain be declared?*
   - With an interval type value (Pascal, Ada), i.e. by providing the lower and upper bounds.
   - Other languages (e.g. PL/I) have fixed lower bounds (in most cases numbering starts with 1), and the programmer is required to provide only the upper boundary of the domain.
   - A small group of languages require that the programmer provides only the lower boundary; upper boundaries are fixed by the language.
   - In rare cases (e.g. in C) it is the number of elements at a given dimension that the programmer has to declare; language derives the index domain.

4. *How can the programmer declare the upper and lower index boundary, or the number of elements?*
   - With literals or named constants (e.g. in FORTRAN, COBOL, Pascal), or with constant expressions (for example in C). Such languages rely on static array boundaries, which implies that the number of value group elements is determined at compile time.
   - With expressions (e.g. in PL/I, Ada). These languages support dynamic array boundaries, i.e. the number of value group elements is determined during run time. However, once the element numbers are determined, they do not change.

The string type is often implemented as an array of characters, and is often supported by special purpose operations not available for other arrays.

The array type also plays an important role in implementations realizing the array representation of abstract data structures.

The record type is the programming language manifestation of the abstract data structure that bears the same name. The record type is always heterogeneous. Its domain elements are value groups which—as opposed to arrays—may be of different types. Value group elements are called fields. Every field has a unique name (which is an identifier), and a type. Fields of different record types may have the same name.

Certain languages consider the record type static (e.g. C), which means that the number of fields is the same in each value group. In other languages (e.g. Ada), there is a common field set in every value group (called the fixed part of the record), and another field set whose fields vary across value groups (that field set is the variable part of the record). A special language feature, the discriminator determines which variable fields should be present in the record and under what circumstances.

Former languages (PL/I, COBOL) preferred the multilevel record type where fields may be divided into further fields at any depth. Only the lowest-level fields have types, and these types must be simply ones. More recent languages (like Pascal, C, Ada) manage one-level record types; one-level record types do not recognize subfields, but field types can be composite.

The name of a programming object with record type can be used to refer to all value group fields (in order of declaration) in one operation.

Individual fields are accessed through a qualified name of the following form:
It is compulsory to qualify the field with the name of object, because names of fields are not necessarily unique. The record type plays an important role in input-output.

4.3. 4.3 Pointer type

The pointer type is a simple type whose domain elements are memory addresses. Pointer types give the programmer access to the memory through indirect addressing. The value of a programming object with pointer type is a memory address, which is why we say that this object addresses the given region, i.e. “points” to the given memory region. One of the most important pointer type operations is accessing the value located at the addressed memory region.

The pointer type plays an all-important role in abstract data structure implementations.

In some of the languages, pointers are restricted to point only to objects in the dynamic area (heap). The only way to create a new pointer value is to call a built-in function that allocates a new object in the heap and returns a pointer to it. In other languages, one can create a pointer to a non-heap object by using an “address of” operator. The question that both methods have to cope with is how and when the storage space is reclaimed for objects that are no longer needed.

A few languages require that the programmer should reclaim space explicitly. Other languages require that the implementation reclaim the unused object automatically. Explicit memory reclamation raises the possibility that the programmer will forget to reclaim the objects that are no longer alive (which will cause memory leaks), or will reclaim objects that are still in use (thereby creating dangling references). Automatic memory reclamation (known as garbage collection) raises the question of how the implementations distinguish garbage from active objects.

4.4. Questions

1. What is a data type?
2. How can we classify data types?
3. Characterize simple types.
4. What are the simple types in languages?
5. Characterize the composite types.
6. Characterize the array type.
7. Characterize the record type.
8. Explain the importance of the pointer type.
9. What is the dangling reference?
5. fejezet

5 NAMED CONSTANT AND VARIABLE

5.1. 5.1 Named constant

A named constant has 3 components:
- name;
- type;
- value.

Named constants must be always declared.

A named constant is represented in the source code by its name. The name always stands for the value component. The value component cannot be changed during run-time, it is determined at declaration.

The role of the named constant is to allow the programmer to give descriptive names to recurring, frequent values. The name of a named constant is therefore indicative of the value’s role. A further advantage of using named constants is that should the given value be modified in the source code, it is sufficient to modify the declaration statement, instead of looking up all the occurrences of the value.

Languages should answer the following questions:
1. Do predefined named constants exist in the language?
2. Is it possible for the programmer to define their own named constants?
3. If it is, of what type?
4. How can you give value to the named constant?

There are no named constants in FORTRAN and PL/I. There are predefined named constants in COBOL. There are predefined named constants in C, and the programmer may also create custom named constants in various ways. The most simple way is using the

#define name literal

macro. In this case, the precompiler substitutes all the occurrences of the name with the literal in the source program.

There are predefined named constants in Pascal and Ada. The programmer may also define custom named constants of both simple and composite types. In Pascal, the value is expressed with literals; in Ada, with the help of expressions.

5.2. 5.2 Variable

A variable has 4 components:
- name;
- attributes;
- address;
- value.

The name is an identifier. Variables are always represented by their names in the source code, which may refer to any of the four components.

Attributes are characteristic features that determine the behavior of the variable during run time. Imperative languages consider the type as the most fundamental attribute, which delimits the values that the variable can take. Attributes are assigned to variables through declaration. Declarations are of many different kinds.

Explicit declaration: The programmer writes an explicit declaration statement which assigns attributes to the variable’s name. Languages usually allow the same attributes to be assigned to different variable names.

Implicit declaration: The programmer assigns attributes to letters in a special declaration statement. If a variable is not listed in any of the explicit declaration statements, it will have the attributes assigned to the first letter of the variable’s name; i.e. variables beginning with the same letter will have the same attributes.

Automatic declaration: The compiler assigns attributes to variables. These variables are not declared in any explicit way, nor are they assigned attributes based on their first letter in an implicit declaration statement. The compiler assigns the attributes to the variable by virtue of arbitrary characters (usually the first character) of name.

All imperative languages support explicit declaration, with some of them recognizing only this kind of declaration. More recent languages demand that every named programming object must be declared explicitly.

The address component of the variable defines the part of the memory where the value of variable is placed. The run-time period during which the variable has an address component is called the lifetime of the variable.

A variable’s address is determined in one of the following ways:

Static memory allocation: The address of the variable is determined before run-time, and it remains unchanged during the execution. When the program is loaded into memory, variables with static allocation occupy their fixed place in memory.

Dynamic memory allocation: The address is assigned by the run time system. The variable gets its address component when the program unit that defines it as a local variable is activated; the address component is taken away once the given program unit stops working. The address component is subject to change during run time; there may be time periods when the variable has no address component at all.

Programmable memory allocation: The address component is assigned to the variable by the programmer at run time. The address component may change, and the variable may also exist without an address component in certain time intervals. There are three basic ways of programming memory allocation:
- The programmer assigns an absolute address to the variable and specifies its exact place.
- The programmer assigns the address of the variable relative to the address of another programming object in the memory (relative address). It is possible that the programmer does not know the absolute address of the variable.
- The programmer defines only that moment in time from which on the given variable has an address component. Allocation is catered for by run time system. The programmer does not know the absolute address.

In all of the above described cases the programmer must be provided with features to help take away the address component.

Programming languages exhibit a great variety of address assignment techniques. Dynamic address assignment is typical of imperative languages.

Multiple memory reference is the phenomenon when two variables with different names (and potentially different attributes) share the same address component, and as a result the same value component in a given moment during run time. This entails that if the programmer changes the value of one of the variables, the other variable changes too. Early languages (e.g. FORTRAN, PL/I) provided explicit language features to manage multiple references, since certain problems were solved only in this way. Similar situations may arise in more recent languages as well—sometimes accidentally—, which leads to unsafe code.

The value component of the variable is the bit combination that occupies the memory address defined by the variable’s address component. The representation prescribed by the value type determines the structure of the bit combination.

The value component of a variable can be determined in the following ways.
Assignment statement: This is by far the most frequent statement in imperative languages, and is also fundamental in coding algorithms. The following are examples of the assignment statement in various languages:

FORTRAN:

```
variable_name = expression
```

COBOL:

```
value TO variable_name [, variable_name ...]
```

PL/I:

```
variable_name [,variable_name ...] = expression;
```

Pascal:

```
variable_name := expression
```

C, Java, C#:

```
variable_name = expression;
```

Ada:

```
variable_name := expression;
```

On the left side of the assignment statement, the name of the variable generally stands for the address component. As opposed to this, the name of the variable stands for the value component in expressions. The order of executing the operations within an assignment statement is implementation-dependent. In most cases, the address component of the left side variable is evaluated first.

In languages with type equivalence, the type of an expression must be the same as the type of the variable, whereas in languages with type compatibility, the type of the expression is always converted to the type of the variable.

Input: The value component of the variable is determined by a piece of data received from a peripheral device (see Chapter 20).

Initialization: There are two kinds of initialization. In explicit initialization, the programmer has to declare the value component of the variable in the explicit declaration statement. Once the variable obtains its address component, the bit sequence which represents the value is also set. The value component can be expressed with the help of a literal or an expression.

Certain reference languages claim that the value component of the variable is undefined until the programmer sets its value. Undefined values should not be used because the address component of the variable may contain arbitrary bit combinations (“junk”), which is useless. Other reference languages support automatic initialization, which means that if the programmer does not initialize explicitly the value of the variable, the address component is set to a predefined bit combination determined by the reference language (variables are “set to zero”). A third class of reference languages does not claim anything about the initialization of variables. However, most implementations do accomplish automatic initialization, sometimes even against the reference language.

### 5.3. Questions

1. Describe the named constant.
2. What are the components of a variable?
3. What do you know about using variables in a program?
4. How can you assign an attribute to a variable?
5. How can a variable get an address component?
6. How can a variable get a value component?
6. fejezet

6 TYPES AND DECLARATIONS IN LANGUAGES

6.1. Turbo Pascal

The programmer must declare every programming object explicitly in Pascal. Turbo Pascal has the following type system:

- simple types
  - ordinal
    - predefined
      - character (char)
      - logical (boolean)
      - integer (integer, shortint, longint, byte, word)
    - enumeration
    - interval
  - real
- string (can be conceived of as a simple and complex type at the same time, i.e. a character sequence, or as a one-dimensional array of characters)
- composite types
  - array (array)
  - record (record)
  - set (set)
  - file (file)
  - object (object)
- pointer (pointer)

Declaring an array type:

```
ARRAY [ interval [, interval ...] ] OF type
```

The declaration of a named constant:

```
CONST name-literal; [ name-literal; ...]
```
The declaration of a variable:

\[
\text{VAR name:type; [ name:type; ...]}
\]

Creating a custom type:

\[
\text{TYPE name-type; [ name-type; ...]}
\]

Custom types declared in this manner will be different from every other type.

### 6.2. Ada

The programmer must declare every programming construct explicitly.

Ada has the following type system:

- **scalar types**
  - enumeration
    - integer (integer)
    - character (character)
    - logical (boolean)
    - real (float)
- **composite types**
  - array (array)
  - record (record)
- **pointer (access)**
- **private (private)**

The scalar type is an enumeration type.

The interval subtype must be declared as follows:

\[
\text{RANGE lower_bound..upper_bound}
\]

Explicit declaration statements must be of the following form:

\[
\text{name [, name ...] : [CONSTANT] type [:=expression];}
\]

The keyword **CONSTANT** indicates that a named constant is being declared; otherwise this is a variable declaration. The expression element is compulsory in named constant declarations; it defines the value of the constant. In variable declarations, the expression is used for setting the initial value of the variable.

Example:

\[
\begin{align*}
A & \text{: constant integer:=111;} \\
B & \text{: constant integer:=A*22+56;} \\
X & \text{: real;} \\
Y & \text{: real:=1.0;}
\end{align*}
\]

Unique custom types are declared as follows:
TYPE name IS type;

Declaring a subtype:
SUBTYPE name IS base_type;

Example:
type BYTE is range 0..255;
subtype LOWERCASE is character range ''a'..'z;

Creating a custom enumeration type:
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
subtype WEEKDAY is RANGE MON..FRI;

Array boundaries are dynamic in Ada. It is possible to define a custom array type without setting the index domain beforehand; in this case, the index domain is set only when the type occurs in a declaration statement.

Example:
type T is array(integer<> , integer<>);
A:T(0..10,0..10);

6.3. C

C has the following type system:

- arithmetic types
  - integral types
    * integer (int, short[int], long[int])
    * character (char)
    * enumeration
  - real (float, double, long double)
- derived types
  - array
  - function
  - pointer
  - structure
  - union
- void type

Arithmetic types are simple types, while derived types are composite types in C. Arithmetic operations can be performed on the elements of an arithmetic type’s domain. There is no logical type in C: true is expressed as `int 1`, false corresponds to `int 0`. The term `unsigned` used before an integer or a character type indicates direct representation, while `signed` is used to mark signed representation. Records have a fixed structure. A union is a record that contains only a variable part, which part comprises exactly one field on every occasion. The domain of the `void` type is empty, which is why it has no representation or operations.
Enumeration type domains are not allowed to overlap. Domain elements may be considered as named constants of `int` type. Element values can be set explicitly with integer literals; if no explicit assignment is used, the values start from 0 and increase by 1 in accordance with their relative position within the enumeration. If the value of an element is explicitly set, but the value of the subsequent element is not, the latter one will be larger by one than the previous item’s value. It is possible to assign the same value to different elements. The enumeration type is declared as follows:

```
ENUM name {identifier [=constant_expression] [, identifier [=constant_expression] ...]};
```

Example:

```
enum color {RED=11, PINK=9, YELLOW=7, GREEN=5, BLUE=3, MAGENTA=3};
```

Explicit declarations are of the following form:

```
[CONST] type_spec object_identifier [=expression] [, type_spec object_identifier [=expression...]];
```

The keyword `CONST` indicates that a named constant is being declared (in this case the `expression` sets the value of the constant, `type_spec` defines the type, and `object_identifier` must be a valid identifier); otherwise the statement declares a programming object of type `type_spec` and with the name `object_identifier`. Variables may be assigned explicit initial values with the help of the `expression` element. In the latter case, the `object_identifier` may be substituted by one of the following items:

- `identifier`: variable of type `type_spec`;
- `(identifier)`: pointer type variable pointing to a function with `type_spec` return type;
- `*identifier`: pointer type variable pointing to an object with `type_spec`;
- `identifier()`: function with `type_spec` return type;
- `identifier[]`: variable of the type array that contains elements of `type_spec`;
- and any combination of these. The `type_spec` may include the same constructions next to the type name.

Example:

```
int i, *j, f(), *g(), a[17], *b[8];
```

In this declaration, `i` is an integer type variable; `j` is variable of a pointer type pointing to an integer; `a` is a variable of one-dimensional array type containing integers, it has 17 elements; `b` is a variable of one-dimensional array type containing pointer type elements pointing at integers, it has 8 elements; `f` is a function with integer return type; `g` is a function with pointer return type.

Declaring a custom type:

```
TYPEDEF type_spec name [,type_spec name...];
```

This statement does not create a true new type, it only defines a synonym for the `type_spec`.

Declaring a structure:

```
STRUCT [structuretype_name] {field_declarations} [variable_list]
```

Declaring a union:

```
UNION [uniontype_name] {field_declarations} [variable_list]
```

C supports one-dimensional arrays. The number of indexes must be stated in the declaration statement; the range of indices is between 0..number-1. The reference language recognizes static array boundaries, but certain implementations manage the boundaries dynamically.

Arrays are mapped onto the pointer type in C. The name of an array type variable is in fact a pointer type variable that points to the elements of the array.

C supports automatic declaration. If the programmer does not define the type of the name of a function, it will be a type `int` construct by default.
7. fejezet

7 EXPRESSIONS

Expressions are syntactic features which are used to derive a new value from other values known at a given point in the program. An expression has two components: a value and a type.

Formally, expressions consist of the following components:

- **operands**: An operand may stand for a literal, a named constant, a variable or a function call.
- **operators**: Define operations executed on values.
- **round brackets**: They affect the order in which the operations are executed. Languages allow the redundant use of brackets.

The simplest form of expression consists of a single operand.

Depending on the number of operands required to perform an operation we distinguish between **unary**, **binary** and **ternary** operators.

Expressions are of three types based on the relative order of the operator and the operands if the operators have two operands. The three possible forms are:

- **prefix**: the operator precedes the operands (\( * 3 5 \))
- **infix**: the operator stands between the operands (\( 3 * 5 \))
- **postfix**: the operator follows the operands (\( 3 5 * \))

Operators with one operand generally precede the operand, occasionally follow it. Operators with three operands are generally infix.

The process that determines the value and the type of an expression is the **expression evaluation**. When evaluating an expression, the operations are executed, values are calculated and the type is determined.

Operations may be executed in one of the following ways:

- In the order of writing, i.e. from left to right.
- In reverse order of writing, i.e. from right to left.
- From left to right in accordance with the precedence table.

The infix form is ambiguous. Infix operators are usually not of the same strength; therefore, languages which support the infix form define the order of operators in a **precedence table**. The precedence table consists of lines where operators in the same line are of equal precedence. The relative strength of the operators decreases from top to bottom. Each line defines the **binding direction**, which determines the order of evaluating neighboring operators in the given line. The binding direction is left to right or right to left.

The evaluation of an infix expression takes place as described below.

The evaluation starts at the beginning of the expression (left-to-right rule). If there is only one operand, its value determines the value of the expression, and its type determines the type of the expression. If there is only one operator, we execute the appropriate expression. Otherwise, we compare the precedence of the first and the second operator. If operator 1 is stronger...
than operator 2, or they are of the same strength, and the relevant line of the precedence table prescribes a left-to-right binding
direction, the left operator is executed first. Otherwise we proceed towards the next two operators (provided we have not yet
reached the end of the expression) and compare their precedence. This approach makes it easy to determine the first operation,
but the evaluation order of the rest of the expression is implementation-dependent. Once the first operator is determined and
executed, the evaluation may continue at the beginning of the expression, or may proceed until the end of the expression, when
the evaluation returns to and continues with the beginning of the expression.

Comment: Expressions are evaluated at run time. It is the compiler that creates unambiguous postfix expressions from infix
expressions, which implies that the previous steps apply to the rewriting of infix expressions, and not their evaluation as such.

Before executing an operation the value of the operands must be determined. Most reference languages prescribe the (usually
left-to-right) order of operand evaluation. Other reference languages do not take sides. The order of operand evaluation is
implementation dependent in C, for example.

Infix expressions may employ round brackets in order to override the execution order as determined by the precedence table.
Expressions enclosed in brackets are always evaluated first. Certain languages indicate round brackets in the first line of the
precedence table.

A completely bracketed infix expression is unambiguous, there is only one possible order of evaluation.

Imperative languages prefer the infix form.

Expressions containing logical operators are special from the perspective of evaluation, because sometimes the final value of
the whole expression is obvious before all the operations are executed. For example, if the first operand of an AND operation
evaluates to false, the result will also be false irrelevant of the value of the second operand (no matter how complex the second
part of the expression may be).

The following are examples of how specific languages treat the evaluation of expressions that contain logical operators:
- Logical expressions must be completely evaluated; this is the complete evaluation (e.g. FORTRAN).
- The evaluation of the expression lasts only until the result is unambiguously revealed. This is the short-circuit evaluation (e.g.
  PL/I).
- There are short-circuit and non-short circuit operators in the language. The programmer may decide the manner of evaluation
  (e.g. Ada’s non-short circuit operators are: and, or; short-circuit operators: and then, or else).
- The manner of evaluation is set as a run time parameter (e.g. Turbo Pascal).

The manner of evaluating the type of an expression is determined by whether the programming language supports type equiva-
lence or type compatibility. This issue is also relevant for assignment statements (see Section 5.2) and parameter evaluation (see
Chapter 12).

Languages with type equivalence say that a binary operator may only have operands with of the same type. There is no conver-
sion, the type of the result is either the shared type of the two operands, or depends on the operator. For example, in the case of
relational operations the result will be an instance of the logical type.

Languages may consider the type of two programming objects identical in the following cases:
- declaration equivalence: the objects have been declared in the same declaration statement, with the same type name.
- name equivalence: the objects have been declared with the same type name (not necessarily in the same declaration statement).
- structural equivalence: the two objects are of a composite type, and the structure of their types is identical (e.g. two array types
  containing 10-10 integers each, irrespective of the index domains).

Languages that support type compatibility allow binary operators to have operands that are of different types. However, the
operations may be executed only if the two operands have identical inner representation, in which case there is conversion
between the operands. In this case, the language defines, on one hand, the valid type combinations, and on the other hand,
the type of the result of the operation. When evaluating an expression, the type of the sub-expressions is calculated after each
operation is executed; the type of the whole expression is calculated after the last operation has been performed.

Ada strictly prohibits all forms of type mismatch. PL/I supports total conversion.

7.1. Constant expressions

An expression which is evaluated by the compiler and whose value is therefore determined at compile time is called a constant
expression. Its operands are literals and named constants.
7.2. Questions

1. How can expressions building up?
2. What is the precedence table?
3. What is the expression evaluation?
4. Short-circuit evaluation.
5. What is the type equivalence?
6. What is the type compatibility?
7. What is the constant expression?
8. fejezet

8 EXPRESSIONS IN C

C is an expression-oriented language, and supports conversion between arithmetic types.

The domain elements of the pointer type may be operands of addition and subtraction operators, in which case they behave as unsigned integers.

The name of a variable with an array type is pointer type such that the expression \( a[i] \) is the same as \( *(a+i) \) if \( a \) and \( i \) are declared as \( \text{int } i; \text{ int } a[10]; \).

Expressions in C have the following recursive definition:

```plaintext
expression:

{ primary_expression |

lvalue++ |

lvalue-- |

++lvalue |

--lvalue |

unary_operator expression |

SIZEOF(expression) |

SIZEOF(type_name) |

(type_name)expression |

equation binary_operator equation |

expression?expression:expression |

lvalue assignment_operator expression |

equation, equation } |

primary_expression:
```
{ literal |
variable |
(expression) |
function_name(actual_parameter_list) |
array_name[expression] |
lvalue.identifier |
primary_expression -> identifier |

lvalue:
{ identifier |
array_name[expression] |
lvalue.identifier |
primary_expression -> identifier |
*expression |
(lvalue)}

The precedence table of C:

```
<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( ) [ ] .  -&gt;</td>
</tr>
<tr>
<td>2</td>
<td>* &amp; + - ! = == ++ -- SIZEOF (type) ←</td>
</tr>
<tr>
<td>3</td>
<td>/ %           →</td>
</tr>
<tr>
<td>4</td>
<td>+ -           →</td>
</tr>
<tr>
<td>5</td>
<td>&gt;&gt; &lt;&lt;         →</td>
</tr>
<tr>
<td>6</td>
<td>&gt;= &lt;=         →</td>
</tr>
<tr>
<td>7</td>
<td>== !=         →</td>
</tr>
<tr>
<td>8</td>
<td>&amp;             →</td>
</tr>
<tr>
<td>9</td>
<td>^             →</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>&amp; &amp;           →</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>?:            →</td>
</tr>
<tr>
<td>14</td>
<td>+= -= *= /= %= &gt;&gt;= &lt;&lt;= &amp;= ˆ=</td>
</tr>
<tr>
<td>15</td>
<td>,             →</td>
</tr>
</tbody>
</table>
```

The last column indicates the binding direction.

Formal descriptions of expressions in C may rely on the following shorthand operator names:

- **unary_operator**: the first 6 operators in the 2. line of the precedence table
- **binary_operator**: operators in lines 3 to 12 of the precedence table
- **assignment_operator**: operators in line 14.

The meaning of each operator in C:

()  
This operator serves two distinct purposes. First, it helps the programmer override the precedence of operators; second, it is the function operator.

[]  
The array operator.

.  
The qualifier operator used in structures and unions qualified by name.

->  
The operator of qualifying with a pointer.

*  
Indirection operator; provides access to the value at the memory address referenced by its pointer type operand.

&  
Returns the address of the operand.

+  
Plus sign.

-  
Minus sign.

!  
Logical NOT operator available for integral and pointer type operands. If the value of the operand is not zero, the result will be zero; otherwise returns 1. The result is of type int.

~  
The ones’ complement operator.

++ and --  
The increment and decrement operators (post and pre). Increase or decrease the value of their operand by 1, respectively.

**Example 1:**

```c
int x,n;
n=5;
x=n++;
```

**Example 2:**

```c
x=++n;
```

In Example 1, \( x \) evaluates to 5, because the assignment operator is applied on the former value of \( n \), i.e. the one before the post increment operator is executed.

In Example 2, \( x \) evaluates to 6, because assignment takes place after the value of \( n \) has been increased.

Note that the value of \( n \) increases by 1 in both cases.

`sizeof(expression)`
The size of the expression’s type in bytes.

\texttt{sizeof(type)}

The size of a data type in bytes.

\texttt{(type)}

Casting operator.

\star

The operator of multiplication.

\slash

The operator of division; integer division if the operands are of integer type.

\%

Modulo operator. The modulo is the remainder of an integer division.

\ast

The addition operator.

\minus

The subtraction operator.

\gg and \ll

Shift operators. Shifts the left operand to the right (or to the left) by the number of bits determined by the right operand. The left shift operator introduces zeros from the right side, while the right shift operator shifts the sign bit along the left side. Works with integral type operands.

\lt, \gt, \leq, \geq, \lt\lt, \ll\ll

Relational operators. The result is \texttt{int 1} if the expression evaluates to \texttt{true}, \texttt{int 0} otherwise.

\&\&, \^\^, \|

Non-short circuit logical operators (AND, exclusive OR, OR). Work with integral types and perform bit comparisons.

\&& and ||

Short circuit logical operators (AND, OR). Work with \texttt{int 0} and \texttt{1} values.

\? : 

The only ternary operator in C, also called the conditional operator. If the value of the first operand is not \texttt{0}, the result of the operation is determined by the value of the second operand, otherwise it is determined by the third operand.

For example, the expression \texttt{(a>b)?a:b} selects the greater value from \texttt{a} and \texttt{b}.

\!, \plus, \minus, \ast, \slash, \ast\ast, \ast\ast\ast, \ll\ll, \ll\ll\ll\ll, \&\&\&, \^\^\^\^, \|\|

Assignment operators. Expressions of the form \texttt{x operator= y} are shorthand for \texttt{x=(x)operator(y)}.

The operator overwrites the value of the first operand.

\# 

Series operator enforces a left-to-right order of evaluation.
9. fejezet

9 STATEMENTS

Statements are imperative tools which on the one hand help formalize the steps of an algorithm, and on the other hand are used by the compiler to generate the object program. The two major groups of statements are the declaration and executable statements.

Declaration statements do not translate into object code. The vast majority of declaration statements address the compiler in order to ask for a service, set a mode of operation, or supply information which is used by the compiler in generating the object code. These statements influence the object code fundamentally, but the statements themselves are not compiled. Declaration statements allow the programmer to introduce their own named programming objects. Object code is generated from executable statements by the compiler. Normally, a high-level executable statement is translated into more than one (sometimes a surprisingly large number of) machine code statement(s).

Every executable statement falls into one of the following categories:

1. Assignment statements
2. The empty statement
3. The GOTO statement
4. Selection statements
5. Loop statements
6. Call statements
7. Control statements
8. I/O statements
9. Other statements

Statements 3 to 7 are called control flow statements. Most procedural languages support the first five statements, a few recognize statements 6 to 8 as well. The most marked difference between the languages is whether the language permits other statements (group 9). Some languages do not contain such statements (e.g. C), while others abound in such language constructs (e.g. PL/I).

9.1. 9.1 Assignment statements

Its role is to set the value component of one (or possibly more) variable at any point in the program. This statement has already been discussed in Section 5.2.

9.2. 9.2 The empty statement

Most imperative programming languages recognize the empty statement. (The syntax of early languages made it almost impossible to avoid the empty statement.) The greatest advantage of empty statements is that they contribute to writing clear and unambiguous programs.
The empty statement makes the processor execute an empty machine instruction.
The empty statement is indicated by a separate keyword in certain languages (e.g. CONTINUE in FORTRAN, NULL in Ada). Other languages do not mark the empty statement (e.g. there is nothing between two statement terminators).

9.3. 9.3 The GOTO statement

The GOTO statement is used in order to transfer control from one point in the program to a labeled executable statement.

The most common form of the GOTO statement:

GOTO label

In early languages (FORTRAN, PL/I), it was impossible to write a program without the GOTO statement. Later languages provide sophisticated control constructions which virtually deem the GOTO statement unnecessary, although these languages usually do contain the statement itself. The irresponsible use of the GOTO statement is inherently dangerous as it may easily lead to unsafe, jumbled, and unstructured code.

9.4. 9.4 Selection statements

9.4.1. 9.4.1 Conditional statements

Conditional statements are used (1) when a choice has to be made between two activities at a given point in the program, or (2) for deciding whether to execute a given activity or not. The conditional statement in most languages is quite similar to (if not the same as) the following construction:

IF condition THEN action [ ELSE action ]

The condition is a logical expression.

The question is what kind of constructs may stand for an action in programming languages. Certain languages (e.g. Pascal) allow only one executable statement to be written. If the activity is too complex to be described with a single statement, several statements may be enclosed in so-called statement brackets. Pascal’s statement brackets are the BEGIN and END keywords. Statements enclosed in such brackets form a statement group. The statement group is formally considered a single statement. Another group of languages (because of their special syntax) allow that actions be expressed as a sequence of any number of executable statements (e.g. Ada). Finally, the third group of languages (e.g. C) claim that an action is either a single executable statement or a block (see Section 11.4).

Conditional statements may take a short (without ELSE) or a long (is ELSE) form.

The semantics of the conditional statement is the following:

First, the condition is evaluated. If the condition evaluates to true, the activity specified in the THEN branch is executed, and the program continues with the statement that follows the IF statement. If the condition evaluates to false, and an ELSE branch is included, the activity of the ELSE branch is executed; then the program continues with the statement immediately following the IF statement. If no ELSE is provided, an empty statement is executed.

IF statements may include other IF statements embedded in the THEN branch or the ELSE branch, which may give rise to the “dangling ELSE” problem. Consider the following scenario:

IF ... THEN IF ... THEN ... ELSE ...

Which conditional statement does the ELSE branch belong to? Is this a short IF statement which contains a long conditional statement, or is this a long IF statement with a short one in it THEN branch?

The following are possible answers:

a. One way to resolve the “dangling ELSE” problem is to always use long IF statements: if one of the branches would otherwise be unnecessary, an empty statement may be used.
b. If the reference language is silent on the issue, the solution is implementation-dependent. Most implementations claim that a free ELSE branch belongs to the nearest THEN branch that has no corresponding ELSE, i.e. interpretation takes place from the inside outwards. If applied on the example above, a long IF statement is embedded into a short one.

c. The syntax of the language makes the pigeonholes straightforward. The syntax of conditional statement in Ada is the following:

```ada
IF condition THEN executable_statements
[ ELSEIF condition THEN executable_statements...
[ ELSE executable_statements]
END IF;
```

In C, the condition is enclosed in round brackets, and there is no THEN keyword.

### 9.4.2 Case/switch statement

The case or switch statement represents a choice from any number of mutually exclusive activities at a given point in the program. The choice is based on the value of an expression. The syntax and semantics of case or switch statements varies with languages. We present some of these statements below.

**Turbo Pascal**

```pascal
CASE expression OF
constant_list : action
[ constant_list : action ...]
[ ELSE action ]
END;
```

The `constant_list` is a series of literals or intervals separated by commas. Literals may be used only once (i.e. two constant lists must not contain the same literal). The `expression` and the `constant_list` must be of enumeration type. The action is a statement or a statement group. It is not obligatory to specify an activity for every possible value of the expression.

The semantics of Pascal’s case statement is the following. After the `expression` is evaluated, its value is compared with the values of the constants in the order they are listed. If the value of the expression matches one of the constants, the activity identified by the constant list is executed. Control is transferred to the statement that follows the `CASE` statement. If the value of the expression does not match any of the constants, and there is an `ELSE` branch, the activity specified in the `ELSE` branch is executed. Control is transferred to the statement that follows the `CASE` statement. If there is no `ELSE` branch, an empty statement is executed.

**Ada**

```ada
CASE expression1 IS
WHEN { expression | range | OTHERS } [ |{expression | range | OTHERS ...}] => executable_statements
[ WHEN { expression | range | OTHERS } [ |{expression | range | OTHERS ...}] =>executable_statements...]
```

In C, the condition is enclosed in round brackets, and there is no THEN keyword.
END CASE;

The values of the expressions and domains in WHEN branches must not overlap. Only one WHEN OTHERS branch is allowed; if it is present, it must also be the last branch. expression1 must be of a scalar type. The programmer is expected to provide activities for all the possible values of expression1.

The semantics of Ada’s case statement is the following. After expression1 is evaluated, its value is compared with the values of the expressions or domains in the WHEN branches. If the value of the expression fits one of these value ranges, the statements in the corresponding WHEN branch are executed, and the program continues with the statement after the CASE statement. If there is no matching value, but there is a WHEN OTHERS branch, the statements of the WHEN OTHERS branch are executed, and the program continues with the statement after the CASE statement. If there is no WHEN OTHERS branch, a run-time error (exception) will occur. If we would rather not cater for some of the values, a WHEN OTHERS branch with an empty statement should be employed.

C

```c
SWITCH (expression) {
    CASE integer_constant_expression : [ action ]
    [ CASE integer_constant_expression : [action ...]]
    [ DEFAULT: action]
}
```

The type of the expression must be convertible to the integer type. The integer_constant_expression values of the CASE branches must be different. The action can be an executable statement or a block.

The semantics of C’s switch statement is the following. After expression is evaluated, its value is compared with the values of the CASE branches from top to bottom. If the value of the expression matches any of the values listed in the CASE branches, the activity in the specified branch is executed. Next, all the activities of the following branches are executed. If there is no matching value, but there is DEFAULT branch, the activity specified by the DEFAULT branch is executed, and control is transferred to the statement following the SWITCH statement. Note that in order to skip the rest of the branches of a switch statement in C a special statement should be used in the CASE branches (see the BREAK statement, Chapter 10).

PL/I

```pli
SELECT [ (expression1) ];

WHEN (expression [, expression...]) action
    [ WHEN (expression [, expression...]) action ...]
    [ OTHERWISE action ]
END;
```

Expressions may be of any type. An action is a statement, a statement group or a block.

The semantics of PL/I’s case statement is the following. If expression1 is present, its function is the same as in Ada, and similarly, all the possible values of expression1 must be taken into consideration. If there is no expression1, the value of every expression specified in the WHEN branches is converted into bit sequences. The first branch whose expression evaluates to a non-zero bit sequence is selected. If all the expressions evaluate to zero, an empty statement is executed.

FORTRAN and COBOL do not support case / switch statements.
9.5. Loop statements

Loop statements make it possible to repeat a certain activity any number of times at a given point in the program.

The general structure of a loop is the following:
- head
- body
- tail

Information concerning the specifics of the repetition is specified in the head or in the tail.

The body contains the executable statements to be repeated.

There are two extreme cases when executing loops. One extreme is when the body is never executed, which is called an empty loop. The other extreme is if loop never ends, which is called an infinite loop.

Programming languages distinguish between the following kinds of loops: (1) conditional, (2) count-controlled, (3) enumerated, (4) infinite, (5) composite loop.

The following subsections describe these loop types in detail.

9.5.1. Conditional loops

The repetition of the statements enclosed by the loop depends on the value of a logical condition. The condition may be placed in the head or in the tail. Based on their semantics, we distinguish between pre-conditional and post-conditional loops.

Loops with pre-condition:

The condition is specified in the head. First, the condition is evaluated. If it is true, the body of the loop is executed once. Then the condition is evaluated again, and the body is executed again, until the condition turns false. There must be one or more statements in the body that contribute to changing the value of the condition.

A pre-conditional loop may be an empty loop if the condition evaluates to false when the control reaches the loop; the pre-conditional loop is an infinite loop if the condition is true when the control reaches the loop, and remains true.

Loops with post-condition:

The post-condition generally belongs to the tail, but some languages put the post-condition in the head.

In post-conditional loops the body is executed first, and then the condition is evaluated. Generally, if the condition is false the body is executed repeatedly. The iteration lasts until the condition turns true. Note that there are languages which repeat under a true condition. Apparently it is necessary to ensure that the value of the condition is altered in the body. Post-conditional loops can never be empty loops, since the body is always executed at least once. They can, however, turn into infinite loops if the value of the condition never changes.

9.5.2. Count-controlled loops

Information on the repetition of the loop (loop parameters) is specified in the head. Loop parameters always include a special variable, the so-called index or loop variable which determines the number of repetitions. The body is executed for every value taken by the variable. The variable may take on values from the range specified in the loop head with lower bound and upper bound. The loop variable may take on all the items from the specified range (including the lower bound and the upper bound values), or just certain values at regular distances (equidistant values). In the latter case the programmer must specify the step size, which determines the distance between neighboring index values. The variable may take the domain values in ascending or descending order; the order in which the variable takes the values from the domain is determined by the direction.

Languages take different sides in following issues raised by count-controlled loops:

1. What types may loop variables have?

   • Every language allows the integer type.
• Certain languages support enumeration types.
• Few languages allow the real type.
• The type of the lower bound value, the upper bound value and the step size must be of the same type as the loop variable, or compatible with it.

2. How are the lower bound, upper bound and the step size specified?
• Every language allows these parameters to be specified with a literal, a variable or a named constant.
• More recent languages support the use of expressions.

3. How can direction be defined?
• In languages which support only numeric type loop variables the direction depends on the sign of the step size. If the step size is positive the direction is ascending; a negative step size indicates descending direction.
• With a keyword.

4. How many times are loop parameters evaluated?
• Usually once, when the control first reaches the loop, and the parameters remain unchanged while the loop is being executed.
• Loop parameters are evaluated each time before the body is executed.

5. How does the loop terminate?
• Regular execution
  – as defined by the loop parameters;
  – with a specific statement in the loop body.
• With a GOTO statement, considered an irregular way of terminating the loop.

6. What is the value of the loop variable after the loop terminates?
• If control has been transferred as a result of a GOTO statement, the value of the loop variable will be the last value it was assigned.
• If the loop has terminated regularly, the value of the variable depends on what the reference language claims. Certain languages do not state anything about the issue, while other languages consider the value of the loop variable undefined.
• Implementations, on the other hand, claim the followings:
  - The value of the loop variable is the value it was assigned before the execution of the last loop.
  - The value of the loop variable is the value it was assigned after the last loop was executed.
  - The value of the loop variable is undefined.

Based on their semantics, count-controlled loops take either the test before or test after approach. Since few reference languages specify their approach, it is implementation-dependent in most cases. Most implementations support a variant of the test before approach.

The semantics of the test before approach:
First the loop parameters are evaluated, after which the run time system examines whether the specified range is empty or not in the specified direction. If the range is empty (e.g. the [10..1] range in ascending order), the loop is considered an empty loop. Otherwise the loop variable is assigned the lower bound as initial value, and the body is executed. Then the run time system examines whether the specified range includes the next value in the specified direction at the given step size, compared to the
present value of the loop variable. If such a value is found the variable accepts it as the next value, and the body is executed. If there is no such value the loop terminates (regular termination).

The semantics of the test after approach:
First the loop parameters are evaluated, after which the loop variable is assigned the lower bound as initial value and the body is executed. Then the run time system examines whether the specified range includes the next value in the specified direction at the given step size, compared to the present value of the loop variable. If such a value is found the variable accepts it as the next value, and the body is executed. If there is no such value the loop terminates.

Procedural languages usually allow the programmer to assign a new value to the loop variable in the body of the loop. Test-before count-controlled loops may be empty loops, but test-after loops cannot. However, it is possible for both test-before and test-after loops to turn into infinite loops in certain languages.

9.5.3. 9.5.3 Enumeration-controlled loops

Enumeration-controlled loops can be conceived of as the generalization of count-controlled loops. Enumeration-controlled loops have a loop variable whose values are defined explicitly. The body is executed for each specified value. The loop variable and its values are defined in the head; the values are specified with expressions. The loop variable may be of any type. Enumeration-controlled loops can never be empty or infinite.

9.5.4. 9.5.4 Infinite loops

Infinite loops provide no information on the iteration neither in the head, nor in the tail. Considering its semantics, an infinite loop can never be an empty loop. Infinite loops must contain a statement which helps terminate the loop. Infinite loops are particularly effective for implementing event-controlled applications.

9.5.5. 9.5.5 Composite loops

Composite loops are arbitrary combinations of the previous loop types. Their head may include any amount of information concerning the number of repetitions, and the semantics of such loops is accordingly intricate. Composite loops allow the programmer to design loops of arbitrary complexity.

Certain languages support statements that transfer control to another part of the program. When used in the body of loops, such statements cause the regular termination of the loop, and are the only way of regularly terminating an infinite loop.

9.6. Questions

1. How can we classify statements?
2. Assignment statement.
3. Empty statement.
5. Conditional statement.
6. Case/switch statements.
7. Loop organizer statements.
8. Conditional loop.
10. Enumerated loop.
11. Infinite loop.
12. Composite loop.
10. fejezet

10 EXAMPLES OF LOOP STATEMENTS IN LANGUAGES

10.1. FORTRAN

Count-controlled loops:

```fortran
DO label variable = l, u[, s]
execuable_statements
label statement
```

The `variable, l, u, s` must be of an integer type. `l, u, s` (lower bound, upper bound, step size) must be declared as variables; the use of expressions is not allowed. If the step size `s` is not specified, its default value is 1, and the direction is ascending. If `s` is given, its sign determines the direction. Implementations prefer the test after approach. Control flow statements are not allowed at the end of the loop, an empty statement should be written there instead.

Later versions support pre- and post-conditional loops as well.

10.2. PL/I

PL/I support all the loop types introduced earlier.

Pre-conditional loop:

```pli
DO WHILE(expression);
body
END;
```

The `expression` must be convertible to a bit sequence; evaluates to false if every bit is 0, otherwise evaluates to true.

Post-conditional loop:

```pli
DO UNTIL(expression);
body
```
Count-controlled loop:

```plaintext
DO variable = l TO u [BY s]
body
END;
```

The variable must be of an arithmetical or a bit sequence type. The direction is determined by the sign of the step size \( s \). An infinite loop if TO \( u \) is omitted.

Enumeration-controlled loop:

```plaintext
DO variable = expression1 [, expression2 ...] ;
body
END;
```

The expressions may be of any type.

It is possible to combine various kinds of loops which result in composite loops.

### 10.3. Pascal

Pre-conditional loop:

```plaintext
WHILE condition DO executable_statement;
```

Post-conditional loop:

```plaintext
REPEAT executable_statements UNTIL condition
```

Count-controlled loop:

```plaintext
FOR variable = l { TO | DOWNTO } u DO executable_statement;
```

The count-controlled loop takes test before approach. The direction depends on the keyword: TO indicates ascending, DOWNTO descending direction. Pascal does not recognize the concept of the step size.

### 10.4. Ada

Pre-conditional loop:

```plaintext
WHILE condition
LOOP
executable_statements
END LOOP;
```
Count-controlled loop:

```ada
FOR variable IN [ REVERSE ] range LOOP
  executable_statements
END LOOP;
```

The count-controlled loop takes test before approach. `REVERSE` indicates a descending direction. If `REVERSE` is omitted, the direction is ascending. The `range` is of an enumeration type. Ada does not recognize the notion of step size. The loop variable is declared implicitly, its type is the same as that of the range elements. The loop variable is considered a named constant in the loop body, which implies that its value cannot be changed.

Infinite loop:

```ada
LOOP
  executable_statements
END LOOP;
```

It is possible to terminate the loop with the `EXIT` statement in Ada. Besides, this is the only way to end an infinite loop.

### 10.5. C

Pre-conditional loop:

```c
WHILE(condition) executable_statement;
```

The `condition` is of an integral type. The loop body is repeated if the `condition` evaluates to other than 0.

Post-conditional loop:

```c
DO executable_statement WHILE(condition);
```

The loop body is repeated if the `condition` evaluates to other than 0.

FOR-loop:

```c
FOR([expression1]; [expression2]; [expression3]) executable_statement;
```

The FOR-loop can be paraphrased as follows:

```c
expression1;
WHILE(expression2) {executable_statement; expression3;}
```

`expression1` is called the initializer expression, it is evaluated before loop body is executed. `expression2` is responsible for terminating the loop, it is a condition; `expression2` is evaluated each time loop body has been executed. If there is no `expression2`, the FOR-loop is an infinite loop, which can be terminated regularly with the `BREAK` statement.
10.6. Control flow statements in C

Besides the ones already discussed, C supports the following control statements.

```
CONTINUE;
```

It can be included in the body of a loop. This statement makes the run time system skip the remaining statements of the loop body, examine the loop condition, and either start a new loop step, or end the loop.

```
BREAK;
```

It can be placed in the body of a loop or in the CASE branch of a SWITCH statement. It makes the loop end regularly, and exits the switch statement.

```
RETURN [expression];
```

It causes a function to end regularly, and gives back the control to the caller (see chapter 14.).
11th chapter

11. THE STRUCTURE OF PROGRAMS

In procedural languages, the text of the program can be broken down into more or less independent parts, called *program units*. Specific languages differ in their treatment of the following issues:

1. *It is necessary to compile the full text of the program in one piece, or is it possible to break the text into smaller parts which can be compiled independently?*

   • In certain languages, the program consists of physically independent parts which can be compiled separately. These parts are not structured in depth.

   ![Diagram of independent parts]

   • In other languages, the program comprises only one compilation unit; however, the text of such program units may be structured. The units are not independent physically.

   ![Diagram of dependent parts]

   • Languages which support the combination of the former types allow programmers to compose physically independent program units with optional inner structure.
2. If program units can be compiled separately, what does an independent compilation unit consist of?

3. What kinds of program units exist?

4. What is the relation between the program units?

5. How do the program units communicate with each other?

Procedural languages support the following program units:

- subprogram;
- block;
- package;
- task.

11.1. 11.1 Subprograms

Subprograms are the first manifestation of procedural abstraction in procedural languages. Subprograms play a fundamental (in fact, determining) role in the procedural paradigm. A subprogram maps an input data group onto an output data group. The specification of the subprogram provides the description of the data, but the actual mapping is not revealed. The programmer has knowledge of the specification, but is not acquainted with the implementation.

Subprograms are an essential means of code reuse. It is good practice to use subprograms when a certain piece of program is repeated at various points in the source code. By extracting the recurring piece of program the code has to be written only once, and referred to wherever the code is meant to be reused (originally repeated), i.e. the subprogram is called or activated.

It is the use of formal parameters that makes subprograms a means of abstraction; parameterized code is more general than its specific occurrences.

Formally, a subprogram is built from the following elements:

- head or specification;
- body or implementation;
- end.

A subprogram consists of four components:

- name;
- formal parameter list;
- body;
- context.

A name is an identifier which always belongs to the head. The formal parameter list is also part of the specification; a formal parameter list contains identifiers which (1) name custom programming objects in the body of the subprogram; (2) describe the role of the parameters to be concretized at specific subprogram calls with actual parameters.
In the early languages, the formal parameter list contained only the names of the parameters. Later languages allow additional information to be specified that control the behaviour of the parameters during the execution of the subprogram. The formal parameter list is enclosed in round brackets. Languages differ in whether they consider brackets part of the formal parameter list, or the name.

The formal parameter list can be empty, in which case the subprogram has no parameters.

The **body** is composed of declarations and executable statements. Certain languages require that these two kinds of statements should be separated from each other; the body has a declaration part and an executable part. Other languages allow the two kinds of statements to be mixed in an arbitrary fashion.

The programming objects declared within a subprogram are the **local objects** of the subprogram; their names are **local names** to the subprogram. Local names are not accessible from outside the subprogram, they are hidden. **Global names**, as opposed to local names, are declared “outside” relative to the subprograms, but can be referred to in the body of subprograms in a regular way.

The **context** of a subprogram is the collection of the global variables.

Subprograms are of two kinds: **procedures** and **functions**.

A procedure is a subprogram that performs a particular activity. The results of the activity can be used at the point where the procedure was called. The effects of a procedure call include changes to the parameters of the subprogram, changes to the context, and the execution of the tasks determined by executable statements in the body.

A function is a subprogram whose role is to determine a single value of the type declared as the return type in the specification of the function. The name of a function stands for its return value, which is transferred to the point of the function call. The role of the executable statements of the function’s body is to determine the return value.

The phenomenon of a function’s changing any of its parameters or the context is called the **side effect** of the function. Side effects are usually considered harmful.

Procedures are activated in a statement-like fashion, i.e. procedure calls can be placed wherever an executable statement may stand. Certain languages dedicate a keyword (very often the keyword `CALL`) to the act of calling a procedure. Other languages do not designate a word to procedure calls. Control is transferred to the procedure when the procedure is called.

Formally, procedure calls are of the following form:

```
[keyword] procedure_name(actual_parameter_list)
```

A procedure completes **regularly** if one of the following conditions holds:
- control reaches the end of the procedure;
- the procedure is terminated with a dedicated statement in the body of the procedure.

In the case of regular completion the program continues with the statement immediately following the procedure call.

The following instances of procedure termination are not considered regular:
- Most languages make it possible to leave the procedure with a `GOTO` statement and continue the program on the given label.
- There exist language features which cause the whole program to terminate. Terminating the program results in the abrupt completion of the procedure, too; control is transferred back to the operating system.

Functions must be invoked in **expressions**. The form of the function call is the following:

```
function_name(actual_parameter_list)
```

On regular completion of the function the control returns into the expression and continues its evaluation.

The return value of a function may be determined in any of the following ways:
- The name of the function can be used as a variable in the body of the function (e.g. as in FORTRAN); its value can be changed in the body as required. The last value is the return value.
- A value must be assigned to the name of the function in the body of the function. The name of the function carries the value it was assigned the last time.
- A special statement is used to determine the return value, which also terminates the function at the same time.

A function terminates regularly if one of the following conditions holds:

- control reaches the end of the function, and a return value has been specified;
- a terminating statement is reached, and a return value has been specified;
- a terminating statement is reached, which determines the return value, too.

The following instances of function termination are not considered regular:

- control reaches the end of the function, but the return value is not specified;
- a terminating statement is applied which does not determine the return value, and the return value has not been specified in any other manner;
- control leaves the function as a result of a GOTO statement.

In the last case, control is transferred to the label, therefore we do not return to the evaluation of the containing expression. It is best to avoid such constructs as they result in unsafe code. In the first two cases, control returns to the evaluation of the expression, but the return value of the function is undefined. This might not pose any difficulties for certain languages (e.g. C) in particular situations, but it is generally regarded as a semantic error.

Every program written in a procedural programming language must contain a special program unit, the main program. Formally, the main program is similar to a subprogram; this is the program unit to which the loader transfers control, and it is responsible for coordinating the other program units. A program completes regularly when the end of the main program is reached, and control is transferred back to the operating system.

11.2. 11.2 The Call Chain and Recursion

A program unit may call another program unit, which in turn may call another program unit etc.; this is how the call chain is built. The first member of the call chain is always the main program. Every member of the call chain is active, but only the most recently activated unit is in operation. Under normal conditions, the most recently activated program unit finishes operating first, and the control returns to the program unit which it was called by. The call chain is built up and destructed dynamically during the execution of the program.

The phenomenon of calling an already active subprogram is recursion.

Recursion may be of two kinds;

- direct: a subprogram calls itself, i.e. the body includes a reference to the self.
- indirect: a subprogram calls another subprogram which precedes it in the call chain.

An algorithm implemented by recursion can always be rewritten as an iterative algorithm.

Certain languages do not support recursion (e.g. FORTRAN). Another group of programming languages claims that every subprogram is recursive by default. Other languages allow the programmer to decide whether a given subprogram should be recursive or not.

11.3. 11.3 Secondary Entry Points

Certain languages allow programmers to call a subprogram not only through the head, but also through a so-called secondary entry point specified in the body of the subprogram. If a secondary entry point is present, the subprogram can be activated either by its name or the name of the secondary entry point. The specification of the secondary entry point must comply with the specification of the subprogram. In the case of functions, the return type of the secondary entry point must be the same as the one specified in the head of the function. If the subprogram is entered through the head, the whole body of the subprogram is executed, while in the case of secondary entry points only a part of the body is executed.
11.4. 11.4 Block

A block is a program unit which may occur only within another program unit. Formally, a block consists of a beginning, a body and an end. The beginning and the end of a block are indicated by a special character sequence or a reserved word. The body contains declarations and executable statements. Similarly to what has been mentioned about the structure of subprograms, these statements may either be mixed in an arbitrary fashion, or there is a separate declaration part and an executable part. Blocks cannot have parameters, but may have a name in certain languages (usually specified in the form of a label placed before the beginning of the block). Blocks may occur anywhere in the source code where executable statements are allowed.

A block is activated when it is reached by the control sequentially, or a GOTO statement jumps to its beginning. A block terminates if control reaches its end, a GOTO statement is used to leave the block, or if the whole program is terminated from within the block.

Not all procedural languages recognize blocks. The role of blocks is to help define the scope of names.

11.5. 11.5 Compilation Unit

In procedural languages, a program is composed of compilation units. Compilation units are pieces of source code which can be compiled in a physically independent or separate way. Compilation units often help define the scope and sometimes the lifetime of the programming constructs they contain.

11.6. Questions

1. What are the components of a subprogram?
2. What is the form of the formal parameter list?
3. What is a procedure?
4. What is a function?
5. What is the side effect of a subprogram?
6. When does a function complete regularly?
7. When does a procedure complete regularly?
8. What is the secondary entry point?
9. What is the block?
10. What is the call chain?
11. What is meant by recursion?
12. What is a compilation unit?
12. fejezet

12 PARAMETER EVALUATION AND PARAMETER PASSING

Parameter evaluation is the process of mapping formal and actual parameters when a subprogram is called; evaluation also determines the information which governs the communication of the parameters.

The formal parameter list is primary for parameter evaluation; formal parameters are defined in the specification of the subprogram, and they are declared only once. As opposed to formal parameters, actual parameters (or arguments) are specified in the calls themselves, which implies that there may be as many actual parameters as the number of times the subprogram is called. Therefore, it is always the formal parameter list which is determinative to parameter evaluation, since actual parameters are assigned to formal parameters.

The following three issues are closely related to parameter evaluation:

1. Which actual parameter will be assigned to which formal parameter?

   The order in which actual parameters are mapped onto formal parameters is determined by binding, which can be ordinal or name binding.

   In the case of ordinal binding, actual parameters are assigned to formal parameters based on their relative order in the parameter list: the first argument is assigned to the first formal parameter, the second argument to the second parameter, and so on. All languages support ordinal binding, and it is the default for most languages.

   In the case of named parameter binding, the mapping of the parameters is specified in the actual parameter list by supplying the name of the formal parameter next to the actual parameter in compliance with the relevant syntactic conventions; the order of the formal parameters is irrelevant. Few programming languages support name binding.

   It is possible to combine ordinal and named binding in such a way that the first parameters in the list are bound by order, while the rest are bound by name.

2. How many actual parameters must be passed to the subprogram call?

   The number of formal parameters is fixed if the formal parameter list contains a predefined number of parameters. In this case, parameter evaluation takes place in one of the following ways:

   - The number of actual parameters must equal the number of formal parameters.
   - The number of actual parameters may be less than the number of formal parameters, provided that parameters are called by value. Formal parameters which have not been assigned an actual parameter receive a default value in the formal parameter list.

   In some languages, the number of formal parameters is not necessarily fixed, i.e. their number is arbitrary. If the formal parameter list is of arbitrary length then the number of actual parameters is also arbitrary. It may be possible to specify the lower bound of the number of actual parameters that must be passed onto a subprogram call.

3. What is the relation between the types of the formal and the actual parameters?

   Certain languages support type equivalence, which means that the type of the actual parameter must be exactly the same as the type of the formal parameter. Other languages recognize the principle of type compatibility, i.e. the type of an actual parameter must be convertible to the type of the corresponding formal parameter.
12.1. Parameter Passing

Parameter passing is a means of communication between subprograms and other program units. Parameters are passed by the caller (any kind of program unit) to the called subprogram. Languages differ in the direction and the type of information they allow to be passed to the subprogram.

Languages distinguish between the following parameter passing modes:

- call by value;
- call by reference;
- call by result;
- call by value-result;
- call by name;
- call by need.

Call by value (or pass by value) means that the formal parameters have an address component at the area of the called subprogram, while the actual parameters necessarily have a value component at the calling side. The value of the actual parameter is determined during parameter evaluation, and then transferred to the area allocated to this component on the subprogram’s side. The formal parameter receives an initial value, which is used by the subprogram. The information flow is unidirectional (from the caller to the called). The called subprogram is not aware of its caller; it works with its own memory area. The call-by-value mode always entails copying a value of arbitrary complexity. A definite drawback of call-by-value is that copying large value groups may take a long time. On the other hand, call-by-value allows the interacting program units to work independently, since they do not affect each other apart from the initial value copy. Actual parameters are expressions.

In call-by-reference evaluation, formal parameters have no address component at the memory region allocated to the called subprogram. Actual parameters, on the other hand, have an address component in the region that belongs to the caller. Parameter evaluation involves determining the address of the actual parameter and passing it to the called subprogram, more specifically, assigning the address component to the appropriate formal parameter. The called subprogram operates on the caller’s side. Information flow is bidirectional: the subprogram may receive values from the caller, and it is allowed to write to the caller’s region that it has access to. Call-by-reference is much faster and more efficient than call-by-value as values are not copied; however, call-by-reference may be unsafe since the called subprogram can abuse its privileges and use the information stored at the caller’s region in an undesirable way. The actual parameter is a variable.

In the case of call-by-result evaluation, both the formal and the actual parameters have an address component in the region of the called subprogram. The address of the actual parameter is determined and passed to the subprogram during evaluation. The subprogram operates on its own memory region, and uses the address of the actual parameter only to copy the result once it has finished its operation. Communication is unidirectional, information flows from the called to the caller. Call-by-result involves copying values. The actual parameter is a variable.

In call by value-result, formal parameters have an address component at the region of the called subprogram, while actual parameters possess both value and address components. Parameter evaluation determines the value and the address of actual parameters and passes these components to the subprogram. The subprogram makes use of the received value in its operation, and copies the value of the formal parameter to the address of the actual parameter once it finishes its operation. Communication is bidirectional; values are copied twice. The actual parameter is a variable.

In call-by-name evaluation, actual parameters are symbol sequences whose interpretation depends on the given context. During evaluation, the context of the subprogram is recorded first, which specifies the interpretation of the actual parameters. This is followed by the replacement of formal parameter names with the corresponding symbol sequences in the source code. Finally, the body of the subprogram is executed. The direction of the information flow depends on the interpretation of the actual parameters in the given context.

Call-by-need is a variant of call-by-name evaluation, the only difference being that the subprogram is launched before parameter evaluation. In call-by-need, evaluation is on demand in the sense that the context of actual parameters is recorded and formal parameter names are replaced only when the execution of the subprogram reaches the first occurrence of a formal parameter’s name.

Subprograms do not accept type parameters.

The choice of parameter evaluation is determined by one of the following conditions:
- The language supports only one parameter passing mode (e.g. C).
- The parameter passing mode must be specified explicitly in the formal parameter list (e.g. Ada).
- Evaluation depends on the type of the actual and the formal parameters at the same time (e.g. PL/I).
- Evaluation depends on the type of the formal parameter (e.g. FORTRAN).

Formal parameters may fall into one of the following categories:
- **Input parameters**: allow the subprogram to receive information from the caller (e.g. call by value);
- **Output parameters**: the called subprogram passes information to the caller (e.g. call by result);
- **Input-output parameters**: information may flow in both directions (e.g. call by value-result).

### 12.2. Questions

1. Describe the process of parameter evaluation.
2. Describe the various parameter passing modes.
3. What can decide the mode of parameter passing?
4. How can we classify formal parameters?
13. fejezet

13 SCOPE

The scope of a name refers to that part of the source code where the name refers to the same programming entity, i.e., its meaning, use, and characteristics are the same. Note that only names have scope. Scope is also known as visibility.

In procedural languages, scope applies to program units and compilation units.

A name declared within a program unit is a local name to that program unit. Names declared outside the program unit but referred to from therein are free names.

The process of determining the scope of a name is called scoping. Scoping may be static or dynamic.

Static scoping is a compile-time activity performed by the compiler. Static scoping is based on the program unit structure of the source code. If the compiler comes across a free name, it exits the current program unit to check whether the name is declared locally by the containing unit. If it is, the process ends; otherwise the compiler keeps looking for the declaration of the name in the containing units until it finds the declaration or reaches the outmost level. If the compiler reaches the outmost level without having found the declaration of the name, there are two possibilities:

- A compilation error occurs if the language requires that programmers declare every custom name.
- The name is considered local to the outmost unit if the language supports automatic declaration. The compiler assigns the required attributes to the name according to the rules of automatic declaration.

With static scoping, the scope of a name includes the program unit in which it was declared and every nested program unit in which the name is not redeclared.

Scopes spread inwards, never outwards. Program units encapsulate their local names so that they are not visible to the outside world. A name which is not local to a program unit but is visible from there is called a global name. The issue of whether a name is global or local is relative and context-dependent. The very same name can be local to program unit A, global from the perspective of unit B, and may remain invisible to unit C.

Example:
Fig. 13-1. Nested program units

Figure 13-1 illustrates nested program units. Variable \( x \) of type \( \text{int} \) is declared in unit 1; \( x \) is a local name of 1. \( x \) can be referenced by program units 2, 4 and 5, its name is global to these units. The name \( x \) is redeclared in unit 3 with a \( \text{float} \) type; the redeclared name in unit 3 is now a different variable. The variable \( x \) of type \( \text{float} \) is local to unit 3, and cannot be seen by any other program units. Note that it is impossible to reference variable \( x \) of type \( \text{int} \) in unit 3 since it is hidden by the new declaration. In other terms, there is a “hole” in the scope of the \( x \) declared by unit 1.

Dynamic scoping is a run-time activity performed by the run-time system. Dynamic scoping is based on the call chain. If the run-time system finds a free name, it steps back up the call chain until it finds the name as a local name or reaches the root of the chain. In the latter case, either a run-time error occurs, or the name is declared automatically.

With dynamic scoping, the scope of a name includes the program unit in which it was declared and every other unit in the call chain starting from this program unit unless the name is redeclared afterwards. In the latter case, only the redeclared name is visible to the further elements of the call chain, there is no “hole” in the scope.

With static scoping, every name has a clear scope based on the structure of the source code. In the case of the dynamic scoping, however, scopes may change during run-time and between the program calls.

Procedural languages support static scopes. The formal parameters of subprograms are considered local objects from the perspective of the subprogram, and their names are local to the subprograms. On the other hand, names of program units are global to the various units. Keywords as names are visible from every point of the program. Standard identifiers as names are accessible to those program units in which they are not redeclared.

### 13.1. Questions

1. What is the scope?
2. What is the scoping?
3. What is meant by a “hole” in the scope of a name?
4. What is the difference between static and dynamic scoping?
14. fejezet

14 EXAMPLES OF SPECIFIC LANGUAGE FEATURES

This chapter illustrates the implementation of features mentioned in Chapters 11-13 in specific languages (FORTRAN, PL/I, Pascal, Ada, C).

14.1. FORTRAN

FORTRAN programs are composed of physically independent program units, which can be compiled separately, but cannot be nested. FORTRAN supports only subprograms. A compilation unit is either a single program unit, or any combination of several program units. Since there are no global variables in FORTRAN, program units use common blocks for communication. The common block can be read and written by each program units that contains the following declaration:

```
COMMON [/n/] a1 [(d1)] [, a2 [(d2)]...]
```

where

- \( a_i \) is a variable of a scalar or an array type; the type of \( a_i \) must be declared in a separate declaration statement;
- \( d_i \) declares the dimension of the array;
- \( n \) is the name of the common block.

A program may define any number of common blocks, which in turn may contain any number of variables. The system places the variables in a dedicated area of the memory in the order they are listed. The organization of the common memory area may differ from subprogram to subprogram; the types of the variables may also vary. The shared memory area is distributed evenly among named blocks; variables which occur in unnamed common blocks are placed at the end of the shared memory region.

FORTRAN support automatic declaration.

The main program has no starting statement.

Procedures are of the following form:

```
SUBROUTINE name[formal_parameter_list)]

declarations

executable_statements

END
```
The formal parameter list contains only names whose type must be specified in the declaration section. Procedures are activated by the `CALL` keyword, and terminated regularly by the `RETURN [n]` statement, where \( n \) is an unsigned integer.

FORTRAN supports secondary entry points of the following syntax:

```fortran
ENTRY name [(formal_parameter_list)]
```

Parameter evaluation is characterized by ordinal binding, type equivalence, and numerical matching. Parameters in the formal parameter list may have any of the following forms:

1. The formal parameter is an identifier which may be used in the body as an ordinary variable, or as name of a subprogram.
2. Identifiers enclosed within slashes (/) may only name variables with a scalar type.
3. The formal parameter may be \* too.

Formal parameters of a scalar type accept expressions as actual parameters. Array type parameters take array type or indexed variables as arguments. Subprogram names are parameters require subprogram name arguments. If the formal parameter is \*, it will accept a label argument of the form `&label;` in the latter case, `RETURN n` is also applicable, which may be used to transfer control back to the \( n \)-th label of the actual parameter list.

Parameter passing depends on the type of the formal parameter:

- For the array type, it is call-by-reference;
- For scalar types, it is call-by-value;
- For names of subprograms, it is call-by-name;
- For `/identifier/`, it is call-by-reference;
- For the `*`, it is call-by-reference.

![Table 14-1. Summary of the behaviour of formal parameters in FORTRAN.](image)

FORTRAN functions are of the following form:

```fortran
[type] FUNCTION name(p1 [, p2...] )
declarations
executable_statements
END
```

FORTRAN functions are required to have at least one formal parameter, but the \* cannot be included among the formal parameters. The type of the function must be declared in the head or in the declaration section. Secondary entry points are also allowed. The name of the function can be referred to as a local variable in the body of the function. The `RETURN` statement is used to exit the function; however, it does not determine the return value. The return value is calculated by assigning values to the function name in the body. It is the last value assigned to the function name which will be returned when the `RETURN` statement is executed. Functions to be called from a subprogram must be declared with their names in the subprogram.

Earlier versions of FORTRAN do not support recursion.

The following FORTRAN function illustrates how factorial values are calculated:
14.2. PL/I

PL/I supports the following program units: subprogram, block, and task. Subprograms may be independent, or nested within each other. PL/I also distinguishes the main program, which is a special subprogram. A compilation unit is composed of the main program, the subprograms and their arbitrary combination.

The syntax of PL/I subprograms is the following:

```plaintext
name:PROCEDURE [(formal_parameter_list)] [OPTIONS(option_list)]
[RECURSIVE] [RETURNS(attributes)];
statements
END [name];
```

The declaration and the executable part are not separated, statements can be mixed. The `name` of the subprogram is specified as a label. The `formal_parameter_list` contains only the names of the parameters; their specifics must be declared in the body.

The `option_list` controls the run-time behaviour of the subprogram. The option `MAIN` indicates that the subprogram is the main program; this option excludes all other options. If `MAIN` is not specified, the subprogram is a function or a procedure. The programmer may decide whether the subprogram should be recursive or not with the `RECURSIVE` keyword.

The keyword `RETURNS` indicates that the subprogram is a function; `attributes` specify the attributes of the return value.

PL/I procedures are activated by the `CALL` statement.

The procedure completes regularly if control reaches the end of the procedure’s body, or if a `RETURN` statement is executed in the body.

A function completes regularly as result of the `RETURN(expression);` statement, where `expression` denotes the return value of the function.

Secondary entry points in PL/I have the following syntax:

```plaintext
name:ENTRY [(formal_parameter_list)] [RETURNS(attributes)];
```

Secondary entry points to functions must have the same attributes and formal parameter list as those specified in the head; secondary entry points to procedures may differ.

Parameter evaluation is characterized by ordinal binding, numerical matching and type compatibility.

Parameter passing may be
• passed-by-reference, if the actual parameter is a label or a variable of the same type as the formal parameter;
• passed-by-name, if the actual parameter is the name of an entry point or a file name;
• passed-by-value otherwise.

Blocks have the following syntax:

```plaintext
[label:] BEGIN
    statements
END [label];
```

Declaration and executable statements may be combined in an arbitrary fashion.

In PL/I, the lifetime of programming objects is controlled by the following attributes:

• **STATIC**: refers to static memory allocation.
• **AUTOMATIC**: refers to dynamic memory allocation; this is the default attribute.
• **CONTROLLED**: memory allocation is controlled by the programmer. The programmer may use the following statements:
  - **ALLOCATE**: instructs the system to place a variable into the memory.
  - **FREE**: release the memory area used by an object.
• **BASED**: refers to the use of based variables. Variables are assigned an address component by the programmer relative to the address of an object allocated previously. The address is relative, absolute addresses are usually hidden. This mode does not support absolute memory allocation by the programmer.

PL/I supports static scopes.

It is possible to declare variables with the same name in different compilation units global by the **EXTERNAL** attribute if all their attributes are the same. Subprogram names at the external level have **EXTERNAL** attribute.

Attributes may be declared automatically in PL/I.

The following code demonstrates the iterative version of a PL/I function which calculates the factorial:

```plaintext
FACT:PROCEDURE(I);
    F=1;
    DO L=2 TO I;
        F=F*L;
    END;
    RETURN(F);
END FACT;
```

The following code shows the recursive version of the same function:

```plaintext
FACT:PROCEDURE(I) RECURSIVE;
    F=1;
    IF I>1 THEN F=I*FACT(I-1)
RETURN(F);
END FACT;
```
14.3. Pascal

The compilation unit of Pascal is the main program. Pascal supports only the subprogram from among the possible program units; certain versions also support packages (e.g. the unit of Turbo Pascal).

The syntax of the main program is the following:

```
PROGRAM name [(environmental_parameters)];

declaration_section

BEGIN

executable_statements

END.
```

The main program has a formal parameter list, but the number of formal parameters is not fixed. The formal parameters of the main program play an important role in communicating with the operating system. Subprograms must be nested in the declaration section of the main program. The structure of subprograms is similar to the structure of the main program, including the way of nesting other subprograms.

Procedures have the following syntax:

```
PROCEDURE name[(formal_parameter_list)];

declaration_section

BEGIN

executable_statements

END;
```

Functions have the following syntax:

```
FUNCTION name[(formal_parameter_list)] : type;

declaration_section

BEGIN

executable_statements

END;
```

The formal parameter list consists of parameter groups separated by semicolons. Parameter groups have the following syntax:

```
[ VAR ] identifier [, identifier...] : type
```

If the `VAR` keyword is included in the parameter group, parameter passing is call-by-reference, otherwise it is call-by-value. Parameter evaluation is characterized by ordinal binding, numerical matching, and type equivalence.

There is no dedicated keyword for procedure calls. Subprograms are recursive by default.
The name of the function must be assigned a value by the executable statements; upon regular completion, the function returns with the value it was assigned the last time. A procedure completes regularly if control reaches its end.

Pascal supports dynamic lifetime management and programmable memory allocation.

In Pascal, a name is visible from its declaration.

The following example demonstrates a Pascal function calculating the factorial:

```pascal
FUNCTION FACT(I:INTEGER):REAL;
BEGIN
    IF I=0 THEN FACT:=1
    ELSE FACT:=FACT(I-1)*I;
END;
```

### 14.4. Ada

Ada supports all the program units that have been introduced in earlier chapters. Whereas other languages use a linkage editor to relate physically independent units which have been compiled separately, the Ada compiler applies consistency checking during compilation. There is no dedicated main program; it is implementation dependent which subprogram starts first.

Procedures in Ada have the following syntax:

```ada
PROCEDURE name[(formal_parameter_list)]
IS
    declaration_part
    executable_statements
END [name];
```

Functions in Ada have the following syntax:

```ada
FUNCTION name[(formal_parameter_list)] RETURN type
IS
    declaration_part
    executable_statements
END [name];
```

There is no dedicated keyword for procedure calls. A procedure completes regularly as a result of a `RETURN` statement, or if the `END` is reached. Functions are terminated by the following statement:

```ada
RETURN expression;
```

The formal parameter list consists of parameter groups separated by semicolons. Parameter groups are declared with the following syntax:
name[,name...] : [mode] type [:= expression]

mode determines the direction of parameter passing. Ada defines three keywords to this end: **IN**, **OUT**, and **IN OUT**. Parameters marked with **IN** are called by value. **OUT** parameters are called by result. Parameters which have both keywords **IN OUT** are called by value-result. If mode is not specified, **IN** is applied by default. Call-by-value (**IN**) parameters may be initialized via expression. Functions must have **IN** parameters; in other terms, functions are not allowed to change their parameters. However, a function may have side effects, because it may change its context.

Parameter evaluation features strict type equivalence. By default, parameters are bound by order; however, it is also possible to bind actual parameters by name with the syntax `formal_parameter_name => actual_parameter`. It is not necessary to pass arguments to formal parameters initialized by expressions. The subprogram will operate with the actual parameter if provided, otherwise the initial value is used.

The following source code illustrates a procedure specification:

```
procedure XY (C: in integer range 1..89; D: in integer:=0)
```

which can be called in any of the following ways:

```
XY(2,9);

XY(2);

XY(D => 9, C => 2);
```

In the first and the third case, `C=2` and `D=9`. In the second example, `C=2` and `D=0` because the latter has been initialized in the specification. The third case demonstrates binding by name, i.e. the formal parameters to which values are assigned are named explicitly.

Subprograms are recursive by default.

Blocks have the following syntax:

```
[blockname:] [DECLARE declaration_section] BEGIN executable_statements END [blockname];
```

If `blockname` is specified at the beginning of the block it must be provided after the keyword **END**, too.

The life-time of variables is dynamic by default.

Although scopes are static, Ada resolves the “hole in the scope” problem by qualifying the names of global objects with the name of the program unit to which they are local.

The following source code shows an Ada function calculating the factorial:

```
function FACT(I : integer) return real is

  F : real:=1.0;

begin
```
if I>1 then F:=FACT(I-1) * float(I);

end if;

return(F);

end FACT;

14.5. C

The C language supports functions and blocks. Functions cannot be nested within other program units. Blocks may be nested in any depth.

Blocks have the following syntax:

```
{  
  declaration_section  
  executable_statements  
}
```

Functions have the following syntax:

```
[type] name([formal_parameter_list])

block
```

If `type` is not specified, the function’s return type is `int` by default. If the return type is `void`, the function acts as a procedure. The main program is a function with the name `main`.

A function may end in one of the following ways:

- `RETURN expression`: the return value of the function is the value of the expression.
- `RETURN`: functions of the type `void` do not return a value; functions with other types will return an undefined value.
- If control reaches the closing `}` the return value of the function is undefined.

Functions are recursive by default.

The `formal_parameter_list` is a comma-separated list of names optionally with its types. It is possible to declare a function with a variable number of parameters by closing a non-empty parameter list with `...`. The empty formal parameter list can be marked explicitly with the keyword `void`.

Parameter evaluation is subject to ordinal binding and type compatibility. If the number of parameters is fixed numerical matching applies.

Actual parameters are passed by value.

The compilation unit of C is the source file, which contains external declarations (of named constants, variables, types, and functions). It is possible to import other compilation units (features to be used in the given unit) by the `#include <source_file_name>` preprocessor statement at the beginning of the file.

C employs the following storage class specifiers for controlling the scope and lifetime of programming objects:

- `extern`: The default storage class of names declared at the level of compilation units; must be indicated explicitly for local names. The scope of `extern` names spans the whole program, their lifetime lasts for the program’s run-time. `extern` variables are initialized automatically.
- **auto**: The default storage class of local names. The scoping of `auto` names is static; they are visible from their declaration. `auto` names have a dynamic lifetime, and are not initialized automatically.

- **register**: A special case of `auto` where the value is stored in the register if there is enough space, otherwise the same rules apply.

- **static**: May be applied explicitly to all kinds of names. The scope of `static` variables encompasses the compilation unit; their lifetime lasts for the program's run-time; variables are initialized automatically.

The following source code demonstrates a function calculating the factorial in C:

```c
long fact(long n)
{
    if (n<=1) return 1;
    else return n*fact(n-1);
}
```
15. fejezet

15 ABSTRACT DATA TYPES AND THE PACKAGE

An abstract data type (ADT) is a data type that implements the principles of information hiding, which means that the representation of the data and the implementation of the operations are unknown to the outside world. The information hidden within such data types can only be reached via the interface of the type, i.e., via its set of operations. Abstract data types are an essential means of secure programming in the sense that it is not possible to corrupt the values accidentally or intentionally. Abstract data types have become prevalent in the programming languages of the last few decades, and radically influenced the development of new languages.

A package is a program unit which satisfies the criteria of both procedural and data abstraction. Packages as a manifestation of procedural abstraction are the collection of the following reusable programming objects:

- type;
- variable;
- named constant;
- custom exception;
- subprogram;
- package.

Packages were employed first in Ada. An Ada package has two parts: a specification and a body.

The specification of an Ada package is of the following form:

```
PACKAGE name IS
  public_declarations
  [PRIVATE
   private_declarations]
END [name];
```

`public_declarations` constitute the visible part of the package specification. The programming objects declared in this section can be referred to from outside the package. Only the specifications of subprograms may be declared public, implementations are hidden. References must be qualified with the name of the package.

`private_declarations` cannot be accessed from outside. Objects declared here are closed by the package, and hidden from the outside world.
The body of the package is optional. However, if the package specification includes subprogram specifications then it is obligatory to provide the implementation of the subprogram in the body. The body is not visible from outside.

The body of a package is specified as follows:

```ada
PACKAGE BODY name IS

declarations

[ BEGIN executable_statements]

END [name] ;
```

Ada packages may be compiled independently, or can be included in the declaration section of another program unit. In the latter case, the visibility of the nested package is controlled by the static scoping of the containing program unit. Packages compiled independently must be made visible to other parts of the program in an explicit way (see Section 16.2).

The following example is the sketch of a package which defines operations on rational numbers.

```ada
package RATIONAL_NUMBERS is

type RATIONAL is

record

  NUMERATOR : integer;

  DENOMINATOR : integer range 1..MAX_INTEGER;

end record;

function """"= ( X,Y : RATIONAL) return boolean;

function """"+ ( X,Y : RATIONAL) return RATIONAL;

function """"- ( X,Y : RATIONAL) return RATIONAL;

function """"* ( X,Y : RATIONAL) return RATIONAL;

function """"/ ( X,Y : RATIONAL) return RATIONAL;

end;

package body RATIONAL_NUMBERS is

procedure SAME_DENOMINATOR (X,Y: in out RATIONAL) is ...

function """"= ...function """"+ ...

function """"- ...

function """"* ...

function """"/ ...
The specification of the package contains only visible (public) elements. The example package defines a private type, RATIONAL, which represents fraction values. A record type is used to implement the fraction’s representation, where the numerator and the denominator are stored in two fields. The representation also defines the range (e.g. denominators can only be positive integers). The rest of the specification of the package defines five operations. Note that Ada does not constrain function names to identifiers; built-in operators may be overloaded, as illustrated in the example. Beside the usual identifiers, special characters enclosed between double quotes may also be used as function names.

Since the specification of the package contains subprogram specifications, the package must have a body, which provides the implementation of the previously declared subprograms. The implementation of the operations must reflect the typical behavior of fractions. Private methods may be employed to extract behaviour common to several operations (e.g. SAME_DENOMINATOR).

In the example above, RATIONAL is not an abstract data type because it does not hide the details of the representation, even though the implementation of the operations is not visible. The programmer is expected to exert self-control and manipulate fractions via the operations provided with the package, instead of manipulating the values stored in the record fields directly.

Data abstraction is implemented by encapsulation in Ada, which is manifest in PRIVATE types and the hidden parts of the package specification. Objects of a type declared private in the public section of the specification support only the equality check (=) and the assignment (:=) operators by default: the programmer has to implement any other operations for the manipulation of object values. The internal representation of private types is unknown.

If the public section of the specification contains the declaration of a private type, the private section must also be present in order to specify the representation of the type.

The following code demonstrates the abstract version of the RATIONAL type; we have left the body unmodified, and have revised the specification as follows:

```ada
package RATIONAL_NUMBERS is

    type RATIONAL is private;

    function "=" ( X,Y : RATIONAL) return boolean;
    function ""+ ( X,Y : RATIONAL) return RATIONAL;
    function ""- ( X,Y : RATIONAL) return RATIONAL;
    function ""* ( X,Y : RATIONAL) return RATIONAL;
    function ""/ ( X,Y : RATIONAL) return RATIONAL;

    private

    type RATIONAL is

        record
            NUMERATOR : integer;
            DENOMINATOR : integer range 1..MAX_INTEGER;
        end record;

end RATIONAL_NUMBERS;
```

The representation of the type has been transferred to the private section of the specification to accomplish proper encapsulation. From now on the values stored in the fields may only be accessed via predefined operators and the operations implemented by the programmer.

The **LIMITED PRIVATE** type of Ada is a restricted variant of the private type, to which built-in operators (equality check and assignment) cannot be applied. The programmer is required to implement these operations too.

Should the limited private type be used, the skeleton of the example package would change as follows:

```ada
package RATIONAL_NUMBERS is
  type RATIONAL is limited private;
  procedure ASSIGN ( X : out RATIONAL; Y : RATIONAL);
  function """"= ( X,Y : RATIONAL) return boolean;
  function """"+ ( X,Y : RATIONAL) return RATIONAL;
  function """"- ( X,Y : RATIONAL) return RATIONAL;
  function """"* ( X,Y : RATIONAL) return RATIONAL;
  function """"/ ( X,Y : RATIONAL) return RATIONAL;
  private
    type RATIONAL is
      record
        NUMERATOR : integer;
        DENOMINATOR : integer range 1..MAX_INTEGER;
      end record;
  end RATIONAL_NUMBERS;

package body RATIONAL_NUMBERS is
  procedure SAME_DENOMINATOR (X,Y: in out RATIONAL) is ...
  procedure ASSIGN ( X : out RATIONAL; Y : RATIONAL)...
  function """"= ...function """"+ ...
  function """"- ...
  function """"* ...
  function """"/ ...
end RATIONAL_NUMBERS;
```
Variables declared in the public section of the package specification are of the type \texttt{OWN}, which means that these variables retain their values between two consecutive subprogram calls. This feature is a further means of communication between the subprograms.

Note that the unit of Turbo Pascal satisfies the criteria of a package. Pascal units have the following structure:

\begin{verbatim}
UNIT name;

INTERFACE

visible_declarations

IMPLEMENTATION

hidden_declarations

BEGIN

executable_statements

END.
\end{verbatim}

15.1. Questions

1. What is meant by an abstract data type?
2. What distinguishes packages from other program units?
3. In what ways do Ada packages support procedural abstraction?
4. In what ways do Ada packages support data abstraction?
5. Describe the \texttt{PRIVATE} and \texttt{LIMITED PRIVATE} data types of Ada.
16. fejezet

16 ON ADA COMPILATION

This chapter gives a brief overview of some of the most intriguing aspects of compilation in Ada. Special attention is given to pragmas and tasks.

16.1. 16.1 Pragmas

Pragmas are instructions placed in the source code which effect the compiler’s functionality. Pragmas target the compiler by invoking its services, or setting its parameters. Although pragmas are not translated into source code, they do have an impact on the workings of the code. Some of the pragmas may be placed anywhere in the program text, while others are restricted to fixed locations.

Pragmas have the following structure:

```
PRAGMA name [(parameters)];
```

The structure of a parameter is as follows:

```
[ name => ] { identifier | expression };
```

Pragmas are one of those few programming constructs which are only partially regulated by the Ada reference language. In other terms, implementations may vary. Ada systems include on average about 50 types of pragmas.

Here are a few examples of Ada pragmas to give a flavour of their syntax and purpose:

- `INTERFACE(programming_language, subprogram);`
  May be set after a subprogram specification to indicate the language in which the specific subprogram body is written.

- `LIST (( ON | OFF ))`
  Compilation may produce a listing from the program text on the standard output (ON); the feature can be disabled with (OFF). This pragma may be placed anywhere in the code.

  If the compiler does not recognize the name of a pragma, the pragma is ignored.

16.2. 16.2 Compilation Units

The following constructs are valid compilation units in Ada:

- subprogram specification;
- subprogram body;
Compilation units which do not depend on other units are called library units. The programmer may define any number of library units provided their names are unique.

In order for a particular compilation unit to use a specific object declared in another unit, the specification of that object must be in an already compiled state, i.e. compilation units must be compiled before their contents may be referred to. It follows that the main program must be written last. The Ada compiler performs a consistency check during compilation.

If the specification changes, it is necessary to recompile the compilation unit in which the specification is declared and also the units which have references to the specification. If the implementation changes only the compilation unit containing the implementation must be recompiled. The separate recompilation of specific parts facilitates developing large programs in parallel, makes the modification of programs more simple, and advances safe programming.

Every compilation unit must be preceded by a so-called context clause in the form of WITH statements. The context clause specifies the library units whose objects are referred to in the given compilation unit.

The syntax of a context clause is the following:

```
WITH library_unit [, library_unit ]...;
```

The library units specified in the WITH statements constitute the context of the compilation unit; objects specified in the referenced units are accessible to the compilation unit.

The nine standard library units of Ada make up the Ada system itself. These libraries must be specified explicitly in the context except for the one named STANDARD, which is appended automatically to all compilation units by the compiler. The STANDARD library defines the basic language objects (e.g. character set, built-in types, etc.) that are indispensible to any program.

Compilation subunits are compilation units that do not exist independently, but are attached to another compilation unit.

Ada allows the programmer to include the implementation of a subprogram, package or task within another compilation unit as a compilation subunit, rather than implementing the body next to its specification. In this case, the implementation is replaced with a stub at its “original” location. The context of the compilation subunit is determined by the stub. The compilation subunit is required to refer to the name of the program unit which contains the stub. The compilation of the unit containing the stub must always precede the compilation of the subunit. It is possible to create a chain of any number of related compilation subunits.

The keyword SEPARATE indicates the stub which replaces the body. Stubs must be referred to in the form SEPARATE(name) at the beginning of compilation units.

The following are examples of compilation units:

```
-- single compilation unit

procedure F is

package D is

    LIMIT : constant := 1000;

    TABLE : array (1..LIMIT) of integer;

    procedure RESTART;

end;

package body D is
```
procedure RESTART is 
begin 
for N in 1..LIMIT loop
     TABLE(N):=N;
end loop;
end;

begin 
RESTART;
end D;

procedure Q(X:integer) is 
begin 
...
D.TABLE(X):= D.TABLE(X)+1;
...
end Q;

begin 
...
D.RESTART;
...
end F;

The example demonstrates a procedure which contains a package and another procedure. Compilation produces a library unit called F.

The previous source code can be split into three distinct compilation units:

-- first compilation unit

package D is 
    LIMIT : constant:=1000;
    TABLE : array (1..LIMIT) of integer;
procedure RESTART;
end D;

-- second compilation unit

package body D is

procedure RESTART is

begin

  for N in 1..LIMIT loop

    TABLE(N):=N;

  end loop;

end;

begin

  RESTART;

end D;

-- third compilation unit

with D;

procedure F is

  procedure Q(X:integer) is

  begin

    ...;

    D.TABLE(X):= D.TABLE(X)+1;

    ...;

  end Q;

begin

  ...;

  D.RESTART;

  ...

...
Since the specification of the referred features must be compiled first, the first compilation unit must be compiled first, followed by the second and the third units – the order of the latter two is irrelevant.

Examples of compilation subunits:

```plaintext
-- outer subprogram

procedure T is
  type REAL is digits 10;
  R, S : REAL := 1.0;

package D is
  PI : constant := 3.14159;
  function F(X:REAL) return REAL;
  procedure G(X,Z:REAL);
end;

package body D is separate; -- stub

procedure Q(U:in out REAL) is separate; -- stub
begin
  ...
  Q(R);
  ...
  D.G(R,S);
  ...
end T;

-- compilation subunit

separate(T) -- reference to stub

procedure Q(U : in out REAL) is
begin
```
... end Q;

-- compilation subunit

separate(T) -- reference to stub

package body D is

...

function F(X:REAL) return REAL is separate; -- stub

procedure G(Y,Z : real) is separate; -- stub

...

end D;

-- compilation subunit

separate(T.D) -- reference to stub in compilation subunit

function F(X:REAL) return REAL is

...

end F;

procedure G(Y,Z:REAL) is

...

end G;

16.3. Questions

1. What is a pragma?
2. Describe compilation units.
3. What is a library unit?
4. What is a compilation subunit?
5. What are stubs used for?
17. fejezet

17 EXCEPTION HANDLING

Exception handling is a means of taking the handling of interruptions from the operating system to the level of the program. Exceptions are events that cause interruptions. Exception handling is an activity performed by the program if an exception occurs. The exception handler is the part of the program that runs after the occurrence of a given exception as a reaction to the event.

Exception handling makes event control possible at the level of programs.

Similarly to the way in which certain interruptions may be masked at the operating system level, it is also possible to mask exceptions at the language level by making the monitoring of certain exceptions disabled or enabled. Disabling the monitoring of an exception is the simplest form of exception handling: an event causes an interruption, which propagates to the level of the program raising an exception, of which the program takes no notice and continues running. The effect of the unhandled exception on the further functioning of the program is completely unknown, it is possible that the program cannot recover from the exception at all, or its operation will be corrupted.

Exceptions usually have a name (which usually plays the role of the message describing the event) and a code (an integer).

Exception handling was introduced first in PL/I and later in Ada, too. However, the two languages have vastly different principles concerning the details of the exception handling mechanism. According to PL/I, the reason why an exception was raised is that the algorithm implemented by the program has not been prepared to handle an exceptional situation, therefore it is the cause of the exceptional event that needs to be managed. The exception handler extinguishes the extraordinary situation, and returns to the normal functioning of the program. The program continues from the point where the exception was thrown.

As opposed to this, Ada claims that exceptional situations cannot be "cured"; the original activity should be abandoned. Exception handlers in Ada perform activities which are adequate substitutes to the original event, and do not return to the point where the exception was thrown.

Languages differ in the following aspects of exception handling:
1. What kind of built-in exceptions are in the language?
2. Can the programmer define custom exceptions?
3. What kind of scoping rules does the exception handler have?
4. Is it possible to bind exception handlers to programming elements (expressions, statements, or program units)?
5. How does the program continue after exception handling?
6. What happens if an exception occurs in the exception handler?
7. Are there built-in exception handlers in the language?
8. Is it possible to write a general exception handler that can handle all kinds of exceptions?
9. Can the exception handler be parameterized?

There are no parameterized and built-in exception handlers in PL/I and Ada.
17.1. Exception Handling in PL/I

PL/I provides the following built-in exceptions:

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONVERSION</td>
<td>conversion error</td>
</tr>
<tr>
<td>FIXEDOVERFLOW</td>
<td>fixed point overflow</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>floating-point overflow</td>
</tr>
<tr>
<td>UNDERFLOW</td>
<td>floating-point underflow</td>
</tr>
<tr>
<td>ZERODIVIDE</td>
<td>division by zero</td>
</tr>
<tr>
<td>SIZE</td>
<td>size error</td>
</tr>
<tr>
<td>SUBSCRIPTRANGE</td>
<td>index out of bounds</td>
</tr>
<tr>
<td>STRINGRANGE</td>
<td></td>
</tr>
<tr>
<td>STRINGSIZE</td>
<td>trace</td>
</tr>
<tr>
<td>CHECK(identifier)</td>
<td></td>
</tr>
<tr>
<td>AREA</td>
<td>addressing error</td>
</tr>
<tr>
<td>ATTENTION</td>
<td>external interrupt</td>
</tr>
<tr>
<td>FINISH</td>
<td>regular ending of the program</td>
</tr>
<tr>
<td>ENDFILE(file_name)</td>
<td>end of the file</td>
</tr>
<tr>
<td>ENDPAGE(file_name)</td>
<td>end of the page</td>
</tr>
<tr>
<td>KEY(file_name)</td>
<td>key error</td>
</tr>
<tr>
<td>NAME(file_name)</td>
<td></td>
</tr>
<tr>
<td>RECORD(file_name)</td>
<td></td>
</tr>
<tr>
<td>TRANSMIT(file_name)</td>
<td></td>
</tr>
<tr>
<td>UNDEFINEDFILE(file_name)</td>
<td></td>
</tr>
<tr>
<td>PENDING(file_name)</td>
<td></td>
</tr>
<tr>
<td>ERROR</td>
<td>general exception</td>
</tr>
</tbody>
</table>

The programmer may declare custom exceptions of the form `CONDITION(name)`. The first five built-in exceptions are monitored by default but can be disabled if required. The following five exception are disabled by default but can be enabled. The rest of the built-in exceptions and all programmer-defined exceptions are always enabled and cannot be disabled.

Every statement can be preceded by a rule that specifies whether the monitoring of exceptions possibly thrown by the statement should be disabled or enabled. Any number of exceptions separated by commas can be enclosed in round brackets

```plaintext
(exception_name [, exception_name...] ): statement
```

to indicate that the given exceptions should be monitored.

To disable the monitoring of an exception, the keyword `NO` must precede the name of the exception, as in the following example:

```plaintext
(NOZERODIVIDE, SIZE): IF ...
```

If the monitoring rule is placed before the first statement of a block or a subprogram its scope spans the entire program unit including the contained program units as well. It is possible to override the rule by individual statements of the program unit. If the rule precedes a statement that contains an expression, the rule applies to the expression alone rather than the entire statement. If the statement contains no expressions, the rule applies to the entire statement.

The statement `SIGNAL exception_name;` is used to throw an exception explicitly. This is the only way of throwing programmer-defined exceptions.

Exception handlers in PL/I have the following syntax:

```plaintext
ON exception_name executable_statement;
```

where blocks may stand in for `executable_statement`.

Exception handlers can be placed anywhere in the source code.

The scope of an exception handler lasts from the moment the control reaches it until:
• control reaches another exception handler intended to handle the same exception, which overrides the effect of the previous handler;

• control reaches the statement `REVERT exception_name;` which undoes the effect of the last ON statement;

• or the termination of the program unit

including every program unit called from inside the scope of the exception handler.

It follows that exception handlers have dynamic scoping.

If an exception is raised in a programming unit, the run-time system examines whether the monitoring of the given exception is enabled or not. If monitoring is disabled, the execution of the program unit continues. If monitoring is enabled, the run-time system examines the availability of an exception handler which contains the name of the given exception. If such an exception handler is found, the exception handler is executed. If the exception handler contains a `GOTO` statement, the program continues on the statement with the given label. If there is no `GOTO` statement, control returns either to the statement in which the exception occurred or to the following statement, depending on the nature of the exception. For example, in the case of a `CONVERSI-ON` exception, the same statement which caused the error will be executed again; in the case of arithmetical errors and custom exceptions, the program continues with the statement immediately following the one which threw the exception.

If a proper named exception handler is not available in the given program unit, control steps back on the call chain and continues looking for appropriate handlers in the caller. If none of the calling units contain an appropriate handler, the exception `ERROR` is raised. Now it is the `ERROR` which needs to be handled. The `ON ERROR` exception handler is used to manage all sorts of unnamed exceptions; it is the general exception handler of PL/I. If no general exception handler is specified, control returns to the operating system. The program failed to handle the given exception.

To assist the programmer in handling exceptions, PL/I provides special built-in functions which can be called only from exception handlers. The `ON` functions have no parameters, and are used to specify the event, place or cause of exceptions.

The following are examples of exception handler functions:

- `ONCODE`: Returns the unique code of the error; useful for handling built-in exceptions (e.g. KEY) which name event groups, in which case the individual events can only be identified via their code.

- `ONCHAR`: If a conversion error is caught, the function returns the character which caused the error during data transfer. Since the function behaves as a pseudo variable (i.e. it can be assigned a value), it is possible to replace the troublesome character; as a result, the I/O operation may be repeated.

- `ONKEY`: Returns the primary key of the record which caused an I/O error.

- `ONLOCK`: Returns the name of the subprogram in which the exception was raised.

The dynamic scoping of exception handlers sometimes cause problems. While names have static scoping, exception handlers support dynamic scoping, which may lead to conflicts. The called program unit inherits the effects of the exception handler activated in the caller program unit. This is considered dangerous, because it may cause non-local jumps. If the exception is not handled by the same program unit in which it occurs, another exception handler may be activated which is located considerably earlier in the call chain, and this conduct may be entirely wrong.

Although programmer-defined exceptions are of a great benefit for testing purposes, they are not considered effective during run-time.

Example:

In PL/I, sequential files may be processed in the following way:

```
DECLARE F FILE;

1 S,

2 ID PICTURE ''9999,

3 OTHERS CHARACTER(91),

EOF BIT(1) INIT''(0B);
```
ON ENDFILE(F) EOF'=-1B;
OPEN FILE(F);
READ FILE(F) INTO(S);
DO WHILE¬(EOF);
...
READ FILE(F) INTO(S);
END;

17.2. 17.2 Exception Handling in Ada

The built-in exceptions of Ada generally name groups of events. These are the following:
- CONSTRAINT_ERROR: The event group that occurs if a declaration constraint is violated, for example, an index bound is exceeded.
- NUMERIC_ERROR: Arithmetic errors, including underflow and overflow errors, division by zero, etc.
- STORAGE_ERROR: Memory errors (including all allocation-related problems): the memory region referred to is not available.
- TASKING_ERROR: Rendezvous is not possible with the given task.
- SELECT_ERROR: SELECT statement error.

Every exception is monitored by default, but the monitoring of certain events (especially checks) can be disabled with the pragma SUPPRESS:

SUPPRESS(name [,ON => { object_name | type }] )

where name is the name of the event to be disabled; identifies a single event, and does not correspond to built-in exception names. A name can take the following values:
- ACCESS_CHECK: checking address;
- DISCRIMINANT_CHECK: checking the discriminant of a record;
- INDEX_CHECK: index checking;
- LENGTH_CHECK: length checking;
- RANGE_CHECK: range checking;
- DIVISION_CHECK: checking division by zero;
- OVERFLOW_CHECK: overflow checking;
- STORAGE_CHECK: checking the availability of storage.

The object_name identifies a programming object (e.g. a variable). If the optional part is not specified, monitoring the event is disabled in the entire program. If, however, the optional part is present, monitoring is disabled only on objects of the type or of the object_name specified.

Custom exceptions can be declared with the EXCEPTION attribute.

Exception handlers can be placed after the body of every program unit, right before the closing END. Exception handlers have the following syntax:
The statements part comprises any number and kind of executable statements. An exception handler may include any number of WHEN branches, but at least one is mandatory. The WHEN OTHERS branch may occur only once, and it must be the last branch. The WHEN OTHERS branch is used to handle unnamed exceptions (i.e. it is the general exception handler of Ada).

The exception handler is accessible to the entire program unit and those program units which have been called from it, unless they specify their own exception handlers. In Ada, exception handlers have dynamic scoping, which is inherited via the call chain.

The statement

```
RAISE exception_name;
```

is used to throw exceptions of any type. Programmer-defined exceptions can only be raised in this way.

If an exception is raised in a program unit, the run-time system first examines whether the monitoring of the given exception is disabled or not. If monitoring is disabled, the execution of the program continues; otherwise, the program unit terminates. Then the run-time system examines the availability of exception handlers within the program unit. If an exception handler is found, the run-time system examines if the handler contains a WHEN branch which names the given exception. If one of the branches satisfies this condition, the run-time system executes the statements contained therein. If the statements include a GOTO statement the program continues on the statement with the given label; otherwise the program continues as if the program unit terminated regularly. If none of the branches match the name of the exception the run-time system examines the availability of a WHEN OTHERS branch. If the WHEN OTHERS branch is specified, the statements contained therein are executed subject to the same rules as in the previous case. If none of branches match and no WHEN OTHERS branch is specified or the unit does not contain an exception handler at all, the program unit propagates the exception. This means that the exception is raised by the program unit call, and the process of looking for an appropriate handler is repeated. The run-time system keeps searching for a proper exception handler up the call chain. If control reaches the beginning of the call chain without having found a proper exception handler the program does not handle the exception and control is transferred to the operating system.

Exceptions thrown in the exception handler are propagated immediately.

The RAISE; statement can be used to propagate the exception only in exception handlers.

Exceptions thrown in declaration statements are propagated immediately.

Exceptions thrown in nested packages are propagated to the containing unit, while exceptions thrown in compilation-level packages abort the main program.

The dynamic scopes of exception handlers raise the same problems as in PL/I, the only difference being that the program units on the call chain terminate regularly. The Ada compiler cannot check the operation of exception handlers.

Custom exceptions have an essential role in Ada programs as they provide a means of communication between program units via event control.

Example:

```
function FACT(N : natural) return float is

begin

if N=1 then return 1.0;

else return float(N)*FACT(N-1);

end function FACT;
```
end if;

exception

when NUMERIC_ERROR => return FLOAT_MAX;

end;

If the function is called with a value too large to calculate its factorial, an overflow error occurs, which is caught by the NUMERIC_ERROR exception handler such that the function returns the largest floating-point value.

### 17.3. Questions

1. What is an exception? What is meant by exception handlers?
2. In what respect may languages differ about exception handling?
3. Describe the exception handling mechanism of PL/I.
4. Describe the exception handling mechanism of Ada.
18. fejezet

18 GENERIC PROGRAMMING

Generic programming is a means of reusability and thus a cornerstone of procedural abstraction. Generic programming is orthogonal to all other paradigms; therefore its feature system can easily be built into all sorts of programming languages. The essence of generic programming is that it enables the parameterization of the source code. The generic text (i.e. the one which contains parameters) is managed by the compiler. Actual parameters are used to generate the concrete text which may then be compiled. Code reuse lies in the fact that any number of concrete texts can be generated from a single generic text. What is more, generic texts may contain type parameters.

The rest of the chapter illustrates the generics of Ada.

In Ada, generic unit is of the following form:

```
GENERIC formal_parameter_list
body
```

where body stands for the declaration of an entire subprogram or a package in which the formal parameters occur.

A generic unit may accept variables, types and subprogram specifications as a parameter. The syntax of the formal_parameter_list is the following:

```
[[ declaration_of_variable |

TYPE name IS {(<>) | RANGE <> | DELTA <> | DIGITS <> | arraytype | pointertype | [ LIMITED] ← PRIVATE } |

WITH subprogram_specification [IS name ] |

WITH subprogram_specification IS <> ]; ...]
```

The following statement is used to generate a concrete subprogram or package from the generic text:

```
{ PROCEDURE | FUNCTION | PACKAGE } new_name IS NEW generic_name [{actual_parameter_list}];
```

This is how generics are “called”. The generation of concrete code involves parameter evaluation and parameter passing.

The number of generic formal parameters is always fixed. By default, parameters are bound by order, but named binding may also be used. It is not necessary to pass actual parameters to the formal parameters of subprogram specifications if the invocation statement includes the keyword IS. Variables accept constant expressions of the same type as actual parameters, while subprogram specifications take procedure or function names with the appropriate specification. Type parameters may be assigned the name of an enumeration, integer, fix-point, floating point, array, pointer, or an arbitrary type (in the order indicated by the syntax of formal parameter list).

Parameters are passed by value for variables, and passed by name for type names. If a subprogram specification is assigned an actual subprogram name, the name of the actual subprogram replaces the name of the subprogram parameter in the generated
text. If the name of the actual subprogram is not specified, it is the name declared after the keyword IS that replaces the generic parameter. In the case of IS<> the generated name is the same as the name of the formal parameter.

The following example illustrates a generic package that implements the abstract data type stack.

generic

    SIZE : integer;

type ELEMENT is private;

package STACKS is

    type STACK is limited private;

    procedure PUSH(S:in out STACK; E:in ELEMENT);

    procedure POP(S:in out STACK; E:out ELEMENT);

    OVERFLOW, UNDERFLOW : exception;

private

    type STACK is

        record

            AREA:array(1..SIZE) of ELEMENT;

            INDEX:integer range 0..SIZE:=0;

        end record;

end;

package body STACKS is

    procedure PUSH(S:in out STACK; E:in ELEMENT) is

        begin

            if S.INDEX=SIZE then raise OVERFLOW; end if;

            S.INDEX:= S.INDEX+1;

            S.AREA(S.INDEX):=E;

        end PUSH;

    procedure POP(S:in out STACK; E:out ELEMENT) is

        begin

            if S.INDEX=0 then raise UNDERFLOW; end if;

            E:=S.AREA(S.INDEX);

            S.INDEX:= S.INDEX-1;

        end POP;

end;
if S.INDEX=0 then raise UNDERFLOW; end if;

E:=S.AREA(S.INDEX);

S.INDEX:=S.INDEX-1;

end POP;

end STACKS;

The generic unit has two formal parameters: SIZE determines the maximal size of the stack, while ELEMENT defines the type of the elements stored on the stack. Since ELEMENT is of a limited private type it may accept all kinds of type names as actual parameter. The generic implementation of the package allows the generation of stacks with an arbitrary number of elements of any type.

Stacks may be represented as one-dimensional arrays. The last-in-first-out (LIFO) behavior requires the implementation of two operations (PUSH and POP). Extraordinary situations (the stack is empty, or the stack is full) are handled by throwing custom exceptions. Since there are no exception handlers in the package itself the exceptions are propagated to the calling environment.

The following two examples show how to generate concrete stack packages:

package INTEGER_STACK is new STACKS(ELEMENT=>integer, SIZE=>1024);

package LOGICAL_STACK is new STACKS(100, boolean);

The first line illustrates named binding, while the second shows parameters bound by order.

18.1. Questions

1. What is meant by generic programming?
2. Describe generic programming in Ada.
19. fejezet

19 PARALLEL PROGRAMMING AND THE TASK

Computers built using the von Neumann architecture are sequential: the processor executes statements broken down into atomic steps in the order prescribed by the program.

The machine code currently executed by the processor is called a process or a thread. The difference between processes and threads is that processes own the resources exclusively, while threads share the resources they use.

Parallel programming is based on the following concepts:

Communication: Processes communicate with each other, they exchange data.

Synchronization: Parallel processes must meet in certain moments. Data exchange and communication takes place at synchronization points. For example, a process may be waiting for information to be produced by another process without which it cannot continue its operation.

Concurrency: Processes compete for resources.

Mutual exclusion: Since the processes use resources exclusively, it must be ensured that only one process is allowed to modify a resource at a time, and other processes are excluded.

Parallel programming features were first introduced in PL/I. Both Pascal and C have versions with features for parallel programming.

In order to support the writing of parallel programs, languages must have features for:

- setting process codes,
- launching and ending processes,
- requesting mutual exclusion,
- synchronization,
- the communication of processes,
- suspending the operation of a process,
- prioritizing processes, and
- scheduling processes.

19.1 Ada Tasks

In Ada, a task is a program unit which is used in parallel programming. Tasks may not exist independently; tasks must be embedded within other program units. The program unit which contains the task is called the parent unit. Any number of sibling
tasks (i.e. tasks declared at the same level in terms of program structure) may occur within the same parent task. Tasks can be embedded into each other in any depth. The processes which back the parent unit and the sibling tasks are executed in parallel.

With multiprocessor systems, it is feasible that each task is executed on a different processor, which is a manifestation of true parallelism. Single processor systems are also suitable for executing programs with parallel structures; however, in this case it is the operating system that simulates parallelism—this is virtual parallelism.

A task begins its operation when its parent unit starts (initial synchronization takes place). It follows that in Ada, the structure of the program determines the schedule of the processes; practically, scheduling is controlled by the programmer.

A task terminates its operation if one of the following conditions holds:

- all of its statements have been executed,
- the task is terminated by its parent unit or one of its sibling tasks with the statement \texttt{ABORT name},
- the task terminates itself explicitly with a dedicated statement.

The parent unit terminates if (1) it finishes its operation as a program unit, and (2) all the sibling tasks it owns end. This is a form of end-synchronization from the perspective of the parent unit.

Formally, a task consist of two parts, a specification and a body:

\begin{verbatim}
TASK [TYPE] name

[ IS entry_declarations
END [name] ];

TASK BODY name IS

[ declarations ]
BEGIN

executable_statements

[ exception_handler ]
END [name];
\end{verbatim}

The specification part is made up of entry specifications, which declare entry points. Formally, entry specifications are identical to procedure specifications, the only difference being that they are introduced by the keyword \texttt{ENTRY} instead of the keyword \texttt{PROCEDURE}. Entry points are a means of synchronisation in Ada. The type \texttt{TASK} is considered a limited private type.

Based on their purpose, tasks are of two kinds: passive and active. Passive tasks provide services and contain entry-specifications which define these services. Active tasks consume the services provided by passive tasks.

In Ada, synchronization is called \textit{rendezvous}. An active task creates a rendezvous point by calling an entry, which formally corresponds to procedure calls.

A passive task is expected to specify at least one \textit{accepting} statement for each of its entry specifications. Accepting statements are of the following form:

\begin{verbatim}
ACCEPT entry_name[(formal_parameter_list)]

[ DO executable_statements END [entry_name] ];
\end{verbatim}
This is how the passive task creates a rendezvous point. The services offered by the passive task are enclosed between the keywords DO and END.

The rendezvous of an active and a passive task follows a fixed protocol. The active task initiates the rendezvous by calling an entry defined by the passive task, which accepts the invitation via the appropriate accepting statement. The two tasks start their operation and execute statements sequentially until one of the tasks reaches a rendezvous point. The task which reaches the rendezvous point first waits for the other task, and suspends its operation in the meantime. Once the other task arrives at the rendezvous point, information is transmitted from the active task to the passive via the IN and IN OUT parameters. Then the DO–END section is executed if present. At the end of the rendezvous, information moves from the passive task to the active via the OUT and IN OUT parameters. Consequently, a rendezvous always entails synchronisation, but does not necessitate communication or common activity.

Although the primary aim of rendezvous is to serve synchronization, they also allow for data exchange via the formal parameters of the entry points.

Tasks may also communicate through variables declared in the parent unit, since their names are global to sibling tasks. The sibling tasks are allowed to use the shared variables at the same time. Mutual exclusion is achieved with the help of the pragma SHARED(variable)

The SHARED pragma must be specified at the declaration of the variable in the parent unit.

Another pragma pertaining to tasks helps define the priority of the tasks, and is included in the specification of the task:

PRIORITY(expression)

Tasks with lower priority or no priority at all may not hinder the operation of tasks with higher priority.

If a service offered by a passive task is called by more than one active task at the same time the active tasks line up in a priority queue.

The statement DELAY expression; may be used in the body of tasks to suspend operation for the number of seconds specified by the expression, which evaluates to a decimal integer. A DELAY statement with a negative value is equivalent to a DELAY statement with a zero value.

The occurrence of a rendezvous can be influenced with the SELECT statement from within the tasks. However, SELECT statements play a different role in active and passive tasks: whereas active tasks initiate rendezvous by calling an entry point themselves, passive tasks may never know for certain whether the services they offer will ever be utilized by an active task.

Active tasks may utilize two forms of the SELECT statements, one for conditional rendezvous, another for timed rendezvous.

The SELECT statement for conditional rendezvous is the following:

SELECT
  entry_call
  [ executable_statements ]
ELSE
  executable_statements
END SELECT;

If nothing prevents the rendezvous initiated by the entry_call then the rendezvous takes place immediately, possibly followed by the execution of other statements, after which the task exits from the SELECT statement. If the immediate rendezvous is not possible then instead of waiting the active task executes “supplemental activity” specified by the statements of the ELSE branch. Finally, the task exits from the SELECT statement.

The SELECT statement for timed rendezvous is the following:
SELECT

entry_call

[executable_statements]

ELSE

delay_statement

[executable_statements]

END SELECT;

If nothing prevents the rendezvous initiated by the `entry_call` then the rendezvous takes place immediately, possibly followed by the execution of other statements, after which the task exits from the `SELECT` statement. If immediate rendezvous is not possible then the active task waits for the time specified in the delay statement. In the meantime, the task makes repeated attempts to initiate the rendezvous, and executes the “supplemental activity” of the `ELSE` branch only when the delay ends. Finally, the task exits from the `SELECT` statement.

`SELECT` used in passive tasks are of the following form:

```
SELECT

[ WHEN condition => ] alternative

[ OR [WHEN condition => ] alternative ...]

[ ELSE executable_statements]

END SELECT;
```

where an alternative may be any of the following:

- accepting alternative:
  ```
  accept_statement [executable_statements]
  ```

- delaying alternative:
  ```
  delay_statement [executable_statements]
  ```

- terminating statement:
  ```
  TERMINATE;
  ```

At least one accepting alternative must be specified; there is no upper limit on the number of accepting alternatives. A passive `SELECT` may specify any number of delaying alternatives; however, only one terminating alternative is allowed. Delaying and terminating alternatives are mutually exclusive.

An alternative is open if it is not preceded by a `WHEN` condition, or if it is preceded by a condition which evaluates to true. Otherwise the alternative is closed.

When the passive task reaches a `SELECT` statement

- The conditions are evaluated in order to determine which alternatives are open and which are closed;

- For those open alternatives that contain a `DELAY` statement the delaying expressions are evaluated in order to determine the waiting times;
- An open accepting alternative is chosen if an active task exists which wants to rendezvous with this point (the active task has called the current entry). In this case, the rendezvous takes place, other possible statements are executed, and the task exits from the SELECT statement. If more than one accepting alternative is suitable any of the alternatives may be executed; the choice is not deterministic. However, only one rendezvous may be executed.

- An open delaying alternative is chosen if none of the accepting alternatives are suitable. If a choice has to made between more delaying alternatives the one with the minimal waiting time is chosen. In this case, the passive task waits for the given time period, and listens for incoming rendezvous requests which may be matched with open accepting alternatives. If such a request is noticed the rendezvous takes place; otherwise the task executes other statements if specified after the waiting time has elapsed, and then exits from the SELECT statement.

- The open terminating alternative is chosen if the sibling tasks, the parent unit, and all the tasks contained by the given task have finished their operation. In this case the passive task terminates its operation.

- If none of the open alternatives are suitable but an ELSE branch is specified the statements contained therein are executed and the task exits from the SELECT statement. If there is no ELSE branch the task waits until an open alternative becomes suitable or a closed alternative becomes open and suitable.

- If all the alternatives are closed but an ELSE branch is specified the statements contained therein are executed and the task exits from the SELECT statement. If there is no ELSE branch a SELECT_ERROR exception is raised.

Active tasks may initiate a rendezvous only with passive tasks which still operate. If an active task initiates a rendezvous with another task that is not available (e.g. it has stopped) a TASKING_ERROR exception is raised.

Beside the ordinary exception handling mechanism provided by Ada additional rules apply to tasks. Exceptions raised in a rendezvous effect both participants of the rendezvous. Ada provides a few task-specific exceptions which can only occur in tasks, therefore it makes sense to handle such exceptions only within the tasks; in other terms, task-specific exceptions are not propagated.

Example

The rest of the chapter demonstrates a producer-consumer pattern in Ada. Two processes are involved, one which creates characters at its own pace, and another which processes these characters at a different pace. The processes operate in parallel, but are not synchronized.

We are going to write three tasks: an active producer task, an active consumer task, and a passive task used for receiving, storing and dispatching the characters.

First, we define the parent unit whose only task is the synchronisation of the producer and the consumer. The following block will play the role of the parent:

```
begin
  -- tasks null;
end;
```

The body of the producer and the consumer tasks contains an infinite loop:

```
loop
  -- produce the next character
  BUFFER.WRITE(CHAR);
  exit when CHAR=END_OF_CHAR;
end loop;
```
loop

BUFFER.READ(CHAR);

-- consume the next character

exit when CHAR=END_OF_CHAR;

end loop;

The passive task stores the data (i.e. the characters) in a cyclic queue:

task BUFFER is

entry READ(C : out character);

entry WRITE(C : in character);

end;

task body BUFFER is

POOL_SIZE: constant integer:=100;

POOL : array (1..POOL_SIZE) of character;

COUNT : integer range 0..POOL_SIZE:=0;

IN_INDEX, OUT_INDEX : integer range 1..POOL_SIZE:=1;

begin

loop

select

when COUNT < POOL_SIZE =>

accept WRITE (C : in character) do

POOL(IN_INDEX:=C); end;

IN_INDEX:=IN_INDEX mod POOL_SIZE+1;

COUNT:=COUNT+1;

or when COUNT > 0 =>

accept READ(C : out character) do

C:=POOL(OUT_INDEX); end;
The body of the passive task contains an infinite loop with one `SELECT` statement. The role of the `SELECT` statement is to synchronise the active tasks. The `SELECT` statement provides three alternatives. The terminating alternative is always open.

First, the parent unit starts and finishes its operation immediately waiting for the tasks it contains to end. The three tasks start their infinite loops. The passive task accepts a rendezvous with the producer task first, from which it receives a character that it stores in its pool. The passive task exits from the `SELECT` statement. Since the loop is infinite control returns to the `SELECT` statement immediately. As the character pool managed by the passive task is not empty the second alternative becomes open and the passive task may accept a rendezvous with either from the producer or the consumer task. During the rendezvous with the consumer the passive task dispatches the first character to the consumer stored in the pool. After a time the two active tasks exit from their infinite loops with the `EXIT` statement, after which they reach the end of their body and finish their operation. As the parent unit is in a waiting state the terminating alternative is chosen and all the processes terminate.

19.2. Questions

1. Explain the fundamental concepts of parallel programming.
2. What features may languages provide for parallel programming?
3. What is the difference between true and virtual parallelism?
4. Describe the structure of an Ada task.
5. What is the difference between active and passive tasks?
6. Describe the operation of a task.
7. How does a rendezvous take place?
8. What are the specifics of conditional rendezvous?
9. What are the specifics of timed rendezvous?
10. What is the difference between open and closed alternatives?
11. Describe the role of the passive task in a rendezvous.
20. fejezet

20 INPUT/OUTPUT

The management of input-output (or I/O for short) is the area where programming languages differ mostly. I/O operations are platform-, operating system- and implementation-dependent. Certain programming languages do not even specify I/O features, delegating the issue entirely to the implementation.

The I/O is responsible for the communication with the peripherals by sending data from the memory to the peripherals and vice versa. I/O tools are centred on the concept of the file, which corresponds to the notion of the abstract file. From the point of view of I/O tools, it is important to distinguish between logical files and physical files. A logical file is a programming feature which has a name and bears the characteristics (record format, access, structure, blocking, and record identifier) of abstract files in the form of attributes. A physical file is the ordinary file which exists at the level of the operating system. Physical files are concrete, may be presented on peripherals and contain data.

Based on their function, files fall into the following categories:

- **input**: must exist before processed; remains invariant during processing; a read-only file.
- **output**: if it does not exist before processing, it is created automatically; a write-only file.
- **input-output**: usually exists both before and after processing, but its content may change; both readable and writeable.

I/O operations move data between the memory and the peripherals. Both the memory and the peripherals have their preferred data representation format, which means that sometimes it is necessary to translate between the two formats. Based on whether the movement of data involves conversion or not two modes of data transfer must be distinguished: continuous (with conversion) and binary or record mode (without conversion).

The continuous mode is required if the representation formats of the memory and the peripherals differ. In the continuous mode, languages regard the data on the peripherals as a continuous character string, while the memory stores bit strings in compliance with the representation of the type. Data transfer means the transfer of individual data by conversion. When reading data, it is necessary to specify how the continuous character string can be split into character groups which represent pieces of data, and also the mapping between the character groups and the data types. When sending data to the peripherals (i.e. writing), the mapping of in-memory data to a continuous string needs to be defined, including the number of characters required to represent the data and their location within the string.

Languages provide the following three kinds of mapping:

- **formatted data transfer**: the type of data and the number of characters must be specified explicitly for each individual piece of data.
- **edited data transfer**: each piece of data is masked, where the mask consists of editor characters and transferable characters. The number of mask elements determines the number of characters to handle; editor characters specify which character category is expected at the given position; the remaining characters are transferred without change.
- **listed data transfer**: it is the continuous character string which contains special delimiting characters that separate the individually meaningful pieces of data. Types are not marked explicitly.

In the case of binary data transfer, the representation format of the data stored in the memory and on the peripherals is the same. This is only possible when communicating with mass storages. Data are transferred by record.

Working with files in a program involves the following steps:
1. **Declaration**: It is always mandatory to declare logical files in the program with the appropriate name and attributes, according to the rules of the given language. Languages specify the file types they support. Certain languages claim that the function of the file is also an attribute, therefore it must be decided at the declaration.

2. **Assignment**: A physical file is assigned to the logical file. The assignment is controlled either from within the source code with the help of specific language features, or mapping is performed by the operating system. Once the files are properly mapped, all the operations directed at the name of the logical file are performed on the corresponding physical file.

3. **Opening of the file**: In order to use a file it must be opened first. On opening a file, operating system routines check whether the attributes of the logical file are in line with the characteristics of the physical file. In certain languages, the function of the file is decided on opening the file (e.g. “opening for input”). A file may be opened several times for different purposes during the execution of a single program.

4. **Processing**: After the file has been opened, it can be read or written as required. In order to read a file it is necessary to provide the name of the logical file and an arbitrary variable list if data transfer is continuous. The variables get their value components from the file. Each variable must be accompanied by a format in the case of formatted mode, or a mask in the case of edited mode. In the listed mode, conversion is determined by the type of the variables. Binary data transfer may be controlled by a single (rarely more) variable of a record-type. In order to write a file (or print to the file) the name of the file must be specified as well as an expression list whose value will be written into the file. The expressions must be accompanied by a format or a mask depending on the mode of continuous transfer. Listed mode is based on the type of the expressions. In the case of binary data transfer the expression must evaluate to a value of a record type.

5. **Closing**: Closing a file invokes operating system routines, which update the contents of directories. Output and input-output files must be closed, input files should be closed. Closing a file terminates the connection between the logical file and the physical file. Most languages are of the opinion that upon the regular termination of a program all the open files are closed automatically.

High-level programming languages allow the programmer to think about I/O operations in terms of the peripherals instead of files; this is the concept of **implicit files**. The logical file and the physical file still exist, but they are managed automatically by the run-time system such that their details are invisible to the programmer. Implicit files need not be declared, assigned, opened or closed. The implicit input file is the standard system-input peripheral unit (generally the keyboard). The implicit output file is the standard system-output peripheral unit (generally the monitor). The programmer may perform any of the file-related operations explicitly (e.g. assign the printer to the implicit output file). However, if the programmer does not specify the name of the logical file during writing or reading operations apply to the implicit file.

### 20.1 I/O Features of Languages

**FORTRAN**

Supports serial, sequential and direct access file structures. Its features system is poor. Manages only the fixed record format. Earlier versions introduce formatted data transfer; listed transfer is added in later versions. Binary data transfer is performed by data rather than by record.

**COBOL**

The feature-rich I/O system of COBOL is characterized by continuous conversion, and a support for serial, sequential, direct access, indexed and inverted file structures; however, only one secondary key is used for lookup at a time. COBOL is the first language to introduce edited data transfer. Prefers the fixed record format. Blocking is possible.

**PL/I**

The file management support of PL/I is highly outstanding. The language supports all known file structures, data transfer types, record formats, and blocking possibilities.

**Pascal**

Pascal has a weak file management support. The language manages serial files with both fixed and variable record formats. Implementations support both binary and continuous transfer (the latter one is a special combination of formatted and listed data transfer), and may employ direct access. Blocks are not supported. The language does not define any I/O statements, integrated subprograms are used instead.

**C**
I/O operations are not part of the language, standard library functions are used instead. Supports both binary and continuous transfer, the latter one being a mixture of formatted and edited transfer. Serial structure is managed with fix and variable record formats. I/O functions implement the writing and reading of single characters, groups of characters, bytes, and groups of bytes.

Ada

Supports the management of every peripheral directly from the language. I/O is not part of the Ada language, the operations are defined in packages. The following packages define I/O operations:
- SEQUENTIAL_IO: For managing sequential files; works with indexed types as well.
- DIRECT_IO: For direct access files.
- TEXT_IO: For text files.
- LOW_LEVEL_IO: Bridges the communication gap between language constructs and the peripherals.
- IO_EXCEPTIONS: For managing I/O exceptions.

Once the logical file declaration is mapped onto a physical file, the function of the file may be changed dynamically during run-time, for example if an exception occurs. On closing the file, it is possible to specify whether the file should be retained or removed.

20.2. Questions

1. What is meant by a logical file?
2. Classify files according to function.
3. What is the difference between continuous and binary data transfer?
4. How can the different representation formats be mapped with continuous transfer?
5. Describe the general process of managing files from programs.
6. Describe the I/O system of a programming language of your choice.
21. fejezet

21 MEMORY MANAGEMENT IN IMPERATIVE LANGUAGES

Most imperative languages divide the available program memory into the following areas:

- **Static area**: contains the code segment and run-time system routines.
- **System stack**: stores the activation records.
- **Dynamic area (heap)**: contains the constructs manages through pointers.

The code segment contains the compiled machine-language statements of the program, system information, and the tables of literals.

Imperative languages use the so-called *activation record* for the management of run-time program units, and the implementation of the call stack. An activation record comprises the following elements:

- **Dynamic pointer**: a field of pointer type that addresses the activation record of the caller program unit. Dynamic pointers are employed upon the successful completion of a program unit, when the relevant activation record is deleted.
- **Static pointer**: a field of pointer type that addresses the activation record of the containing program unit. The static pointer provides access to the container environment if static scoping are used.
- **Return address**: Addresses the part of the code segment where the program must continue on regular completion.
- **Local variables**
- **Formal parameters** (only in the case of subprograms)
- **Return value** (only in the case of functions)

Local variables of a simple scalar type are allocated the amount of memory required by the internal representation of their specific type (fixed point, floating point, logical, character, address, enumeration). Memory allocation is a more complicated process for composite types. A popular method of allocating memory for arrays is to place an array descriptor at the beginning of the reserved memory area; the descriptor provides information on both the lower and upper index bounds for each dimension, the type of the elements, and also the number of bytes required for storing one element. The array descriptor is followed by the array elements in raw-major or column-major representation. The representation of a record type is determined by the type of the record’s fields, which are positioned one after the other. Records of variable length are allocated the largest amount of memory they may possibly occupy. Strings may be stored at fixed length or variable length; in the latter case, the length is indicated by a length descriptor or with an end of string sign. Small sets (i.e. which contain few elements) are represented with a characteristic function, while large sets are frequently mapped onto hash tables.

The allocation of memory for formal parameters depends on the parameter passing mode. Call by value parameters are allocated the amount of memory required by the type of the parameter. Call by reference and call by result parameters are reserved only
a few bytes appropriate for storing a memory address. Call by value-and-result parameters combine the allocation techniques of the two modes. Call by name and call by need parameters invoke a system routine which has no parameters; this routine is executed for each occurrence of the formal parameter references. The routine determines the context and evaluates the actual parameter by overwriting the names of the formal parameters.

Activation records are stored on the stack. The activation record of the main program is always positioned at the bottom of the stack. On regular program completion the stack is emptied. Whenever a subprogram or block is invoked, an activation record is constructed and placed on the top of the stack. (The specific program unit is considered active.) It is always the currently running program unit whose activation record is on top of the stack. On regular program completion the activation record is deleted.

In the case of tasks, a "cactus" stack is constructed (on single processor). The activation record of the parent unit is set on the top of the stack, and the activation records of each non-task program unit invoked are placed on top of that. Each sibling task has a separate stack with the activation record of the parent unit at its bottom. These stacks exist simultaneously and contain the activation records of the chain of invocation generated by the specific task. The activation record of the parent unit can be deleted only when the stacks of all the sibling tasks are empty.

21.1. Questions

1. How do imperative languages manage memory?
2. Describe the structure of the activation record.
3. Describe how formal parameters are managed.
4. Describe how local variables are managed.
22. fejezet

22 OBJECT-ORIENTED PARADIGM

The object-oriented (OO) paradigm focuses on increasing the abstraction level of programming languages, as a result of which modeling becomes easier and simpler, the real world can be better described, and problems can be solved more efficiently.

According to the object-oriented approach, data and functional models are inseparable; the two models cannot be managed in isolation. The real world has to be described with a single unified model, in which static (data) and dynamic (behavior) features are managed together; this is the principle of encapsulation. The encapsulation mechanism enables the programmer to group the data and their operations together in one place, and hide the irrelevant details from the users.

The object-oriented paradigm is based on the notion of abstract data types. The most important feature of object-oriented languages is the class; classes implement abstract data types. Classes are an abstract language construct. Implementations of object-oriented languages are often simple collections of classes.

Classes may have attributes and methods. Attributes help the programmer describe data structures of arbitrary complexity, while methods encode behavior. Methods are the object-oriented counterparts of the subprograms of procedure-oriented languages.

Another fundamental OO programming construct is the object, which is a concrete language feature. An object is always generated as an instance of a class through the process of instantiation. All instances of a specific class bear the same data structure and have the same behavior.

Every object has an address. It is the memory space where the elements of data structures are placed. The value group located at the specific address is the state of the object. As a result of the instantiation process the object comes to its initial state.

In the object-oriented paradigm, objects exist in parallel and are in interaction with each other. Objects communicate by sending messages. The object’s class determines the interface which defines what instance attributes and method specifications are visible to other objects. (Visibility features will be explained later.) An object sends a message to another (usually by invoking one of its visible methods), and the other object replies with the return value of the method or with an output parameter. The object may change its state as a result of the message.

Objects are self-conscious; every object is identical only with itself, and differs from every other object. Objects have a unique object identifier (OID) generated by language-specific mechanisms.

Attributes and methods may be of class or instance level. Instance level attributes are placed into the memory during instantiation, and their values comprise the state of the instance. Class level attributes are bound to the class, they do not replicate. An example of such an attribute may be the extent of the class, which registers the number of the instances of the specific class in a specific moment.

Instance level methods determine the behavior of instances. Such methods must be invoked through a specific object. The object on which a method is currently operating is called the current instance. Current instances are referenced by the this or self keywords in most languages.

There is no current instance in class level methods; class level methods are used for manipulating class level attributes.

Instance level methods can be setters or getters. Setter methods cause the current instance to change its state by modifying one (or perhaps all) of its attributes. Setter methods bear the characteristics of procedures; new attribute values are specified with the help of parameters. Getter methods are more similar to ordinary functions; such methods return the current state of the current instance.
The process of instantiation includes reserving memory space, allocating instance level attributes, and setting the initial state of the object. The language system sets the object identifier (OID). From this point the object is *alive* and knows which class created it. Object-oriented languages use a special programming construct, the constructor to set the initial state of the object.

Object-oriented languages manage the current instance with implicit references.

Object-oriented languages define a special asymmetric relation between classes, the *inheritance* relation. Inheritance is a primary means of reusability.

The inheritance relation attaches a new class (*child class*, *derived class*, or *subclass*) to an already existing class (the *parent*, *base* or *superclass*). The essence of inheritance is that the subclass receives (inherits) all the attributes and methods (subject to visibility constraints) of its own superclass, and can use them immediately. Furthermore, the subclass may define new attributes and methods, rename inherited constructs, redeclare inherited names, modify visibility, and re-implement methods.

We distinguish between *single* and *multiple inheritance*. In the case of single inheritance, a class is allowed to have exactly one superclass, while multiple inheritance supports more than one parents. It is true in both cases that any number of subclasses may be derived from a given superclass.

A subclass may be the superclass of another class, resulting in a *hierarchy of classes*. A class hierarchy is a tree in the case of single inheritance, and an acyclic graph in the case of multiple inheritance. All languages, however (even those supporting multiple inheritance) distinguish a dedicated *root class* which has no super class. *Leaf classes*, on the other hand, have no subclass.

Identical names in different super classes may cause conflicts in the case of multiple inheritance.

The following figure demonstrates an inheritance tree:
In the inheritance hierarchy, classes on the same path are in a direct or indirect inheritance relation. From one direction, the subclasses are called **derived** classes, while classes in the opposite direction are called **ancestors**. In the example above, **Closed shape**, **Polygon** and **Triangle** are derived from the **Shape** class, while **Ellipse**, **Closed shape**, and **Shape** are the ancestors of the **Circle** class.

Classes that are not connected to each other with a derived-ancestor relation are called **client** classes.

A class may constrain the **visibility** of its elements. In general, object-oriented languages support the following visibility levels. Elements of **public** visibility are accessible to all client classes. Elements at the **protected** level are accessible only to derived classes. **Private** elements can only be used in the specific class; more specifically, a subclass inherits the private attributes and methods, too, but it is unable to refer to these elements directly (**invisible inheritance**). Several object-oriented languages distinguish a fourth level as well, which is usually based on the structural units of the program.

The **substitutability** of instances related by inheritance is a distinctive manifestation of reusability. Instances of a derived class may substitute instances of the ancestor class anywhere in the source code. A **Circle** object is an **Ellipse** and a **Closed shape** object at the same time, thus it is object of these classes, too. However, it is instance of only the **Circle** class.

A subclass may re-implement its inherited methods, which implies that identical method specifications in different classes may have different implementations. Such methods are called **polymorphic**. Polymorphism raises the question as to which method implementation will be executed if such a method is invoked. The answer is given by a language mechanism, the **binding**.

Object-oriented languages may support two types of binding. With **static binding**, the question is decided at compile-time; substitutability is not an issue. The method of the object’s declared class—as specified in the source text—will be executed in every case.
Dynamic binding chooses the appropriate method implementation at run-time, on the basis of substitutability. The method implemented in or inherited by the object’s instantiating class will be executed.

Thus, if there is a Perimeter method in the Polygon class which is re-implemented in the Quadrangle class, then, in the case of static binding, when the Perimeter method is invoked on a polygon object Polygon's implementation is executed, even if that polygon object is in fact a quadrangle object. If the language supports dynamic binding, the implementation of the Quadrangle class will be chosen.

One group of object-oriented languages supports dynamic binding. Other languages provide for both types of binding: one is the default; the other must be set by the programmer in an explicit manner.

Object-oriented languages usually allow the programmer to overload method names. This means that it is possible to define methods with the same name but with different implementations in the very same class. However, in this case the method specifications must differ (at least) in the number, order and type of their parameters in order to help resolve the references.

Most object-oriented languages recognize the notion of the abstract class. An abstract class is a language feature which specifies behavior patterns that are to be refined by derived classes.

An abstract class contains abstract methods, i.e. methods which have specifications alone, without any implementation. Both abstract and concrete classes may be derived from an abstract class. All methods of a concrete class must have implementations. A class remains abstract as long as at least one of its methods is abstract.

Abstract classes cannot be instantiated.

Certain object-oriented languages enable the programmer to create classes which may have no subclass (such classes are the “leafs” of the inheritance hierarchy). Leaf classes cannot be abstract because that would make it impossible to ever concretize the class.

There are object-oriented languages that support the concept of parameterized class (collection). Parameterized classes are essentially the generics of the object-oriented world.

Objects are OO constructs managed in the memory. An object is always transient, which means that it ceases to exist after the program that created it ends. However, I/O tools make it possible to save the state of any object, and later reconstruct another object with the same state. Non-language OO systems (e.g. database management systems) manage persistent objects. A persistent object survives the application it was created by. A persistent object loaded into memory again retains its identity, i.e. it remains the same object.

Every now and then the memory should be freed from objects that are no longer used. Object-oriented languages provide two kinds of solution to this end. One group of languages requires that the programmer should terminate or destroy objects in an explicit manner. The other group of languages offers automatic memory reclamation (garbage collection). Garbage collection is an asynchronous, automatic background mechanism which looks for unused objects and reclaims the memory wasted by such objects.

Object-oriented languages are of two kinds. Pure object-oriented languages follow closely the OO paradigm, and contain no features from other paradigms. Certain pure OO languages support only one class hierarchy, which provides the language system and the development environment at the same time. In such languages, programming consists of defining custom classes, placing them in the class hierarchy, and instantiating these classes.

Hybrid object-oriented languages are based on a non-OO paradigm (e.g. procedural, functional, logic etc.), whose basic features are complemented with object-oriented features. It is possible to write programs along both paradigms in hybrid languages. Generally, hybrid languages do not contain built-in class hierarchies (class libraries may be used instead). The programmer is expected to create his or her own class hierarchies.

Certain pure object-oriented languages advocate the principle of uniformity. Such languages support only one programming feature, the object. All other constructs—e.g. methods, classes—are expressed as objects.

The object-oriented paradigm has been designed along the principles of the imperative paradigm. As a result, OO languages are algorithmic, and are originally compiler based. SIMULA 67 was the first language to contain object-oriented features. Later the developer group of Smalltalk completed the OO paradigm with other features. A great variety of hybrid object-oriented languages has emerged since that time, including the more recent trend of declarative object-oriented languages (CLOS, Prolog++).

22.1. Questions

1. What is an object?
2. What is a class?
3. What is instantiation?
4. What is the difference between instance and class level elements?
5. What is inheritance?
6. What is binding?
7. Describe inheritance hierarchies.
8. How do you classify OO languages?
9. What is meant by uniformity?
10. What is the constructor?
11. What is the polymorphism?
12. What is the substitutability?
13. What is the visibility?
14. What is the abstract class?
23. fejezet

23 JAVA

This chapter has been written on the basis of The Java™ Language Specification Third Edition, often taking over certain parts directly.

23.1. Java Basics

The Java programming language has a compiler. The object code does not contain code that is native to the processor; it instead contains bytecodes — the machine language of the Java Virtual Machine (JVM). The JVM is an abstract machine, it is an interpreter. The JVM is a platform-independent solution; it can be a bit slower than native code. However, advances in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability. The JVM is available on many different operating systems. Some virtual machines perform additional steps at runtime to give the application a performance boost.

The Java programming language includes automatic storage management, typically using a garbage collector, to avoid the safety problems of explicit deallocation. High-performance garbage-collected implementations can have bounded pauses to support systems programming and real-time applications. The language does not include any unsafe constructs, such as array accesses without index checking, since such unsafe constructs would cause a program to behave in an unspecified way.

Java programs are written using the Unicode character set. Java differentiates upper and lower case letters. Most Java keywords have been taken over from C. There are no standard identifiers. The length of identifiers is unlimited.

23.2. Types

The built-in type system of Java is the following:

• primitive types
  – numeric types
    * integral types (BYTE SHORT INT LONG CHAR)
    * floating point types (FLOAT DOUBLE)
  – BOOLEAN
• reference types
  – class
  – interface
  – array
Primitive values do not share state with other primitive values. A variable whose type is a primitive type always holds a primitive value of that same type.

The operators of primitive types are the following: comparison, numerical, string concatenation, conditional, and logical operators.

An object is a class instance or an array. The reference values (often just references) are pointers to these objects, and a special null reference, which refers to no object.

The operators on references to objects are:
- Field access
- Method invocation
- The cast operator
- The string concatenation operator +
- The instanceof operator
- The reference equality operators == and !=
- The conditional operator ? :  

There may be many references to the same object. Objects have state, stored in the fields of objects that are instances of classes or in the variables that are the components of an array object. If two variables contain references to the same object, the state of the object can be modified using one variable’s reference to the object, and then the altered state can be observed through the reference in the other variable.

Java handles one-dimensional arrays. The number of indexes must be given; the range of indexes is 0..number-1. Arrays also have class, which is of a reference type.

There is also a special null type, the type of the expression NULL, which has no name. Because the null type has no name, it is impossible to declare a variable of the null type or to cast to the null type. The null reference is the only possible value of an expression of null type. The null reference can always be cast to any reference type. In practice, the programmer can ignore the null type and just pretend that NULL is merely a special literal that can be of any reference type.

### 23.3. Literals

Literals are the following:

- integer
  - decimal
  - hexadecimal
  - octal
- floating point
  - decimal
  - hexadecimal
- boolean (TRUE FALSE)
- character
  ' characters
- string
  
  " [character...]"

A string literal always refers to the same instance of class String.

- null (NULL)
23.4. Names

Names are used to refer to entities declared in a program. A declared entity is a package, class type, interface type, member (class, interface, field, or method) of a reference type, type parameter (of a class, interface, method or constructor), parameter (to a method, constructor, or exception handler), or local variable.

Names in programs are either simple, consisting of a single identifier, or qualified, consisting of a sequence of identifiers separated by ".".

Every declaration that introduces a name has a scope, which is the part of the program text within which the declared entity can be referred to by a simple name.

Access control can be specified in a class, interface, method, or field declaration to control when access to a member is allowed. Access is a different concept from scope; access specifies the part of the program text within which the declared entity can be referred to by a qualified name. The default access is that a member can be accessed anywhere within the package that contains its declaration; other possibilities are PUBLIC, PROTECTED, and PRIVATE.

23.5. Block

Java supports the block program unit.

23.6. Variables

Java distinguishes the following kinds of variables:

• class
• instance
• array components
• method parameters
• constructor parameters
• exception handler parameter
• local

A local variable declaration statement declares one or more local variable names.

```
local_variable_declaration_statement:

type variable_declarators;

variable_declarators:

variable_declarator [, variable_declarator...]

variable_declarator:

variable_declarator_id [= variable_initializer]

variable_declarator_id:
```
Every local variable declaration statement is immediately contained by a block. Local variable declaration statements may be intermixed freely with other kinds of statements in the block.

A local variable declaration can also appear in the header of a FOR statement. In this case it is executed in the same manner as if it were part of a local variable declaration statement.

Each variable_declarator in a local variable declaration declares one local variable, whose name is the identifier that appears in the declarator.

The scope of a local variable declaration in a block is the rest of the block in which the declaration appears, starting with its own initializer and including any further declarators to the right in the local variable declaration statement.

The name of a local variable \( v \) may not be redeclared as a local variable of the directly enclosing method, constructor or initializer block within the scope of \( v \), or a compile-time error occurs. The name of a local variable \( v \) may not be redeclared as an exception parameter of a CATCH clause in a TRY statement of the directly enclosing method, constructor or initializer block within the scope of \( v \), or a compile-time error occurs. However, a local variable of a method or initializer block may be shadowed anywhere inside a class declaration nested within the scope of the local variable.

A local variable cannot be referred to using a qualified name; only a simple name.

If a name declared as a local variable is already declared as a field name, then that outer declaration is shadowed throughout the scope of the local variable. Similarly, if a name is already declared as a variable or parameter name, then that outer declaration is shadowed throughout the scope of the local. The shadowed name can sometimes be accessed using an appropriately qualified name.

In Java a local variable declaration statement is an executable statement. Every time it is executed, the declarators are processed in order from left to right. If a declarator has an initialization expression, the expression is evaluated and its value is assigned to the variable. If a declarator does not have an initialization expression, then a Java compiler must prove that every reference to the variable is necessarily preceded by the execution of an assignment to the variable. If this is not the case, then a compile-time error occurs. Each initialization (except the first) is executed only if the evaluation of the preceding initialization expression completes normally. Execution of the local variable declaration completes normally only if evaluation of the last initialization expression completes normally; if the local variable declaration contains no initialization expressions, then executing it always completes normally.

### 23.7. Expressions

The Java programming language guarantees that the operands of operators are evaluated from left to right. The left-hand operand of a binary operator is fully evaluated before any part of the right-hand operand is evaluated.

Operators in Java have been taken over from C.

When an expression in a program is evaluated, the result denotes one of three things:

- A variable
- A value
- Nothing (the expression is said to be void)

The evaluation of an expression can also produce side effects, because expressions may contain embedded assignments, increment operators, decrement operators, and method invocations.

An expression denotes nothing if and only if it is a method invocation that invokes a method that does not return a value, that is, a method declared void. Such an expression can be used only as an expression statement, because every other context in which an expression can appear requires the expression to denote something. An expression statement that is a method invocation may also invoke a method that produces a result; in this case the value returned by the method is quietly discarded.
23.8. Statements

The assignment, expression, empty, IF, SWITCH, WHILE, DO, FOR, and RETURN statements correspond to the respective statements in C. There is no GOTO statement.

```
break_statement:

BREAK [label];
```

A BREAK statement with no label attempts to transfer control to the innermost enclosing SWITCH, WHILE, DO, or FOR statement of the immediately enclosing method or initializer block; this statement then immediately completes normally.

A BREAK statement with label label attempts to transfer control to the enclosing labelled statement that has the same label as its label; this statement then immediately completes normally.

A BREAK statement must refer to a label within the immediately enclosing method or initializer block. There are no non-local jumps.

A CONTINUE statement may occur only in a WHILE, DO, or FOR statement. Control passes to the loop-continuation point of an iteration statement.

```
continue_statement:

CONTINUE label;
```

A CONTINUE statement with no label attempts to transfer control to the innermost enclosing WHILE, DO, or FOR statement of the immediately enclosing method or initializer block, then immediately ends the current iteration and begins a new one.

A CONTINUE statement with label label attempts to transfer control to the enclosing labelled statement that has the same label as its label, then immediately ends the current iteration and begins a new one. The labelled statement must be a WHILE, DO, or FOR statement or a compile-time error occurs. A CONTINUE statement must refer to a label within the immediately enclosing method or initializer block. There are no non-local jumps.

23.9. Packages

Java programs are organized as sets of packages. Each package has its own set of names for types, which helps to prevent name conflicts. A top level type is accessible outside the package that declares it only if the type is declared PUBLIC. The naming structure for packages is hierarchical. The members of a package are class and interface types, which are declared in compilation units of the package, and subpackages, which may contain compilation units and subpackages of their own.

A package can be stored in a file system or in a database.

A package consists of a number of compilation units. A compilation unit has automatically access to all types declared in its package and also imports automatically all of the public types declared in the predefined package java.lang.

For small programs and casual development, a package can be unnamed or have a simple name, but if code is to be widely distributed, unique package names should be chosen. This can prevent the conflicts that would otherwise occur if two development groups happened to pick the same package name and these packages were later to be used in a single program.

The members of a package are its subpackages and all the top level class types and top level interface types declared in all the compilation units of the package.

The form of a compilation unit:

```
compilation_unit:

[package_declaration] [import_declarations] [type_declarations]
```

import_declarations:
Types declared in different compilation units can depend on each other, circularly. A Java compiler must arrange to compile all such types at the same time.

A package declaration in a compilation unit specifies the name of the package to which the compilation unit belongs. The package_name mentioned in a package declaration must be the fully qualified name of the package.

A compilation unit that has no package declaration is part of an unnamed package.

An import declaration allows a static member or a named type to be referred to by a simple name that consists of a single identifier. Without the use of an appropriate import declaration, the only way to refer to a type declared in another package, or a static member of another type, is to use a fully qualified name.
A single-static-import declaration imports all accessible static members with a given simple name from a type. This makes these static members available under their simple name in the class and interface declarations of the compilation unit in which the single-static-import declaration appears.

A static-import-on-demand declaration allows all accessible static members declared in the type named by a fully qualified name to be imported as needed.

Each compilation unit automatically and implicitly imports all of the public type names declared in the predefined package java.lang, as if the declaration

```java
import java.lang.*;
```

appeared at the beginning of each compilation unit, immediately following any package statement. So the names of all those types are available as simple names.

```java
type_declaration:
{class_declaration | interface_declaration | ;}
```

A top level type declaration declares a top level class type or a top level interface type.

By default, the top level types declared in a package are accessible only within the compilation units of that package, but a type may be declared to be public to grant access to the type from code in other packages. The scope of a top level type is all type declarations in the package in which the top level type is declared.

### 23.10. Classes

A class declaration specifies a new named reference type. A nested class is any class whose declaration occurs within the body of another class or interface. A top level class is a class that is not a nested class.

The form of class declaration:

```java
[class_modifiers] CLASS identifier [type_parameters] [super] [interfaces] class_body
class_modifiers:
class_modifier [class_modifier...] class_modifier:
{PUBLIC | PROTECTED | PRIVATE | ABSTRACT | STATIC | FINAL}
type_parameters : <type_parameter [, type_parameter...]>
super:
EXTENDS class_type
interfaces:
IMPLEMENTES {interface_type [, interface_type...]}
```
identifier is the name of class. A named class may be declared ABSTRACT (that means an abstract class) and must be declared ABSTRACT if it is incompletely implemented; such a class cannot be instantiated, but can be extended by subclasses. A class may be declared FINAL, in which case it cannot have subclasses (it is a leaf class).

If a class is declared PUBLIC, then it can be referred to from other packages. Not all modifiers are applicable to all kinds of class declarations. The access modifier PUBLIC pertains only to top level classes and to member classes. The access modifiers PROTECTED and PRIVATE pertain only to member classes within a directly enclosing class declaration. The access modifier STATIC pertains only to member classes.

A class is generic if it declares one or more type variables. These type variables are known as the type parameters of the class and given by type_parameters. A generic class declaration defines a set of parameterized types, one for each possible invocation of the type parameter section. All of these parameterized types share the same class at runtime.

The EXTENDS clause in a class declaration specifies the direct superclass of the current class. Java supports single inheritance. If there is no EXTENDS clause the direct superclass is Object. Object is the root of the inheritance tree in Java. All other classes are derived from this class.

The IMPLEMENTS clause lists the names of interfaces that are direct superinterfaces of the class being declared.

class_body:
{
 [class_body_declarations]
}

class_body_declarations:
{
class_body_declaration | class_body_declarations class_body_declaration
}
class_member_declaration:
{
 class_member_declaration | instance_initializer | static_initializer | constructor_declaration
}

class_body: may contain declarations of members of the class, that is, fields, classes, interfaces and methods. A class body may also contain instance initializers, static initializers, and declarations of constructors for the class. These are not members and therefore are not inherited.

The members of a class type are all of the following:
• Members inherited from its direct superclass
• Members inherited from any direct superinterfaces
• Members declared in the body of the class

class_member_declaration:
{
 field_declaration | method_declaration | class_declaration | interface_declaration
}

23.11. Fields

field_declaration:
{
 [field_modifiers] type _variable_declarators;

 field_modifiers:
field_modifier [field_modifier...]

field_modifier:

{PUBLIC | PROTECTED | PRIVATE | STATIC | FINAL}

variable_declarators:

variable_declarator [, variable_declarator...]

variable_declarator:

variable_declarator_id [= variable_initializer]

variable_declarator_id:

{identifier | variable_declarator_id[]} 

variable_initializer:

{expression | array_initializer}

PUBLIC fields are accessible in all packages, PRIVATE field are accessible only in the class where they are declared, PROTECTED fields are accessible in derived classes and in the package where the class is declared. If there is no modifier, the field is accessible in the package where the class is declared.

If a field is declared STATIC, there exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created. A static field, sometimes called a class variable, is incarnated when the class is initialized.

A field that is not declared static is called an instance variable. Whenever a new instance of a class is created, a new variable associated with that instance is created for every instance variable declared in that class or any of its superclasses.

Every field can be declared FINAL. A final field may only be assigned to once. A blank final is a final field whose declaration lacks an initializer. It is a compile time error if a blank final class variable is not definitely assigned by a static initializer of the class in which it is declared. A blank final instance variable must be definite by the end of every constructor of the class in which it is declared; otherwise a compile time error occurs. Once a final variable has been assigned, it always contains the same value. If a final variable holds a reference to an object, then the state of the object may be changed by operations on the object, but the variable will always refer to the same object. This also applies to arrays, because arrays are objects; if a final variable holds a reference to an array, then the components of the array may be changed by operations on the array, but the variable will always refer to the same array.

Declaring a field final can serve as useful documentation that its value will not change and can help avoid programming errors.

23.12. Methods

method_declaration:

method_header method_body

method_header:

[method_modifiers] [type_parameters] result_type method_declarator [throws]
method_modifiers:

method_modifier [method_modifier]

method_modifier:

{PUBLIC | PROTECTED | PRIVATE | ABSTRACT | STATIC | FINAL}

type_parameters :

<type_parameter [, type_parameter...]>

result_type:

{type | VOID}

method_declarator:

identifier ( [formal_parameter_list] )

throws:

THROWS exception_type_list

exception_type_list:

exception_type [, exception_type...]

exception_type:

{class_type | type_variable}

method_body:

{block | ;}

The access modifiers PUBLIC, PROTECTED, and PRIVATE have been discussed above. An abstract method declaration introduces the method as a member, specifying its method_header, but does not provide an implementation. The declaration of an abstract method \( m \) must appear directly within an abstract class (let it be \( A \)). Every subclass of \( A \) that is not abstract must provide an implementation for \( m \), or a compile-time error occurs.

A method that is declared static is called a class method. A class method is always invoked without reference to a particular object. It is a compile time error for a static method to be declared abstract. A method that is not declared static is called an instance method. An instance method is always invoked with respect to an object, which becomes the current object to which the keywords THIS and SUPER refer during the execution of the method body.

A method can be declared final to prevent subclasses from overriding or hiding it. It is a compile time error to attempt to override or hide a final method. A private method and all methods declared immediately within a final class behave as if they are final, since it is impossible to override them. It is a compile-time error for a final method to be declared abstract.

A method is generic if it declares one or more type variables. These type variables are known as the formal type parameters of the method.

The result_type of a method declares the type of value a method returns, if it returns a value, or states that the method is VOID.
identifier is the name of the method. It is a compile-time error for the body of a class to declare as members two methods with override-equivalent signatures (name, number of parameters, and types of any parameters). Methods and fields may have the same name, since they are used in different contexts and are disambiguated by different lookup procedures.

Two methods have the same signature if they have the same name and argument types.

Two method or constructor declarations M and N have the same argument types if all of the following conditions hold:

- They have the same number of formal parameters (possibly zero)
- They have the same number of type parameters (possibly zero)
- Let <A1,...,An> be the formal type parameters of M and let <B1,...,Bn> be the formal type parameters of N. After renaming each occurrence of a Bi in N’s type to Ai the bounds of corresponding type variables and the argument types of M and N are the same.

The signature of a method m1 is a subsignature of the signature of a method m2 if either

- m2 has the same signature as m1, or
- the signature of m1 is the same as the erasure of the signature of m2. (Type erasure is a mapping from types (possibly including parameterized types and type variables) to types (that are never parameterized types or type variables)).

```latex
formal_parameter_list:
{last_formal_parameter | formal_parameters, last_formal_parameter}

formal_parameters:
{formal_parameter | formal_parameters, formal_parameter}

formal_parameter:
{[FINAL] type variable_declarator_id}

last_formal_parameter:
{[FINAL] type[...] variable_declarator_id | formal_parameter}

variable_declarator_id:
{identifier | variable_declarator_id[1]}
```

If a method or constructor has no parameters, only an empty pair of parentheses appears in the declaration of the method or constructor.

It is a compile-time error if a method or constructor parameter that is declared FINAL is assigned to within the body of the method or constructor.

When the method or constructor is invoked, the values of the actual argument expressions initialize newly created parameter variables, each of the declared type, before the execution of the body of the method or constructor. The identifier that appears in the declaratorId may be used as a simple name in the body of the method or constructor to refer to the formal parameter.

If the last formal parameter is a variable arity parameter of type T, it is considered to define a formal parameter of type T[]. The method is then a variable arity method. Otherwise, it is a fixed arity method. Invocations of a variable arity method may contain more actual argument expressions than formal parameters. All the actual argument expressions that do not correspond to the formal parameters preceding the variable arity parameter will be evaluated and the results stored into an array that will be passed to the method invocation.

The scope of a parameter of a method or constructor is the entire body of the method or constructor. These parameter names may not be redeclared as local variables of the method, or as exception parameters of catch clauses in a try statement of the method.
or constructor. However, a parameter of a method or constructor may be shadowed anywhere inside a class declaration nested within that method or constructor. Such a nested class declaration could declare either a local class or an anonymous class. Formal parameters are referred to only using simple names, never by using qualified names.

A `THROWS` clause is used to declare any checked exceptions that can result from the execution of a method or constructor.

A `method body` is either a block of code that implements the method or simply a semicolon, indicating the lack of an implementation. The body of a method must be a semicolon if the method is abstract.

An instance method `m1` declared in a class `C` overrides another instance method, `m2`, declared in class `A` if all of the following are true:

1. `C` is a subclass of `A`.
2. The signature of `m1` is a subsignature of the signature of `m2`.
3. Either
   - `m2` is `PUBLIC`, `PROTECTED` or declared with default access in the same package as `C`, or
   - `m1` overrides a method `m3`, `m3` is distinct from `m1`, `m3` is distinct from `m2`, such that `m3` overrides `m2`.

Moreover, if `m1` is not abstract, then `m1` is said to implement any and all declarations of abstract methods that it overrides.

If a class declares a `STATIC` method `m`, then the declaration `m` is said to hide any method `mi`, where the signature of `m` is a subsignature of the signature of `mi`, in the superclasses and superinterfaces of the class that would otherwise be accessible to code in the class. A compile time error occurs if a static method hides an instance method.

If two methods of a class (whether both declared in the same class, or both inherited by a class, or one declared and one inherited) have the same name but signatures that are not override-equivalent, then the method name is said to be overloaded. This fact causes no difficulty and never of itself results in a compile time error. There is no required relationship between the return types or between the throws clauses of two methods with the same name, unless their signatures are override-equivalent.

Methods are overridden on a signature-by-signature basis. If, for example, a class declares two public methods with the same name, and a subclass overrides one of them, the subclass still inherits the other method. When a method is invoked, the number of actual arguments (and any explicit type arguments) and the compile time types of the arguments are used, at compile time to determine the signature of the method that will be invoked. If the method that is to be invoked is an instance method, the actual method to be invoked will be determined at run time, using dynamic method lookup.

### 23.13. Instance initializer

```java
instance_initializer:

block
```

An instance initializer declared in a class is executed when an instance of the class is created.

### 23.14. Static initializer

```java
static_initializer:

STATIC block
```

Any static initializers declared in a class are executed when the class is initialized and, together with any field initializers for class variables, may be used to initialize the class variables of the class.
23.15. Constructors

constructor_declaration:

[constructor_modifiers] constructor_declarator [throws] constructor_body

constructor_modifiers:

constructor_modifier [constructor_modifier...]

constructor_modifier:

{PUBLIC | PROTECTED | PRIVATE}

constructor_declarator:

[type_parameters] simple_type_name ( [formal_parameter_list] )

type_parameters:

<formal_parameter [, type_parameter...]>

formal_parameter_list:

{last_formal_parameter | formal_parameters,last_formal_parameter}

formal_parameters:

{formal_parameter | formal_parameters, formal_parameter}

formal_parameter:

{FINAL} type variable_declarator_id

last_formal_parameter:

{[FINAL] type[...] variable_declarator_id | formal_parameter}

variable_declarator_id:

{identifier | variable_declarator_id[]}

throws:

THROWS exception_type_list

exception_type_list:

exception_type [, exception_type...]
The access modifiers PUBLIC, PROTECTED, and PRIVATE have been discussed above. It is possible for a constructor to be declared generic, independently of whether the class the constructor is declared in is itself generic. A constructor is generic if it declares one or more type variables. These type variables are known as the formal type parameters of the constructor.

The formal parameters and formal type parameters of a constructor are identical in structure and behavior to the formal parameters and formal type parameters of a method.

The simple_type_name must be the simple name of the class that contains the constructor declaration; otherwise a compile-time error occurs. In all other respects, the constructor declaration looks just like a method declaration that has no result type.

Constructors are invoked by class instance creation expressions and by explicit constructor invocations from other constructors. Constructors are never invoked by method invocation expressions.

Constructor declarations are not members. They are never inherited and therefore are not subject to hiding or overriding.

The THROWS clause for a constructor is identical in structure and behavior to the THROWS clause for a method.

The first statement of a constructor body may be an explicit invocation of another constructor of the same class or of the direct superclass. If a constructor body does not begin with an explicit constructor invocation and the constructor being declared is not part of the primordial class Object, then the constructor body is implicitly assumed by the compiler to begin with a superclass constructor invocation super();, an invocation of the constructor of its direct superclass that takes no arguments.

Except for the possibility of explicit constructor invocations, the body of a constructor is like the body of a method. A return statement may be used in the body of a constructor if it does not include an expression.

Explicit constructor invocation statements can be divided into two kinds:

• Alternate constructor invocations begin with the keyword THIS (possibly prefaced with explicit type arguments). They are used to invoke an alternate constructor of the same class.

• Superclass constructor invocations begin with the keyword SUPER (possibly prefaced with explicit type arguments). They are used to invoke a constructor of the direct superclass.

Overloading of constructors is identical in behavior to overloading of methods.

If a class contains no constructor declarations, then a default constructor that takes no parameters is automatically provided:

• If the class being declared is the primordial class Object, then the default constructor has an empty body.

• Otherwise, the default constructor takes no parameters and simply invokes the superclass constructor with no arguments.

A compile time error occurs if a default constructor is provided by the compiler but the superclass does not have an accessible constructor that takes no arguments.

A default constructor has no THROWS clause.

It follows that if the nullary constructor of the superclass has a THROWS clause, then a compile time error will occur.
23.16. Instantiation

A class instance creation expression is used to create new objects that are instances of classes.

```
class_instance_creation_expression:

NEW [type_arguments] class_type([argument_list])

argument_list:
{expression | argument_list , expression}
```

A class instance creation expression specifies a class to be instantiated, possibly followed by type arguments (if the class being instantiated is generic), followed by (a possibly empty) list of actual value arguments to the constructor. It is also possible to pass explicit type arguments to the constructor itself (if it is a generic constructor). The type arguments to the constructor immediately follow the keyword NEW.

23.17. Interfaces

An interface declaration introduces a new reference type whose members are classes, interfaces, named constants and abstract methods. This type has no implementation, but otherwise unrelated classes can implement it by providing implementations for its abstract methods.

A nested interface is any interface whose declaration occurs within the body of another class or interface. A top-level interface is an interface that is not a nested interface.

Programs can use interfaces to make it unnecessary for related classes to share a common abstract superclass or to add methods to Object.

An interface may be declared to be a direct extension of one or more other interfaces, meaning that it implicitly specifies all the member types, abstract methods and named constants of the interfaces it extends, except for any member types and named constants that it may hide.

A class may be declared to directly implement one or more interfaces, meaning that any instance of the class implements all the abstract methods specified by the interface or interfaces. A class necessarily implements all the interfaces that it’s direct superclasses and direct superinterfaces do. This multiple interface inheritance allows objects to support multiple common behaviors without sharing any implementation.

A variable whose declared type is an interface type may have as its value a reference to any instance of a class which implements the specified interface. It is not sufficient that the class happen to implement all the abstract methods of the interface; the class or one of its superclasses must actually be declared to implement the interface, or else the class is not considered to implement the interface.

```
interface_declaration:

[interface_modifiers] INTERFACE identifier [type_parameters] [extends interfaces] ↦
interface_body
```

The identifier specifies the name of the interface.

```
interface_modifiers:

{interface_modifier | interface_modifiers interface_modifier}
```

```
interface_modifier:

{PUBLIC | PROTECTED | PRIVATE | ABSTRACT | STATIC}
```
The modifiers PUBLIC, PROTECTED, PRIVATE, and STATIC have been discussed above. Not all modifiers are applicable to all kinds of interface declarations. The access modifiers PROTECTED and PRIVATE pertain only to member interfaces within a directly enclosing class declaration. The access modifier STATIC pertains only to member interfaces. Every interface is implicitly ABSTRACT. This modifier is obsolete and should not be used in new programs.

Each interface_type in the EXTENDS clause of an interface declaration must name an accessible interface type; otherwise a compile-time error occurs.

All interface members are implicitly PUBLIC. They are accessible outside the package where the interface is declared if the interface is also declared PUBLIC or PROTECTED.

The members of an interface are:

- Those members declared in the interface.
- Those members inherited from direct superinterfaces.

Every named constant declaration in the body of an interface is implicitly PUBLIC, STATIC, and FINAL. It is permitted, but strongly discouraged as a matter of style, to redundantly specify any or all of these modifiers.
abstract_method_modifiers:

{abstract_method_modifier | abstract_method_modifiers abstract_method_modifier}

abstract_method_modifier:

{PUBLIC | ABSTRACT}

Every method declaration in the body of an interface is implicitly ABSTRACT and PUBLIC. It is permitted, but strongly discouraged as a matter of style, to redundantly specify these modifiers.

23.18. Exception handling

In Java an exception is thrown for one of three reasons:

- An abnormal execution condition was synchronously detected by the Java virtual machine. Such conditions arise because:
  - the evaluation of an expression violates the normal semantics of the language, such as integer division by zero;
  - an error occurs while loading or linking parts of the program;
  - a limitation on a resource is exceeded, such as using too much memory.

  These exceptions are not thrown at an arbitrary point in the program, but rather at a point where they are specified as a possible result of an expression evaluation or statement execution.

- A THROW statement was executed.

- An asynchronous exception occurred (for example an internal error has occurred in the virtual machine).

The possible exceptions in a program are organized in a hierarchy of classes, rooted at class Throwable, a direct subclass of Object. The classes Exception and Error are direct subclasses of Throwable. The class RuntimeException is a direct subclass of Exception. Programs can use the pre-existing exception classes in THROW statements, or define additional exception classes as subclasses of Throwable or of any of its subclasses, as appropriate. To take advantage of the Java platform’s compile-time checking for exception handlers, it is typical to define most new exception classes as checked exception classes, specifically as subclasses of Exception that are not subclasses of RuntimeException. The class Exception is the superclass of all the exceptions that ordinary programs may wish to recover from.

A compiler for the Java programming language checks, at compile time, that a program contains handlers for checked exceptions, by analyzing which checked exceptions can result from execution of a method or constructor. For each checked exception which is a possible result, the THROWS clause for the method or constructor must mention the class of that exception or one of the superclasses of the class of that exception. This compile-time checking for the presence of exception handlers is designed to reduce the number of exceptions which are not properly handled.

The unchecked exception classes are the class RuntimeException and its subclasses, and the class Error and its subclasses. All other exception classes are checked exception classes. The Java API defines a number of exception classes, both checked and unchecked. Additional exception classes, both checked and unchecked, may be declared by programmers.

The checked exception classes named in the THROWS clause are part of the contract between the implementor and user of the method or constructor. The THROWS clause of an overriding method may not specify that this method will result in throwing any checked exceptions which the overridden method is not permitted, by its THROWS clause, to throw. When interfaces are involved, more than one method declaration may be overridden by a single overriding declaration. In this case, the overriding declaration must have a THROWS clause that is compatible with all the overridden declarations.

Exceptions are handled with the TRY statement:

try_statement:
{TRY block catches | TRY block [catches] FINALLY block}

catches:

{catch_clause | catches catch_clause}

catch_clause:

CATCH ( formal_parameter ) block

formal_parameter:

variable_modifiers type variable_declarator_id

variable_declarator_id:

{identifier | variable_declarator_id[]}

A TRY statement executes a block. If an exception is thrown and the TRY statement has one or more CATCH clauses that can catch it, then control will be transferred to the first such CATCH clause. If the TRY statement has a FINALLY clause, then another block of code is executed, no matter whether the TRY block completes normally or abruptly, and no matter whether a CATCH clause is first given control.

A TRY statement may have CATCH clauses (also called exception handlers). A CATCH clause must have exactly one parameter (which is called an exception parameter); the declared type of the exception parameter must be the class Throwable or a subclass of Throwable. The scope of a parameter of an exception handler that is declared in a CATCH clause of a TRY statement is the entire block associated with the CATCH.

If the execution of the TRY block completes abruptly because an exception E has been detected, the exception is handled in one of the following ways:

- If the run-time type of E is assignable to the parameter of any of the CATCH clauses of the TRY statement, then the first (leftmost) such CATCH clause is selected. The exception E is assigned to the parameter of the selected CATCH clause, and the block of that CATCH clause is executed. If that block completes normally, then the TRY statement completes normally; if that block completes abruptly for any reason, then the TRY statement completes abruptly for the same reason.

- If the run-time type of E is not assignable to the parameter of any CATCH clause of the TRY statement, then the TRY statement completes abruptly.

A THROW statement causes an exception to be thrown. The result is an immediate transfer of control that may exit multiple statements and multiple constructor, instance initializer, static initializer and field initializer evaluations, and method invocations until a TRY statement is found that catches the thrown value. If no such TRY statement is found, then execution of the thread that executed the throw is terminated after invocation of the uncaughtException method for the thread group to which the thread belongs.

The form of THROW statements:

THROW expression;

23.19. Parallel programming

The Java virtual machine supports the simultaneous execution of multiple threads. These threads execute code in parallel that operates on values and objects residing in a shared main memory. Threads may be implemented on several hardware processors, by time-slicing a single hardware processor, or by time-slicing several hardware processors.

Threads are represented by the Thread class. The only way for a user to create a thread is to create an object of this class; each thread is associated with such an object.

The details of parallel programming and further language features are beyond the scope of this book.
23.20.  Questions

1. What is the JVM?
2. Explain the difference between primitive types and reference types.
3. What is meant by access control? What levels of access does Java distinguish?
4. What is a package, and how is it used?
5. Describe Java classes.
6. What is the difference between instance level and class level members?
7. What are the rules for instantiating generic classes?
8. When does a method override another method?
10. What is the difference between interfaces and classes?
11. Describe the class hierarchy of Java exceptions.
12. How can exceptions be handled in Java?
13. What is the role of the Thread class?
24. fejezet

24 THE FUNCTIONAL PARADIGM

Functions are the focal constructs of the functional paradigm. A program written in a functional (or applicative) language is composed of type declarations, class declarations, function declarations, function definitions, and an initial expression. The initial expression is a series of (possibly nested) function calls. The execution of the program boils down to the evaluation of the initial expression. The evaluation of expressions can be conceived of as the textual replacement of function calls with the body of function definitions (with the appropriate parameters). The exact semantics of replacement is determined by the evaluation (transcription) model of the given language.

In the case of functional languages, the language system cannot be separated from the development environment. Functional languages are predominantly interactive interpreter-based systems, but they also contain a compiler. The central component of a functional language is the reduction (transcription) system. The reduction system is confluent if the order of the transcription of expressions does not affect the outcome.

The most important building blocks of functional programs are custom functions. In theory, custom functions are the same as the functions of procedural languages. The body of the function determines how the return value should be calculated in light of the actual parameters. The body consists of functional language expressions.

Functional language systems offer a great number of built-in functions. Custom functions may be defined with the help of built-in and previously defined functions (function combination).

Most functions of a functional language are recursive by default; it is also possible to define mutually recursive functions.

The reduction of the original expression (based on the evaluation strategy implemented in the language) starts with the transcription of a reducible sub-expression (redex). Expressions that cannot be reduced any further are normal form expressions.

With lazy evaluation, the outer redex at the left side of the expression is reduced first. This entails that if the expression is a function call actual parameters are evaluated only if they are needed (this is essentially the call by need parameter passing). Lazy evaluation always arrives at the normal form, if such exists.

Eager evaluation starts with the inner redex at the left; actual parameters are evaluated first (call by name parameter passing). Eager evaluation is often more effective, but it is not guaranteed to terminate, even if the normal form exists.

24.1. Questions

1. What is the initial expression?
2. What kinds of evaluation exist?
25. fejezet

25 THE LOGIC PARADIGM AND PROLOG

The logic paradigm was born in the early 1970’s when the first logic programming language, Prolog was released. The logic paradigm builds upon the concepts and features of mathematical logic. Prolog is based on first-order predicate calculus and the resolution algorithm.

A logical program is a set of logical statements about an abstract model. The statements formalize the attributes of the elements of the model and the relations between them.

The subset of statements that describe a concrete relation is called a predicate. In general, the execution of a logic program is the constructive proof of the thesis that follows from the statements. The statements of a program define a solution environment, and the question (or the task) is formulated in this environment; the question is answered by the inference engine or theorem-prover.

A statement can be a fact or a rule. Phrase is the umbrella term for statements and questions. Certain logic languages contain phrases that go beyond the logical paradigm. Declarations refine the application of predicates, while directives define the runtime environment of the program.

In Prolog, the task of solving a problem is distributed between the programmer, who is responsible only for ensuring the truth of the programs expressed in logical form, and the theorem-prover, which is responsible for solving problems efficiently.

Prolog is a general-purpose high-level language. The full version of Prolog contains metalogical and non-logical language elements beside the logical feature system, and is embedded into an interactive development environment. First, we introduce the functionality of pure (abstract) Prolog, and then take a look at other elements at the end of the chapter.

A program written in pure Prolog is a collection of user-defined predicates which do not reference build-in predicates.

Prolog is an untyped language with an interpreter. Its character set is standard; certain versions include national language characters among letters. The language distinguishes between upper case and lower case letters.

The programmer may place a comment after the % sign; comments last until the end of the line.

Phrases end with the . (dot) sign.

The building block of Prolog programs is the term, which can be simple or complex. A simple term is a constant or a variable. Constants are names or numbers.

A name is an identifier which starts with a lower case letter or one of the following characters: +, -, *, /, \, ^, <, >, =, ~, :, ?, @, #, &. The following are reserved names: ;, !, [ ], { }.

A number is a character string that formally corresponds to the integer and real numeric literal of procedural languages.

Variables in Prolog are special since they have no type, and their address is not accessible. The name of a variable is an identifier that starts with an underscore sign or an upper case letter. The value component is also managed in a special way. Prolog variables correspond to the unknown of mathematical equations. Prolog variables are subject to the rule of single assignment. A variable either has a value or not. If the variable does not have a value, its name represents itself as a character string inside its scope. If the variable has a value component the name stands for the value component. The value cannot be overwritten. A variable in pure Prolog receives its value from the Prolog inference engine.

The scope of a variable is the phrase in which its name occurs. An exception to this rule is the variable whose name is the _ (unnamed variable); every occurrence of _ marks a different variable.
The aim of executing a pure Prolog program is to determine the possible values of the variables in questions. It is a general Prolog convention to start the name of variables irrelevant to the result with an underscore sign.

A complex term is of the following form:

```
name(argument [, argument ...])
```

where argument is an arbitrary term, or an arbitrary character string enclosed in single quotes.

A fact is a complex term which is a true statement.

Rules consist of a head and a body separated by a delimiter, namely the `:-` character sequence in Prolog. The head is a complex term; the body is a sequence of complex terms separated by commas (these are predicates).

Rules are inference rules: the head is true if the body is true; the comma represents a short-circuit and operation.

Questions have only a body.

Prolog declarations and directives are of the following form:

```
:- body
```

Variables in Prolog statements are universally quantified, while variables in questions are existentially quantified.

Facts are rules without a body, which implies that the body can always be considered true. Questions are rules without a head, and may express inquiries about which elements of the solution environment make the predicates true, or may express a “YES/NO” type question.

Rules can be recursive.

Any number of rules may have identical heads. In such cases, the relation between the arguments is the union of the relations between the statements.

Example:

```
father(john,frank). % this is a fact

father(frank,peter). % this is a fact

grandpa(X,Z):- father(X,Y),father(Y,Z). % this is a rule

grandpa(Somebody,peter). % this is a question
```

The Prolog inference engine builds a search tree in the memory and traverses it in a preorder manner. The engine never builds the tree completely, only the current path is available. The nodes of the tree represent the actual form of the question; edges are labelled by the variable substitutions performed in the given step.

Answering a question operates on the facts and the rules in the order of writing. Two techniques are available to solve a question: matching and backtracking.

Answering a question involves the following steps:

1. The root of the search tree contains the original question; this is the current node at the beginning of the program’s execution.
2. If the body of the question is empty in the current node, a solution is found. The system prints the solution, and then asks the user if he or she wants another solution. If the user responds with a no, the program ends. If the user requires further solutions, the search continues at Step 4.
3. If the body of the question is not empty in the current node, the inference engine takes the first predicate of the body, and performs a match on the first fact. If the match fails, the engine takes the next fact and attempts to match it. Once a matching fact is found, a new node is created which contains the actual form of the question as it has left the first predicate. The edge is labelled with the variable substitutions required by the matching.

If no matching facts are found, the engine tries to match the heads of the rules. If a match is found, a new node is added to the tree and the edge is labelled. The predicate in the current node is overwritten by the body of the matching head.
If no facts or heads of rules match, Prolog reports that it has reached a dead end, and continues at Step 4. Otherwise the new node becomes the current node and execution continues at Step 2.

4. This is the backtracking step. If the current node is the root of the tree the program terminates, and there are no further solutions (including the case when no solutions have been found at all). Otherwise the current node is deleted, and the previous node will become the current node; at the same time variable substitutions that label the edge connecting the two nodes are also deleted. Then the execution continues at Step 3, and tries to find a match with the help of statements that follow the statements previously used.

**The matching algorithm:**

1. If both term series to match are empty the algorithm ends (successful match). Otherwise the algorithm compares the first elements of the two sequences according to Step 2. If these match, the remaining elements of the sequences are examined as specified in Step 1.

2. If both terms are constants, matching is successful or fails depending on whether the terms are identical as character strings or not.

3. A constant and a complex term do not match.

4. In the case of two complex terms, matching is unsuccessful if the names of the terms or the number of their arguments are different. Otherwise the argument sequences are compared as specified in Step 1.

5. If both terms are variables, any one of them can be substituted with the other. Usually statement variables are assigned values.

6. If only one of the terms is a variable, it is substituted with the other (constant or complex) term.

We are going to illustrate the workings of the matching algorithm with the following example. The tree is of the following form at the beginning:

```
grandpa(Somebody,peter)
```

Step 4 of the algorithm rules out matching the predicate with facts (because of their different names); however, Steps 4, 5, and 6 (variable substitution steps) allow the predicate to be matched with the head of the rule. Following the variable substitution and the single value assignment, the tree is built as follows.

```
grandpa(Somebody,peter)
  X ← Somebody
  Z ← peter
  father(Somebody,Y), father(Y,peter)
```

Now the first predicate can be matched with the first fact on the basis of Steps 4 and 6:
Considering Steps 4 and 2, the second fact matches, while the first fails:

\[
\text{grandpa}(\text{Somebody}, \text{peter})
\]

\[
\begin{align*}
X &\leftarrow \text{Somebody} \\
Z &\leftarrow \text{peter} \\
\text{father}(&\text{Somebody}, Y), \text{father}(&Y, \text{peter}) \\
\text{Somebody} &\leftarrow \text{john} \\
Y &\leftarrow \text{frank} \\
\text{father}(&\text{frank}, \text{peter})
\end{align*}
\]

The body of the question is now empty, which means that we have arrived at the first solution: \text{john}. Prolog prints the name and asks if we wish to continue searching for further solutions. Should we continue, backtracks and dead ends would result in the following tree sequences and no further solutions:
Since the next step would require the engine to backtrack from the root, the algorithm ends; no further solutions have been found.

Matching the predicates corresponds to the procedure calls of procedural languages. The call itself is the predicate, and matching determines which procedure to call. The evaluation of the parameters is characterized by sequential binding and numerical matching. Matching substitutes parameter passing; finally, the procedure call is overwritten with the body (similarly to macros).

Variable arguments act as output parameters; other kinds of arguments are input parameters.

The arguments of the “YES/NO” type questions may not contain variables.

Practical problems (e.g., counting) often require solutions where pure logical constructs cannot be applied or they are not efficient enough. This is where Prolog’s metalogical features are of great use.

The expression construct in Prolog is evaluated only in certain contexts. Prolog has built-in operators (e.g., arithmetic and relational operators), and the programmer is allowed to define custom operators, or even overload the built-in ones, usually with the help of directives.

The +, −, ∗, /, ** arithmetic operators of Prolog bear a conventional meaning (the last operator being the exponentiation operator). Arithmetic operators can be infix or prefix; their operands must be numbers. Note that these operators are complex term names, i.e., the expression 2+3 in fact corresponds to the complex term +(2, 3). As a result, expressions that contain
arithmetic operators are evaluated only in specific contexts, such as the right side of the infix binary operator `is`. The `is` operator expects an arithmetic expression on its right side, which is evaluated and then matched with the left operand.

The `is` operator can be employed by the programmer to assign a value to a variable relying on the rules of matching. Namely, if the left side of the `is` operator contains a variable which has not been assigned a value yet, the variable is assigned the value of the right-side expression.

Since the evaluation of an `is`-expression relies on matching, the expression `S is S+1` is completely meaningless. Namely, if `S` already has value, matching will be unsuccessful (the values of `S` and `S+1` cannot be identical). If `S` has no value, an error occurs (it is impossible to increase a non-existing value by 1).

One of the built-in operators of Prolog is the cut operator `!`, which has no operands (it is a complex term without any arguments). The cut operator is used to control the process of searching for solutions. If it occurs in a question body as a predicate it “cuts” the search tree at the given node; more specifically, the cut operator causes the program to terminate if a backtrack is required from the given node. The role of the cut operator is essential in the definition of recursive rules as it helps avoid infinite recursion.

The following example of factorial calculation illustrates the proper use of the cut operator:

```prolog
factorial(0, X) :- !, X is 1.
factorial(N, X) :- M is N-1, factorial(M, Y), X is N*Y.
```

Effective programming languages support features like I/O, string handling, exception handling, or graphics. Such features also exist in Prolog, but these are non-logical predicates, and have no declarative interpretation.

### 25.1. Questions

1. Define the following concepts:
   - phrase, statement, question, predicate;
   - declarations and directives;
   - term;
   - fact, rule.

2. Describe how the inference engine answers a question.

3. What are the major steps of the matching algorithm?

4. What is the role of Prolog’s metalogical features? Give examples.

5. What are the conditions that make backtracking impossible?
26. fejezet

BIBLIOGRAPHY

Colophon

This electronic book was prepared in the framework of project TÁMOP-4.1.2-08/1/A-2009-0046 Eastern Hungarian Informatics Book

Nemzeti Fejlesztési Ügynökség http://ujszechenyiterv.gov.hu/ 06 40 638-638