Programming in Oberon

**A derivative of
Programming in Modula-2
(1982)**

Niklaus Wirth

Oct. 2004

O

This text is an introduction to programming in general, and a guide to programming in the language Oberon in particular. It is primarily oriented towards people who have already acquired some basic knowledge of programming and would like to deepen their understanding in a more structured way. Nevertheless, an introductory chapter is added for the benefit of the beginner, displaying in a concise form some of the fundamental concepts of computers and of programming them. The text is therefore also suitable as a self-contained tutorial. The notation used is Oberon, which lends itself well for a structured approach and leads the student to a working style that has generally become known under the heading of structured programming.

As a manual for programming in Oberon, the text covers (almost) all facilities of this language. Part 1 covers the basic notions of variable, expression, assignment, conditional and repeated statement, and the data structure of the indexable array. Together with Part 2, which introduces the important concept of the procedure or subroutine, it contains essentially the material commonly discussed in introductory programming courses. Part 2 also presents data types and structures and constitutes the core of an advanced course on programming. Part 3 introduces the notion of a module, a concept that is fundamental in the design of large pro-

gramming systems, and to programming in teams. The two basic notions of files and texts are introduced in the forms of modules containing the operations performed on files and texts. Texts are presented as our standard medium for program input and output.

The language Oberon, published in 1988, has a long history. It is a descendant of Algol 60 (1960), Pascal (1970), and Modula (1979). Algol 60 [1] was designed by an international committee of 13 scientists, and it was the first language to be specified with a rigorous formalism, a syntax, in machine-independent form. Algol was largely oriented toward numerical algorithms, as were computers at that time. Pascal [2] was the result of an enduring debate about widening the range of application of Algol, and it became widely used, in particular but not only, in education. Modula-2 [3] introduced the concept of modules, the parts of large systems connected by clearly specified interfaces. The module allows to hide details of procedures and variables from its users (clients), and it embodies the principle of information hiding. The language Oberon [4] emerged from the urge to reduce the complexity of programming languages, of Modula in particular. This effort resulted in a remarkably concise language. The extent of Oberon, the number of its features and constructs, is smaller even than that of Pascal. Yet it is considerably more powerful.

The one feature that was added in Oberon was the extensibility of data types (record types). Whereas in strongly types languages, such as Algol, Pascal, and Modula, every constant, variable, or function has a fixed type, recognizable from the program text, Oberon allows to define hierarchies of types, and to determine the actual type of a variable

(within the hierarchy) at run- time. This, together with records containing fields of procedural types, is the stem of object-oriented programming. Such records are then called objects, and the procedural fields are called methods. Here, we do not touch upon this subject.

Zurich, Oct. 1, 2004 N.W.

REFERENCE

1. P. Naur, Ed. *Revised Report on the Algorithmic Language Algol 60.* Comp. J. 5, 349-367 (1962), and Comm. ACM, 6 (1963) 1-17.

2. N. Wirth. *The Programming Language Pascal.* Acta Informatica, 1 (1971), 35-63.

3. N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1982. ISBN 0-387-50150-9

4. N. Wirth. *The Programming Language Oberon.* Software - Practice and Experience, 18, 7, (July 1988), 671-690.

Part I

VAR

INTRODUCTION

Although this manual assumes that its reader is already familiar with the basic notions of computer and programming, it may be appropriate to start out with the explanation of some concepts and their terminology. We recognize that - with rare exceptions - programs are written - more appropriately: designed - with the purpose of being interpreted by a computer. The computer then performs a process, i.e. a sequence of actions, according to the specifications given by that program. The process is also called a computation.

The program itself is a text. Since it specifies a usually fairly complex process, and must do so with utmost precision and care for all details, the meaning of this text must be specified very precisely. Such precision requires an exact formalism. This formalism has become known as a *language*. We adopt this name, although a language is normally spoken and much less precisely defined. Our purpose here is to learn the formalism or language Oberon. It has a long tradition among programming languages. Its ancestor was Algol 60, followed by Pascal and Modula-2.

A program usually specifies a process that causes its interpreter, i.e. the computer, to read data (the so-called input) from some sources and to vary its subsequent actions according to the accepted data. This implies that a program

does not only specify a (single) process, but an entire - usually unbounded - class of computations. We have to ensure that these processes act according to the given specifications (or should we say expectations?) in all cases of this class. Whereas we could verify that this specification is met in the case of a single computation, this is impossible in the general case, because the class of all permitted processes is much too large. The conscientious programmer ensures the correctness of his program by careful design and analysis. Careful design is the essence of professional programming.

The task of designing a program is further complicated by the fact that the program not only must describe an entire class of computations, but often should also be interpreted (executed) by different interpreters (computers). At earlier times, this required the manual transcription of the program from its source form into different computer codes, taking into account their various characteristics and limitations. The difficulties have been drastically reduced, albeit not eliminated, by the creation of high level languages with formal definitions and the construction of automatic translators converting the program into the codes of the various computers.

In principle, the formal language should be defined in an abstract, perhaps axiomatic fashion without reference to an actual computer or interpretation mechanism. If this were achieved, the programmer would have to understand the formal language only. However, such generality is costly and often restrictive, and in many cases the programmer should still know the principal characteristics of his computer(s). Nevertheless, the qualified programmer will make as little reference to specific computer characteristics as possible

and rely exclusively on the rules of the formal language in order to keep his program general and portable. The language Oberon assists in this task by confining computer dependencies to specific objects, by allowing to encapsulate them in specific, small parts of a program text.

From the foregoing it follows that a translation process lies between the program's formulation and its interpretation. This process is called a compilation, because it condenses the program's source text into a cryptic computer code. The quality of this compilation may be crucial to the efficiency of the program's ultimate interpretation. We stress the fact that there may be many compilers for a given language (even for the same computer). Some may be more efficient than others. We recognize that efficiency is a characteristic of implementations rather than the language. It therefore is important to distinguish between the concepts of language and implementation.

We summarize:

- Aprogram is a piece of text.

- The program specifies computations or processes.

- A process is performed by an interpreter, usually a computer, interpreting (executing) the program.

- The meaning of the program is specified by a formalism called programming language.

- A program specifies a class of computations, the input data acting as parameter of each individual process.

- Prior to its execution, a program text is translated into computer code by a compiler. This process is called a compilation.

Program design includes ensuring that all members of this class of computations act according to specification. This is done by careful analytic verification and by selective empirical testing of characteristic cases.

Programs should refrain from making reference to characteristics of specific interpreters (computers) whenever possible. Only the lack of such reference ensures that their meaning can be derived from rules of the language.

A compiler is a program translating programs from their source form to specific computer codes. Programs need to be compiled before they are executed. Programming in the wider sense not only includes the formulation of the program, but also the concrete preparation of the text, its compilation, correction of errors, so-called debugging, and the planning of tests. The modern programmer uses many tools for these tasks, including text editors, compilers, and debuggers. He also has to be familiar with the environment of these components. We shall not describe these aspects, but concentrate on the language Oberon.

1ST EXAMPLE

Let us follow the steps of development of a simple program and thereby explain some of the fundamental concepts of programming and of the basic facilities of Oberon. The task shall be, given 2 natural numbers x and y, to compute their greatest common divisor (*gcd*). The mathematical knowledge needed for this problem is the following:

1. if x equals y, x (or y) is the desired result

2. the *gcd* of 2 numbers remains unchanged, if we replace the larger number by the difference of the numbers, i.e. subtract the smaller number from the larger one.

Expressed in mathematical terms, these rules take the form

1. $gcd(x, x) = x$

2. if $x > y$, $gcd(x, y) = gcd(x - y, y)$

The basic recipe, the so-called algorithm, is then the following: Change the numbers x and y according to rule 2 such that their difference decreases. Repeat this until they are equal. Rule 2 guarantees that the changes are such that $gcd(x, y)$ always remains the same, and rule 1 guarantees that we finally find the result.

Now we must put these recommendations into terms of Oberon. A 1st attempt leads to the following sketch. Note that the symbol # means "unequal".

```
WHILE x # y DO
  "apply rule 2, reducing the difference
   and maintaining x > 0 and y > 0"
END
```

The sentence within quotes is plain English. The 2nd version refines the 1st one by replacing it with formal terms:

```
WHILE x # y DO
  IF x > y THEN x := x-y ELSE y := y-x END
END
```

This piece of text is not yet a complete program, but it shows already the essential characteristic of a structured programming language. Version 1 is a statement, and this statement contains another, subordinate statement (within quotes). In version 2 this is elaborated, and yet further subordinate statements emerge (expressing the replacement of a value x by another value $x - y$). This hierarchy of statements expresses the underlying structure of the algorithm. It becomes explicit due to the structure of the language, allowing the nesting of components of a program. It is therefore important to know the language's structure (syntax) in full detail. Textually we express nesting or subordination by appropriate indentation. Although this is not required by the rules of the language, it helps in the understanding of a text very considerably.

Reflecting an algorithm's inherent structure by the textual structure of the program is a key idea of structured programming. It is virtually impossible to recognise the meaning of a program when its structure is removed, such as done by a compiler when producing computer code. And we should keep in mind that a program is worthless, unless it exists in some form in which a human can understand it and gain confidence in its design.

We now proceed towards the goal of producing a complete program from the above fragment. We realize that we need to specify an action that assigns initial values to the variables x and y, as well as an action that makes the result visible. For this purpose we should actually know about a computer's facilities to communicate with its user. Since we do not wish to refer to a specific machinery, and particularly not in such a frequent and important case as the generation of output, we introduce abstractions of such communication facilities. These facilities are not directly part of the language, but are procedures declared in some (library) modules, to which all programs have access. We postulate read and write procedures as shown in the following example, and will assume that data are thus read from a keyboard and written on a display.

```
Texts.Scan(S); x := S.i;
Texts.Scan(S); y := S.i;
WHILE x # y DO
   IF x > y THEN x := x-y ELSE y := y-x END
END;
Texts.WriteInt(W, x, 6)
```

The procedure Scan scans an input text and reads a (non-negative) integer (*S.i*). The procedure WriteInt outputs an integer as specified by its 1st parameter (*x*). The 2nd parameter (6) indicates the number of digits available for the representation of this value in the output text. Both procedures are taken from the imported (library) module Texts, and they use a locally declared scanner *S* and a globally declared writer *W*, connecting to an input source (the text following the command) and output sink (the Log text) imported from the environment Oberon. Details are explained in Chapter **??** of this tutorial.

In the next and final version we complete it such that it becomes a genuine Oberon text:

```
PROC Gcd*;
  VAR x, y: INT; S: Texts.Scanner;
BEGIN
  Texts.OpenScanner(S, Oberon.Par.text,
                       Oberon.Par.pos);
  Texts.Scan(S); x := S.i;
  Texts.WriteString(W, " x =");
  Texts.WriteInt(W, x, 6);

  Texts.Scan(S); y := S.i;
  Texts.WriteString(W, " y =");
  Texts.WriteInt(W, y, 6);

  WHILE x # y DO
    IF x > y THEN x := x-y ELSE y := y-x END
  END;
  Texts.WriteString(W, " gcd =");
  Texts.WriteInt(W, x, 6);
```

```
    Texts.WriteLn(W);
    Texts.Append(Oberon.Log, W.buf)
  END Gcd.
```

The essential additions in this step are declarations. In Oberon, all names of objects occurring in a program, such as variables and constants, have to be declared. A declaration introduces the object's identifier (name), specifies the kind of the object (whether it is a variable, a constant, or something else) and indicates general, invariant properties, such as the type of a variable or the value of a constant.

The entire piece of text is called a procedure, given a name, and has the following format (see also Chapter **??**):

```
  PROC name;
    <declarations>
  BEGIN
    <statements>
  END name
```

In this and the following examples, a particular property of the Oberon system becomes apparent. Whereas in older languages, the notions of program and executable unit were identical, we distinguish carefully between them in Oberon terminology: What used to be called a program is now called a module, a system unit typically incorporating variables and executable statements resident in a system (see also Chapter **??**). Compilers accept a module as a compilable unit of text. An executable unit of program we call a procedure. In the Oberon system, any parameterless procedure can be used as a command, if its name is marked with an asterisk. It can be activated by clicking on its name visible

anywhere on the display. A module may contain one or several commands. As a consequence, the preceding and following examples of "programs" appear in the form of procedures (commands), and we will always assume that they are embedded in a module importing 2 service modules Texts and Oberon and containing the declaration of a writer *W* for output:

```
MODULE Pattern;
  IMPORT Texts, Oberon:
  VAR W: Texts.Writer;

  PROC command*; (*such as Gcd*)
  BEGIN ... (*using W for output*) ...
    Texts.Append(Oberon.Log, W.buf)
  END command;
BEGIN Texts.OpenWriter(W)
END Pattern.
```

A few more comments concerning our Gcd example are in order. As already mentioned, the procedures WriteLn, WriteString, WriteInt and Scan are not part of the language Oberon itself. They are defined in a module called Texts which is presumed to be available. The often encountered modules Texts, Files and Oberon will be defined and explained in Chapter **??** of this book. Here we merely point out that they need to be imported in order to be known in a program. This is done by including the names of these modules in the import list in the importing module's heading.

The procedure WriteString outputs a string, i.e. a Sequence of characters (enclosed in quotes). The procedure WriteLn

inserts a line break in the output text. For further explanations we refer to chapters **??** and **??** of this book.

And this concludes the discussion of our 1st example. It has been kept quite informal. This is admissible because the goal was to explain an existing program. However, programming is designing, creating new programs. For this purpose, only a precise, formal description of our tool is adequate. In the next chapter, we introduce a formalism for the precise description of correct, "legal" program texts. This formalism makes it possible to determine in a rigorous manner whether a written text meets the language's rules.

SYNTAX NOTATION

A formal language is an infinite set of sequences of symbols.
The members of this set are called sentences, and in the case
of a programming language these sentences are programs.
The symbols are taken from a finite set called the vocabulary.
Since the set of programs is infinite, it cannot be enumer-
ated, but is instead defined by rules for their composition.
Sequences of symbols that are composed according to these
rules are said to be syntactically correct programs; the set of
rules is the syntax of the language.

Programs in a formal language then correspond to grammat-
ically correct sentences of spoken languages. Every sentence
has a structure and consists of distinct parts, such as subject,
object, and predicate. Similarly, a program consists of parts,
called syntactic entities, such as statements, expressions, or
declarations. If a construct A consists of B followed by C, i.e.
the concatenation BC, then we call B and C syntactic factors
and describe A by the syntactic formula

```
A = BC
```

If, on the other hand, an A consists of a B or, alternatively,
of a C, we call B and C syntactic terms and express A as

```
A = B|C
```

Parentheses may be used to group terms and factors. It is noteworthy that here A, B, and C denote syntactic entities of the formal language to be described, whereas the symbols =, | , parentheses, and the period are symbols of the meta-notation describing syntax. The latter are called meta-symbols, and the meta-notation introduced here is called Extended Backus-Naur Formalism (EBNF).

In addition to concatenation and choice, EBNF also allows to express option and repetition. If a construct A may be either a B or nothing (empty), this is expressed as

```
A = [B]
```

and if an A consists of the concatenation of any number of Bs (including none), this is denoted by

```
A = {B}
```

This is all there is to EBNF! A few examples show how sets of sentences are defined by EBNF formulas:

```
(A|B)(C|D): AC AD BC BD
     A[B]C: ABC AC
     A{BA}: A ABA ABABA ABABABA ...
    {A|B}C: C AC BC AAC ABC BBC BAC...
```

Evidently, EBNF is itself a formal language. If it suits its purpose, it must at least be able to describe itself! In the following definition of EBNF in EBNF, we use the following names for entities:

- statement: a syntactic equation

- expression: a list of alternative terms

- term: a concatenation of factors

- factor: a single syntactic entity or a parenthesized expression

The formal definition of EBNF is now given as follows:

```
syntax = {statement}
statement = identifier "=" expression
expression = term {"|" term}
term = factor {factor}
factor = identifier | string | "(" expression ")"
        | "[" expression "]" | "{" expression "}"
```

Identifiers denote syntactic entities; strings are sequences of symbols taken from the defined language's vocabulary. For the denotation of identifiers we adopt the widely used conventions for programming languages, namely:

- An identifier consists of a sequence of letters and digits, the 1st of which must be a letter;

- A string consists of any sequence of characters enclosed by quote marks (or apostrophes).

A formal statement of these rules in terms of EBNF is given in subsequent chapters.

REPRESENTATION OF OBERON PROGRAMS

The preceding chapter has introduced a formalism, by which the structures of well-formed programs will subsequently be defined. It defines, however, merely the way in which programs are composed as sequences of symbols, in contrast to sequences of characters. This "shortcoming" is quite intentional: the representation of symbols (and thereby programs) in terms of characters is considered too much dependent on individual implementations for the general level of abstraction appropriate for a language definition. The creation of an intermediate level of representation by symbol sequences provides a useful decoupling between language and ultimate program representation. The latter depends on the available character set. As a consequence, we need to postulate a set of rules governing the representation of symbols as character sequences. The symbols of the Oberon vocabulary are divided into the following classes:

1. identifiers,

2. numbers,

3. strings,

4. operators and delimiters, and

5. comments.

The rules governing their representation in terms of the standard ISO character set are the following:

1. Identifiers are sequences of letters and digits, the first of which must be a letter. They are case-sensitive.

   ```
   id = letter {letter|digit}.
   ```

   Examples of well-formed identifiers are:

   ```
   Alice likely jump BlackBird SR71
   ```

   Examples of invalid identifiers (and the reason):

`sound proof`	(blank space is not allowed)
`sound-proof`	(neither is a hyphen)
`Miller's`	(no apostrophe allowed)
`2N`	(1st character must be a letter)

   Sometimes an identifier has to be qualified by another identifier; this is expressed by prefixing i with j and a period ($j.i$); the combined identifier is called a *qualified identifier* (abbreviated as `qid`). Its syntax is

   ```
   qid = {id "."} id
   ```

2. Numbers are either integers or real numbers. The former are denoted by sequences of digits. Numbers must not include any spaces. Real numbers contain a decimal point and a fractional part. In addition, a scale factor may be appended. It is specified by the letter `E` and an integer which is possibly preceded by

a sign. The E is pronounced as "times 10 to the power of".

Examples of well-formed numbers are:

```
1981 1 3.25 51E3 4.0E-10
```

Examples of character sequences that are not recognized as numbers (and the reason):

```
1,5         (no comma may appear)
1'000'000   (neither may apostrophs)
3.5En       (no letters allowed, except E)
```

The exact rules for forming numbers are given by the following syntax:

```
number = int | real
int    = dec | hex H
real   = [sign]dec.{digit}[E[sign]dec]
sign   = + | -
dec    = digit {digit}
hex    = digit {hexdig}
hexdig = digit | A-F
```

Note: Integers are taken as octal numbers, if followed by the letter E, or as hexadecimal numbers, if followed by the letter H.

3. Strings are sequences of any characters enclosed in quote marks. In order that the closing quote is recognized unambiguously, the string itself evidently cannot contain a quote mark.

Examples of strings are:

```
"no comment"      '"'      "Buck's Corner"
```

4. Operators and delimiters are either special characters or reserved words. Those composed of special characters are:

+	addition, set union
-	subtraction, set difference
∗	multiplication, set intersection
/	division, symmetric set difference
:=	assignment
&	logical AND
~	logical NOT
=	equal
#	unequal
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
( )	parentheses
[ ]	index brackets
{ }	set braces
(* *)	comment brackets
^	dereferencing operator
, . ; : .. |	punctuation symbols

Reserved words are written in upper-case letters and must not be used as identifiers. Hence it is advantageous to memorize the following enumeration list,

whose meaning will be explained throughout subsequent chapters:

BY	DIV	CASE	ARRAY	IMPORT
DO	END	ELSE	BEGIN	MODULE
IF	FOR	PROC	CONST	RECORD
IN	MOD	THEN	ELSIF	REPEAT
IS	SET	TRUE	FALSE	RETURN
OF	VAR	TYPE	UNTIL	POINTER
OR	TO		WHILE	

It is customary to separate consecutive symbols by a space, i.e. one or several blanks. However, this is mandatory only in those cases where the lack of such a space would merge the two symbols into one. For example, in "IF x = y THEN" spaces are necessary in front of x and after y, but could be omitted around the equal sign.

5. Comments may be inserted between any 2 symbols. They are arbitrary sequences of characters enclosed in the comment brackets (* and *). Comments are skipped by compilers and serve as help info to us human beings. They might also serve to signal instructions (options) to the compiler.

STATEMENTS AND EXPRESSIONS

The specification of an action is called a statement. Statements can be interpreted (executed), and that interpretation (execution) has an effect. The effect is a transformation of the state of the computation, the state being represented by the collective values of the program's variables. The most elementary action is the assignment of a value to a variable. The assignment statement has the form

```
assignment = designator ":=" expression.
```

and its corresponding action consists of 3 parts in this sequence:

1. Evaluate the designator designating a variable.

2. Evaluate the expression yielding a value.

3. Replace the value of the variable identified in 1. by the value obtained in 2.

Simple examples of assignments are:

```
i := 1    x := y+z
```

Here i obtains the value 1, and x the sum of y and z, and the previous values are lost. Evidently, every variable in

an expression must previously have been assigned a value. Observe that the following pairs of statements, executed in sequence, do not have the same effect:

```
i := i+1; j := 2*i
j := 2*i; i := i+1
```

Assuming the initial value $i = 0$, the 1st pair yields $i = 1$, $j = 2$, whereas the 2nd pair yields $j = 0$. If we wish to exchange the values of variables i and j, the statement sequence

```
i := j; j := i
```

will not have the desired effect. We must introduce a temporary value holder, say k, and specify the 3 consecutive assignments

```
k := i; i := j; j := k
```

An expression is in general composed of several operands and operators. Its evaluation consists of applying the operators to the operands in a prescribed sequence, in general taking the operators from left to right. The operands may be constants, or variables, or functions. (The latter will be explained in a later chapter.) The identification of a variable will, in general, again require the evaluation of a designator. Here we shall confine our presentation to simple variables designated by an identifier. Arithmetic expressions (there exist other expressions too) involve numbers, numeric variables, and arithmetic operators. These include the basic operations of addition (+), subtraction (-), multiplication (*),

and division. They will be discussed in detail in the chapter on basic data types. Here it may suffice to mention that the slash (/) is reserved for dividing real numbers, and that for integers we use the operator DIV which truncates the quotient.

An expression consists of consecutive terms. The expression

```
T0 + T1 + ... + Tn
```

is equivalent to

```
((T0 + T1) + ...) + Tn
```

and is syntactically defined by the rules

```
SimpleExpression =[+|-] term {AddOperator term}.
AddOperator = + | - | OR
```

Note: for the time being, the reader may consider the syntactic entities expression and SimpleExpression as equivalent. Their difference and the operators OR, AND, and NOT will be explained in the chapter on the data type BOOL.

Each term similarly consists of factors. The term

```
F0 * F1 * ... * Fn
```

is equivalent to

```
((F0 * F1) * ... ) * Fn
```

and is syntactically defined by the rules

```
   term = factor {MulOperator factor}.
 MulOperator = * | / | DIV | MOD | &.
```

Each factor is either a constant, a variable, a function, or an expression itself enclosed by parentheses.

Examples of (arithmetic) expressions are

```
2*3 + 4*5         = (2*3)+(4*5)     = 26
15 DIV 4 * 4     = (15 DIV 4)*4   = 12
15 DIV (4*4)     = 15 DIV 16      = 0
2 + 3*4 - 5       = 2 + (3*4) - 5 = 9
6.25/1.25 + 1.5 = 5.0 + 1.5       = 6.5
```

The syntax of factors, implying that a factor may itself be an expression, is evidently recursive. The general form of designators will be explained later; here it suffices to know that an identifier denoting a variable or a constant is a designator.

```
 factor = number | string | set | designator
   [ActualParameters] | (expression) | ~factor.
```

The rules governing expressions are actually quite simple, and complicated expressions are rarely used. Nevertheless, we must point out a few basic rules that are well worth remembering.

1. Every variable in an expression must previously have been assigned a value.

2. 2 operators must never be written side by side. For instance a * -b is illegal and must be written as a*(-b).

3. The multiplication sign must never be omitted when a multiplication is required. For example, 2n is illegal and must be written as 2*n.

4. MulOperators are binding more strongly than AddOperators.

5. When in doubt about evaluation rules (i.e. precedence of operators), use additional parentheses to clarify. For example, a+b*c may just as well be written as a+(b*c).

The assignment is but one of the possible forms of statements. Other forms will be introduced in the following chapters. We enumerate these forms by the following syntactic definition

```
statement = [ assignment | ProcedureCall | IfStatement
   | WhileStatement | RepeatStatement | ForStatement ].
```

Several of these forms are structured statements, i.e. some of their components may be statements again. Hence, the definition of statement is, like that of expressions, recursive.

The most basic structure is the sequence. A computation is a sequence of actions, where each action is specified by a statement, and is executed after the preceding action is completed. This strict sequentiality in time is an essential assumption of sequential programming. If a statement S_1 follows S_0, then we indicate this sequentiality by a semicolon

$$S_0; S_1$$

This statement separator (not terminator) indicates that the action specified by S_0 is to be followed immediately by the action corresponding to S_1. A sequence of statements is syntactically defined as

```
StatementSequence = statement {; statement}.
```

The syntax of statements implies that a statement may con-
sist of no symbols at all. In this case, the statement is said
to be empty and evidently denotes the null action. This
curiosity among statements has a definite reason: it allows
semicolons to be inserted at places where they are actually
superfluous, such as at the end of a statement sequence.

6

CONTROL STRUCTURES

It is a prime characteristic of computers that individual actions can be selected, repeated, or performed conditionally depending on some previously computed results. Hence the sequence of actions performed is not always identical with the sequence of their corresponding statements. The sequence of actions is determined by control structures indicating repetition, selection, or conditional execution of given statements.

6.1 REPETITIVE STATEMENTS

The most common situation is the repetition of a statement or statement sequence under control of a condition: the repetition continues as long as the condition is met. This is expressed by the *while statement*. Its syntax is

```
WhileStatement = WHILE expr DO StatementSequence END
```

and its corresponding action is:

1. Evaluate the condition which takes the form of an expression yielding the value TRUE or FALSE;

2. If the value is TRUE, execute the statement sequence and then repeat with step 1; if the value is FALSE, terminate.

The expression is of type BOOL. This will be further discussed in the chapter on data types. Here it suffices to know that a simple comparison is a Boolean expression. An example was given in the introductory example, where repetition terminates when the 2 comparands have become equal. Further examples involving while statements are:

1. Initially, let $q = 0$ and $r = x$; then count the number of times y can be subtracted from x, i.e. compute the quotient $q = x$ DIV y, and remainder $r = x$ MOD y, if x and y are natural numbers.

```
WHILE r >= y DO r := r-y; q := q+1 END
```

2. Initially, let $z = 1$ and $i = k$; then multiply z k times by x, i.e. compute $z = xk$, if z and k are natural numbers:

```
WHILE i > 0 DO z := z*x; i := i-1 END
```

When dealing with repetitions, it is important to remember the following points:

1. During each repetition, progress must be made towards meeting the goal, namely "getting closer" to satisfying the termination condition. An obvious corollary is that the condition must be somehow affected from within the repeated computation. The following statements are either incorrect or dependent on some critical precondition as stated.

```
WHILE i > 0 DO
    k := 2*k (*i is not changed*)
END
```

```
WHILE i # 0 DO
  i := 1-2 (*i must be even and positive*)
END

WHILE n # i DO
  n := n*i; i := i+1
END
```

2. If the condition is not satisfied initially, the statement is vacuous, i.e. no action is performed.

3. In order to obtain a grasp of the effect of the repetition, we need to establish a relationship that is stable, called an invariant. In the division example above, this is the equation $q * y + r = x$ holding each time the repetition is started. In the exponentiation example it is $z * x^i = x^k$ which, together with the termination condition $i = 0$ yields the desired result $z = x^k$.

4. The repetition of identical computations should be avoided (although a computer has infinite patience and will not complain). A simple rule is to avoid expressions within repetitive statements, in which no variable changes its value. For example, the statement

```
WHILE i < 3*N DO tab[i] := x + y*z + z*i;
                 i := i+1 END
```

should be formulated more effectively as

```
n := 3*N; u := x + y*z;
WHILE i < n DO tab[i] := u + z*i; i := i+1 END
```

In addition to the while statement, there is the *repeat statement*, syntactically defined as

```
RepeatStatement = REPEAT StatementSequence UNTIL expr
```

The essential difference to the while statement is that the termination condition is checked each time after (instead of before) execution of the statement sequence. As a result, the sequence is executed at least once. An advantage is that the condition may involve variables that are undefined when the repetition is started.

```
REPEAT i := i+5; j := j+7; k := i DIV j UNTIL k > 23
REPEAT r := r-y; q := q+1 UNTIL r < y
```

6.2 CONDITIONAL STATEMENTS

The conditional statement, also called if statement, has the form

```
IfStatement = IF expr THEN StatementSequence
         {ELSIF expr THEN StatementSequence}
                  [ELSE StatementSequence]
             END
```

The following example illustrates its general form.

```
    IF R1 THEN S1
  ELSIF R2 THEN S2
  ELSIF R3 THEN S3
          ELSE S4 END
```

The meaning is obvious from the wording. However, it must be remembered that the expressions R1 ... R3 are evaluated one after the other, and that as soon as one yields the value TRUE, its corresponding statement sequence is executed, whereafter the IF statement is considered as completed. No further conditions are tested. Examples are:

```
     IF x = 0 THEN s :=   0
 ELSIF x < 0 THEN s := -1
               ELSE s :=   1 END


 IF ODD(k) THEN z := z*x END
 IF k > 10 THEN k := k-10; d := 1
              ELSE d := 0 END
```

The constructs discussed so far enable us to develop a few simple, complete programs as subsequently described. The 1st example is an extension of our introductory example computing the greatest common divisor *gcd* of 2 natural numbers *x* and *y*. The extension consists in the addition of 2 variables *u* and *v*, and statements which lead to the computation of the least common multiple *lcm* of *x* and *y*. The *lcm* and *gcd* are related by the equation

```
 lcm(x,y) * gcd(x,y) = x*y
```

As explained earlier, this and following examples will be formulated as command procedures and are assumed to be embedded in module Pattern defined in **??** which imports modules Texts and Oberon.

```
 PROC gcdlcm*;
```

```
    VAR x, y, u, v: INT; S: Texts.Scanner;
  BEGIN Texts.OpenScanner(S, Oberon.Par.text,
                              Oberon.Par.pos):
    Texts.Scan(S); x := S.i;
    Texts.WriteString(W, " x =");
    Texts.WriteInt(W, x, 6);
    Texts.Scan(S); y := S.i;
    Texts.WriteString(W, " y =");
    Texts.WriteInt(W, y, 6);
    u := x; v := y;
    (*gcd(x,y) = gcd(x0,y0), x*v + y*u = 2*x0*y0*)
    WHILE x # y DO
      IF x > y THEN x := x-y; u := u+v
               ELSE y := y-x; v := v+u
      END
    END;
    Texts.WriteInt(W, x, 6);
    Texts.WriteInt(W, (u+v) DIV 2, 6);
    Texts.WriteLn(W);
    Texts.Append(Oberon.Log, W.buf)
  END gcdlcm.
```

This example again shows the nesting of control structures. The repetition expressed by a while statement includes a conditional structure expressed by an if statement, which in turn includes 2 statement sequences, each consisting of 2 assignments. This hierarchical structure is made transparent by appropriate indentation of the "inner" parts.

Another example demonstrating a hierarchical structure computes the real number x raised to the power i, where i is a non-negative integer.

```
  PROC Power*;
    VAR i: INT; x, z: REAL; S: Texts.Scanner;
  BEGIN Texts.OpenScanner(S, Oberon.Par.text,
                               Oberon.Par.pos);
    Texts.Scan(S);
    WHILE S.class = Texts.Real DO
      x := S.x; Texts.WriteString(W, " x =");
      Texts.WriteReal(W, x, 16); Texts.Scan(S);
      i := S.i; Texts.WriteString(W, " i =");
      Texts.WriteInt(W, i, 4);
      z= 1.0:
      (* z * x^i = x0^i0 *)
      WHILE i > 0 DO z := z*x; i := i-1 END;
      Texts.WriteReal(W, z, 16);
      Texts.WriteLn(W); Texts.Scan(S)
    END;
    Texts.Append(Oberon.Log, W.buf)
  END Power.
```

Here we subject the computation to yet another repetition:
Each time a result has been computed, another value pair x, i
is requested. This outermost repetition is controlled by the
relation S.class = Texts.Real, which indicates whether
a real number x had actually been read. As an example,
activation of the command

```
  M.Power 5.0 2   2.0 5   3.0 3 ~
```

will generate the results 25.0, 32.0, 27.0.

The straight-forward computation of a power by repeated
multiplication is, although obviously correct, not the most
economical. We now present a more sophisticated and more

efficient solution. It is based on the following consideration: The goal of the repetition is to reach the value $i = 0$. This is done by successively reducing i, while maintaining the invariant $z * x^i = x_0^{i_0}$, where x_0 and i_0 denote the initial values of x and i. A faster algorithm therefore must rely on decreasing i in larger steps. The solution given here halves i. But this is only possible, if i is even. Hence, if i is odd, it is 1st decremented by 1. Of course, each change of i must be accompanied by a corrective action on z in order to maintain the invariant.

- A detail: the subtraction of 1 from i is not expressed by an explicit statement, because it is performed implicitly by the subsequent division by 2.

- 2 further details are noteworthy:

  1. The function ODD(i) is TRUE, if i is an odd number, FALSE otherwise.

  2. x and z denote real values, as opposed to integer values. Hence, they can represent fractions, too.

```
PROC Power*;
  VAR i: INT; x, z: REAL; S: Texts.Scanner;
BEGIN Texts.OpenScanner(S, Oberon.Par.text,
                             Oberon.Par.pos);
  Texts.Scan(S);
  WHILE S.class = Texts.Real DO
    x := S.x; Texts.WriteString(W, " x =");
    Texts.WriteReal(W, x, 16): Texts.Scan(S);
    i := S.i; Texts.WriteString(W, " i=");
    Texts.WriteInt(W, i, 4);
    z := 1.0;
```

```
      WHILE i > 0 DO
        (* z* x^i = x0^i0 *)
        IF ODD(i) THEN z := z*x END;
        x := x*x; i := i DIV 2
      END;
      Texts.WriteReal(W, z, 16);
      Texts.WriteLn(VW); Texts.Scan(S)
    END;
    Texts.Append(Oberon.Log, W.buf)
  END Power.
```

The next sample command has a structure that is almost
identical to the preceding program. It computes the loga-
rithm of a real number x whose value lies between 1 and 2.
The invariant in conjunction with the termination condition
($b = 0$) implies the desired result $sum = \log_2 x$.

```
  PROC Log2*;
    VAR x, a, b, sum: REAL; S: Texts.Scanner;
  BEGIN Texts.OpenScanner(S, Oberon.Par.text,
                                 Oberon.Par.pos);
    Texts.Scan(S);
    WHILE S.class = Texts.Real DO
      x := S.x; Texts.WriteReal(W, x, 15); (*1<=x<2*)
      a :=x; b := 1.0; sum := 0.0;
      REPEAT (*log_2(x) = sum + b*log_2(a)*)
        a := a*a; b := 0.5*b;
        IF a >= 2.0 THEN
          sum := sum + b; a := 0.5*a
        END
      UNTIL b < 1.0E-7;
      Texts.WriteReal(W, sum, 16);
      Texts.WriteLn(W); Texts.Scan(S)
```

```
      END;
      Texts.Append(Oberon.Log, W.buf)
  END Log2.
```

Normally, routines for computing standard mathematical functions need not be programmed in detail, because they are available from a collection of modules similar to those for input and output. Such a collection is, somewhat inappropriately, called a library. In the following example, again exhibiting the use of a repetitive statement, we use routines for computing the cosine and the exponential function from a library called Math and generate a table of values for a damped oscillation. Typically, the available standard routines include the *sin*, *cos*, *exp*, *ln* (logarithm), *sqrt* (square root), and the *arctan* functions.

```
  PROC Oscillation*;
    CONST dx = 0.19634953; (*pi/16*)
    VAR i, n: INT; x, y, REAL; S: Texts.Scanner;
  BEGIN Texts.OpenScanner(S, Oberon.Par.text,
                                  Oberon.Par.pos):
    Texts.Scan(S); n := S.i; Texts.WriteInt(W, n, 6);
    Texts.Scan(S); r := S.x; Texts.WriteReal(W, r, 15);
    Texts.WriteLn(W); i := 0; x := 0.0;
    REPEAT x := x + dx; i := i+1;
            y := Math.exp(-r*x) * Math.cos(x);
            Texts.WriteReal(W, x, 15);
   Texts.WriteReal(W, y, 15);
   Texts.WriteLn(W)
    UNTIL i >= n;
    Texts.Append(Oberon.Log, W.buf)
  END Oscillation.
```

7

ELEMENTARY DATA TYPES

We have previously stated that all variables need be declared. This means that their names are introduced in the heading of the program. In addition to introducing the name (and thereby enabling a compiler to detect and indicate misspelled identifiers), declarations have the purpose of associating a data type with each variable. This data type represents information about the variable which is permanent in contrast, for example, to its value. This information may again aid in detecting inconsistencies in an erroneous program, inconsistencies that are detectable by mere inspection of the program text without requiring its interpretation.

The type of a variable determines its set of possible values and the operations which may be applied to it. Each variable has a single type that may be deduced from its declaration. Each operator requires operands of a specific type and produces a result of a specific type. It is therefore visible from the program text, whether or not a given operator is applicable to a given variable.

Data types may be declared in the program. Such constructed types are usually based on composition of basic types. There exist a number of most frequently used, elementary types that are basic to the language and need not be declared. They are called standard types and will

be introduced in this chapter, although some have already appeared in previous examples.

In fact, not only variables have a type, but so do constants, functions, operands and (results of) operators. In the case of a constant, the type is usually deducible from the constant's notation, otherwise from its explicit declaration.

First we describe the standard data types of Oberon, and thereafter elaborate on the form of declarations of variables and constants. Further kinds of data types and declarations are deferred to later chapters.

7.1 INT

This type represents the whole numbers, and any value of type INT is therefore an integer. Operators applicable to integers include the basic arithmetic operations

	Operations
$+$	addition
$-$	subtraction
$*$	multiplication
DIV	division
MOD	modulus (remainder)

Table 1: Operations on sets

Integer division is denoted by DIV. If we define the integer quotient and the modulus of x and y by q = x DIV y, r = x MOD y, then q and r are related by the equation x = q*y +r and by the constraint $0 \leq r < y$.

For example

```
 15 DIV 4 =  3    15 MOD 4 = 3    15 =    3*4+3
-15 DIV 4 = -4   -15 MOD 4 = 1   -15 = (-4)*4+1
```

Sign inversion is denoted by the monadic minus sign. Fur-
thermore, there exist the operators ABS(x) and ODD(x), the
former yielding the absolute value of x, the latter the boolean
result "x is odd".

Every computer will restrict the set of values of type INT to
a finite set of integers, usually the interval $[-2^{N-1}, 2^{N-1} - 1]$,
where N is a small integer, often 16 or 32, depending on the
number of bits a computer uses to represent an integer. If
an arithmetic operation produces a result that lies outside
that interval, then overflow is said to have occured. The
computer will give an appropriate indication and, usually,
terminate the computation. The programmer should en-
sure that overflows will not result during execution of the
program.

7.2 REAL

Values of type REAL are real numbers. The available op-
erators are again the basic arithmetic operations and ABS.
Division is denoted by / (instead of DIV). Constants of
type REAL are characterized by having a decimal point and
possibly a decimal scale factor. Examples of real number
denotations are

```
1.5   1.50   1.5E2   2.34E-2   0.0
```

The scale factor consists of the capital letter E followed by
an integer. It means that the preceding real number is to be
multiplied by 10 raised to the scale factor. Hence

```
   1.5E2 = 150.0,    2.34E-2 = 0.0234
```

The important point to remember is that real values are internally represented as pairs consisting of a fractional number and a scale factor. This is called floating-point representation. Of course, both parts consist of a finite number of digits. As a consequence, the representation is inherently inexact, and computations involving real values are inexact because each operation may be subject to truncation or rounding.

The following program makes that inherent imprecision of computations involving operands of type REAL apparent. It computes the harmonic function

```
  H(n) = 1 + 1/2 + 1/3 + ... + 1/n
```

in 2 different ways, namely once by summing terms from left to right, once from right to left. According to the rules of arithmetic, the 2 sums ought to be equal. However, if values are truncated (or even rounded), the sums will differ for sufficiently large n. The correct way is evidently to start with the small terms.

```
  PROC Harmonic*;
    VAR i, n: INT; x, d, s1, s2: REAL;
        S: Texts.Scanner;
  BEGIN
    Texts.OpenScanner(S, Oberon.Par.text,
                         Oberon.Par.pos);
    Texts.Scan(S);
    WHILE S.class = Texts.Int DO
      n := S.i; Texts.WriteString(W, "n =");
               Texts.WriteInt(W, n, 6);
```

```
      s1 := 0.0; d := 0.0; i := 0;
      REPEAT d := d + 1.0; INC(i);
            s1 :=s1 + 1.0/d;
      UNTIL i >= n;
      Texts.WriteReal(W, s1, 16);
      s2 := 0.0;
      REPEAT s2 := s2 + 1.0/d;
             d :=  d - 1.0; DEC(i)
      UNTIL i = 0;
      Texts.WriteReal(W, s2, 16);
      Texts.WriteLn(W); Texts.Scan(S)
    END;
    Texts.Append(Oberon.Log, W.buf)
  END Harmonic.
```

The major reason for strictly distinguishing between real numbers and integers lies in the different representation used internally. Hence, also the arithmetic operations are implemented by instructions which are distinct for each type. Oberon postulates that expressions with mixed operands always are of type REAL.

A real number x is transformed into an integer by the standard function FLOOR(x). It yields the largest integer not greater than x.

```
  FLOOR(3.1416) = 3   FLOOR(-2.7) = -3   FLOOR(-5.0) = -5
```

7.3   BOOL

A BOOL value is one of the two logical truth values denoted by the symbols TRUE and FALSE. Boolean variables are usually denoted by identifiers which are adjectives, the value

TRUE implying the presence, FALSE the absence of the indicated property. A set of logical operators is provided which, together with BOOL variables, form BOOL expressions. These operators are & (and), OR, and ˜ (not). Their results are explained as follows:

```
p  & q = "both p and q are TRUE"
p OR q = "either p or q or both are TRUE"
~p = "p is FALSE"
```

The operators' exact definition, however, is slightly different, although the results are identical:

```
p  & q = IF p THEN q    ELSE FALSE
p OR q = IF p THEN TRUE ELSE q
```

This definition implies that the second operand need not be evaluated, if the result is already known from the evaluation of the first operand. The notable property of this definition of Boolean connectives is that their result may be well-defined even in cases where the second operand is undefined. As a consequence, the order of the operands may be significant.

Sometimes it is possible to simplify Boolean expressions by application of simple transformation rules. A particularly useful rule is de Morgan's law stating the equivalences

```
~p  & ~q = ~(p OR q)
~p OR ~q = ~(p  & q)
```

Relations produce a result of type BOOL, i.e. TRUE if the relation is satisfied, FALSE if not. For example

```
7 = 12        FALSE
7 < 12        TRUE
1.5 >= 1.6    FALSE
```

Relations are syntactically classified as expressions, and the two comparands are so-called SimpleExpressions (see also chapter on expressions and statements). The result of a comparison is of type BOOL and can be used in control structures such as if, while, and repeat statements. The symbol # stands for unequal.

```
expr = exp [rel exp]
rel  = = | # | < | <= | > | >= | IN
```

It should be noted that, similar to arithmetic operators, there exists a precedence hierarchy among the Boolean operators. ~ has the highest precedence, then follows &, then OR, and last are the relations. As with arithmetic expressions, parentheses may be used freely to make the association of operators explicit. Examples of Boolean expressions are

```
x = y
(x <= y) & (y <Z)
(x > y) OR (y >= Z)
~p OR q
```

Note that a construct such as x<y & z<w is illegal.

The example above draws attention to the rule that the operands of an operator (including relational operators) must be of the same type. The following relations are therefore illegal:

```
   1   = TRUE
   5   = 5.0
i + j = p OR q
```

Also incorrect is e.g. x <= y < z, which must be expanded into (x <= y) & (y < z). However, the following are correct Boolean expressions:

```
it < k - m
p OR q = (i < j)
```

A final hint: although "p = TRUE" is legal, it is considered poor style and better expressed as p. Similarly, replace "p = FALSE" by " p".

7.4   CHAR

Every computer system communicates with its environment via some input and output devices. They read, write, or print elements taken from a fixed set of characters. This set constitutes the value range of the type CHAR. Unfortunately, different brands of computers may use different character sets, which makes communication between them (i.e. the exchange of programs and data) difficult and often tedious. However, there exist internationally standardised sets. Here we refer to ISO's Latin-1 set.

The Latin-1 standard defines a set of 128 characters, 33 of which are so-called (non-printing) control characters. The remaining 95 elements are visible printing characters shown in the following table. The set is ordered, and each character has a fixed position or ordinal number. For example, A is the 66th character and has the ordinal number 65. The ISO

standard, however, leaves a few places open, and they can be filled with different characters according to national desires to establish national standards. Most widely used is the American standard, also called ASCII (American Standard Code for Information Interchange). Here we tabulate the ASCII set. The ordinal number of a character is obtained

	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	'	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
A	lf	sub	*	:	J	Z	j	z
B	vt	esc	+	;	K	[	k	{
C	ff	fs	,	<	L	\	l		
D	cr	gs	-	=	M	]	m	}
E	so	rs	.	>	N	^	n	~
F	si	us	/	?	O	_	o	del

Table 2: Table of ASCII characters

by taking its row number and adding its column number. These numbers are customarily given in hexadecimal form, and we follow this habit here too. The first two columns contain the control characters; they are customarily denoted by abbreviations hinting at their intended meaning. However, this meaning is not inherent in the character code, but

only defined by its interpretation. At this point it suffices to remember that these characters are (usually) not printable.

Constants of type CHAR are denoted by the character being enclosed in quote marks (single- character string). They can be assigned to variables of type CHAR. These values cannot be used in arithmetic operations. Arithmetic operators can be applied, however, to their ordinal numbers obtained from the transfer function ORD(c). Inversely the character with the ordinal number n is obtained with the transfer function CHR(n). These two complementary functions are related by the equations

```
CHR(ORD(c)) = c and ORD(CHR(n)) =n
```

for $0 \leq n < 128$. They permit to compute the numeric value represented by the digit c as

```
ORD(c) - ORD("0")
```

and to compute the digit representing the numeric value n as

```
CHR(n + ORD("0"))
```

These 2 formulas depend on the adjacency of the 10 digits in the ISO character set, where `ORD("0")` = 30H = 48. They are typically used in routines converting sequences of digits into numbers and vice-versa. The following program piece reads digits and assigns the decimally represented number to a variable x.

```
x := 0; Texts.Read(R, c);
```

```
WHILE ("0" <= c) & (c <= "9") DO
   x := 10*x + (ORD(c) - ORD('0"));
   Texts.Read(R, c)
END
```

Control characters are used for various purposes, mainly to control the functions of devices, but also to delineate and structure text. An important role of control characters is to specify the end of a line or of a page of text. There is no universally accepted standard for this purpose. We shall denote the control character signifying a line end by the identifier EOL (end of line); its actual value depends on the system used.

In order to be able to denote non-printable characters, Oberon uses their hexadecimal ordinal number followed by the capital letter X. For example, 0DX is the value of type CHAR denoting the control character cr (carriage return, line end) with ordinal number 13.

7.5 SET

The values which belong to the type SET are sets of integers between 0 and N-1, where N is a constant defined by the computer system used. It is usually the computer's word length or a small multiple of it, typically 32. Constants of this type are denoted as sets. Examples are

```
{2, 3, 5, 7, 11} {0} {8, 15}
```

The notation m..n is a shorthand for m, m+1,... , n-1, n.

```
set = [qualident] "{" [element{, element}] "}"
```

```
element = expression [.. expression]
```

Operations on sets u and v		Set of elements that are in
$+$	union $u + v$	either u or v
$-$	difference $u - v$	u but not v
$*$	intersection $u * v$	both u and v
$/$	symmetric difference u/v	either u or v, but not both

Table 3: Operations on sets

The membership operator IN is regarded as a relational operator. The expression i IN u is of type BOOL. It is TRUE, if i is a member of the set u. Sets are represented in computer systems as sets of bits, i.e. by the characteristic function of the set. The i-th bit of u, for example, is 1, if i is a member of u, 0 otherwise. Hence, the set operators are implemented as logical operations applied to the N members of the set variable. They are therefore very efficient and their execution time is usually even less than that of an addition of integers.

CONSTANT & VARIABLE DECLARATIONS

It has been mentioned previously that all identifiers used in a program must be declared in the program's heading, unless they are standard identifiers known in every program (or are imported from some other module).

If an identifier is to denote a constant value, it must be introduced by a constant declaration which indicates the value for which the constant identifier stands. A constant declaration has the form:

```
ConstantDeclaration = identifier = ConstExpression
ConstExpression = expression
```

A `ConstExpression` is an expression containing constants only. More precisely, its evaluation must be possible by a *mere textual scan* without execution of the program. A sequence of constant declarations is preceded by the symbol `CONST`. Example:

```
CONST N = 16;
     EOL = ODX;
   empty = {};
      M = N - 1;
```

Constants with explicit names aid in making a program readable, provided that the constants are given suggestive

names. If, e.g., the identifier N is used instead of its value throughout a program, a change of that constant can be achieved by changing the program in a single place only, namely in the declaration of N. This avoids the common mistake that some instances of the constant, spread over the entire program text, remain undetected and therefore are not updated, leading to inconsistencies.

A variable declaration looks similar to a constant declaration. The place of the constant's value is taken by the variable's type which, in a sense, can be regarded as the variable's constant property. Instead of an equal sign, a colon is used.

```
VariableDeclaration = IdentList: type
IdentList = identifier {, identifier}
```

Variables of the same type may be listed in the same declaration, and a sequence of declarations is preceded by the symbol VAR. Example:

```
VAR i, j, k: INT;
    x, y, Z: REAL;
         ch: CHAR
```

THE DATA STRUCTURE ARRAY

So far, we have given each variable an individual name. This is impractical, if many variables are necessary that are treated in the same way and are of the same type, such as, for example, if a table of data is to be constructed. In this case, we wish to give the entire set of data a name and to denote individual elements with an identifying number, a so-called *index*. The data type is then said to be structured - more precisely: *array* structured. In the following example, the variable *a* consists of N elements, each being of type INT, and the indices range from 0 to $N - 1$.

```
VAR a: ARRAY N OF INT
```

An element is then designated by the array's identifier followed by its selecting index, e.g. $a[i]$, where i is an expression whose value must lie within the index range specified in the array's declaration. Syntactically, $a[i]$ is a *designator*, and the expression i is the *selector*. If, for example, all N elements of *a* are to be given the value 0, this can be expressed conveniently by a repetitive statement, where the index is given a new value each time.

```
i := 0;
REPEAT a[i] := 0; INC(i) UNTIL i = N
```

INC(i) is synonymous with i := i+1. This example illustrates a situation that occurs so frequently that Oberon provides a special control structure which expresses it more concisely. It is called the *for statement*.

```
FOR i := 0 TO N-1 DO a[i] := 0 END
```

Its general form is

```
ForStatement = FOR id := expr TO expr [BY const] DO
                 StatementSequence
               END
```

The expressions before and after the symbol TO define the range through which the so-called *control variable* (i) progresses. An optional parameter determines the incrementing (decrementing) value. If it is omitted, 1 is assumed as default value.

It is recommended that the for statement be used in simple cases only; in particular, no components of the expressions determining the range must be affected by the repeated statements, and, above all, the control variable itself must not be changed by the repeated statements. The value of the control variable must be considered as undefined after the for statement is terminated.

Some additional examples should demonstrate the use of array structures and the for statement. In the 1st one, the sum of the N elements of array $a[i]$ to be computed:

```
sum := 0:
FOR i:= 0 TO N-1 DO sum := a[i] + sum END
```

In the 2nd example, the minimum value is to be found, and also its index. The repetition's invariant is $min = minimum(a[0], ..., a[i-1])$.

```
min := a[0]; k := 0;
FOR i: = 1 TO N-1 DO
   IF a[i] < min THEN k := i; min := a[k] END
END
```

In the 3rd example, we make use of the 2nd to sort the array in ascending order:

```
FOR i := 0 TO N-2 DO
  min := a[i]; k := i;
  FOR j := i TO N-1 DO
    IF a[j] < min THEN k := j; min := a[k] END
  END;
  a[k] := a[i]; a[i] := min
END
```

The for statement is evidently appropriate, if all elements within a given index range are to be processed. It is inappropriate in most other cases. If, for example, we wish to find the index of an element equal to a given value x, we have no advance knowledge how many elements have to be inspected. Hence, the use of a while (or repeat) statement is recommended. This algorithm is known as *linear search*.

```
i := 0;
WHILE (i < N) & (a[i] # x) DO INC(i) END
```

From the negation of the continuation condition, applying *de Morgan's law*, we infer that upon termination of the while

statement the condition $(i = N)$ `OR` $(a[i] = x)$ holds. If the latter term is `TRUE`, the desired element is found and i is its index; if $i = N$, no $a[i]$ equals x.

We draw attention to the fact that the termination condition is a composite one, and that it is possible to simplify this condition by a common technique. Remember, that repetition must terminate, either if the desired element is found, or if the array's end is reached. The "trick" now consists in marking the end by a sentinel, namely the value x itself, which will automatically stop the search. All that is required is the addition of a dummy element $a[N]$ at the array's end, serving as sentinel:

```
a: ARRAY N+1 OF INT;
```

```
a[N] := x; i := 0;
WHILE a[i] # x DO INC(i) END
```

If upon termination $i = N$, no original element has the value x, otherwise i is the desired index. A more challenging problem is the search for a desired element with value x in an array that is ordered, i.e. $a[i - 1] \leq a[i]$ for all $i = 1 \ldots N - 1$. The best technique is the so-called *binary search*. Inspect the middle element, then apply the same method to either the left or right half. This is expressed by the following piece of program, assuming $N > 0$. The repetition's invariant is

```
a[k] < x for all k = 0 ... i-1 and
a[k] > x for all k = j+1 ... N-1
```

```
i := 0; j := N-1; found := FALSE;
REPEAT mid := (i+j) DIV 2;
```

```
     IF x < a[mid] THEN j := mid-1
   ELSIF x > a[mid] THEN i := mid+1
   ELSE found := TRUE END
 UNTIL (i > j) OR found
```

Because each step halves the interval in which x is searched, the number of needed comparisons is only $\log_2 N$. A somewhat more efficient version which avoids the composite termination condition is

```
 i := 0; j := N-1;
 REPEAT mid := (i+j) DIV 2:
   IF x <= a[mid] THEN j := mid-1 END:
   IF x >= a[mid] THEN i := mid+1 END
 UNTIL i > j:
 IF i > j+1 THEN (*found*) ELSE (*not found*) END
```

An even more sophisticated version is given below. Its ingenious idea is not to terminate immediately when the element is found, which is rare compared to the number of unsuccessful comparisons.

```
 i := 0; j := N;
 REPEAT mid := (i+j) DIV 2:
   IF x < a[mid] THEN j := mid ELSE i := mid+1 END
 UNTIL i >= j:
 IF (j < N) & (a[j] = x) THEN (*found*) ... END
```

This ends our list of examples of typical uses of simple arrays.

The elements of an array are all of the same type, but this type may again be an array (in fact, it may be any struc-

ture, as will be seen later). An array of arrays is called a *multidimensional array* or *matrix*, because each index may be considered as spanning a dimension in a Cartesian space. Examples of 2-dimensional arrays are:

```
a: ARRAY N, N OF REAL
T: ARRAY M, N OF CHAR
```

These are actually abbreviations of the full forms

```
a: ARRAY N OF ARRAY N OF REAL
T: ARRAY M OF ARRAY N OF CHAR
```

The general syntax of an array type is:

```
ArrayType = ARRAY const{, const} OF type
```

where the const denotes the *lengths* of the array.

The syntax of designators allows for a similar abbreviation as used in declarations, namely $a[i, j]$ in place of $a[i][j]$. The latter form expresses more clearly that j is the selector on the array $a[i]$. The syntax for the array element designator is:

```
designator = qualident {"[" ExpList "]"}
ExpList = expression {, expression}
```

In uses of matrices, the for statement comes to its full bloom, particularly in numeric applications. The canonical example

is the multiplication of 2 matrices, where each element of the product $c = a * b$ is defined as:

$$
\begin{aligned}
c[i, j] = \quad & a[i, \quad 0] * b[\quad 0, j] \\
+ \ & a[i, \quad 1] * b[\quad 1, j] \\
+ \ & \dots \\
+ \ & a[i, K{-}1] * b[K{-}1, j]
\end{aligned}
$$

Given the declarations

```
a: ARRAY M, K OF REAL;
b: ARRAY K, N OF REAL;
c: ARRAY M, N OF REAL
```

the multiplication algorithm consists of 3 nested repetitions as follows:

```
FOR i := 0 TO M-1 DO
   FOR j := 0 TO N-1 DO
      sum := 0.0:
      FOR k := 0 TO K-1 DO
         sum := a[i, k]*b[k, j] + sum
      END;
      c[i,j] := sum
   END
END
```

In a 2nd example we demonstrate the search of a word in a table, so-called table search, where each word is an array of characters. We assume the table is declared by the T given in the example above, and that x is given as:

```
x: ARRAY N OF CHAR
```

Our solution employs the typical linear search

```
i := 0; found := FALSE;
WHILE ~found & (i < M) DO
   found := T[i] = x; INC(i)
END
```

If we define equality between 2 words x and y as $x[j] = y[j]$ for all $j = 0\ldots N - 1$, then the "inner" search can be expressed as

```
j = 0; equal := TRUE;
WHILE equal & (j < N) DO
   equal := T[i, j] = x[j]; INC(j)
END;
found := equal
```

This solution appears to be somewhat cumbersome. It is transformable into a simpler form, if $M > 0$ and $N > 0$. The complete table search is then expressible as:

```
i := 0;
REPEAT j := 0;
   REPEAT b := T[i, j] # x[j]; INC(j)
   UNTIL b OR (j = N);
   INC(i)
UNTIL ~b OR (i = M)
```

The result b means "the word x has not been found".

We have now laid enough ground work to develop meaningful, entire programs and shall present 3 examples, all of them involving arrays.

In the 1st example, the goal is to generate a list of powers of 2, each line showing the values 2^i, and 2^{-i}. This task is quite simple, if the type REAL is used. The program then contains the core:

```
d := 1: f := 1.0;
FOR i := 1 TO N DO
  d := 2*d: write(d); (*d = 2^i *)
  write(i);
  f := f/2.0; write(f) (*f = 2^(-i) *)
END
```

However, our task shall be to generate exact results with as many decimal digits as needed. For this reason, we present both the whole number $d = 2^i$ and the fraction $f = 2^{-i}$ by arrays of digits, each in the range $0 \ldots 9$. For f we require N, for d only $\log_{10} N$ digits. Note that the doubling of d proceeds from right to left, the halving of f from left to right. The table of results is shown below:

```
PROC PowersOf2*;
  CONST M = 11; N = 32; (*M ~ N*log(2) *)
  VAR i, j, k, exp, c, r, t: INT;
      d: ARRAY M OF INT;
      f: ARRAY N OF INT;
BEGIN d[0] := 1; k := 1;
  FOR exp := 1 TO N-1 DO
    (* compute d = 2"exp *) c := 0; (*carry*)
    FOR i:=0 TO k-1 DO
      t := 2*d[i] + c;
      IF t >= 10 THEN d[i] := t-10; c := 1
      ELSE d[i] := t; c := 0 END
    END;
```

```
        IF c > 0 THEN d[k] := 1; INC(k) END;
        ("output d[k-1] ... d[0]*) i := M;
        REPEAT DEC(i); Texts.Write(W, "") UNTIL i = k;
        REPEAT DEC(i); Texts.Write(W, CHR(d[i]J+ORD("0")))
        UNTIL i = 0;
        Texts.Writelnt(W, exp, 4);
        (*compute and output f = 2^(-exp) *)
        Texts.WriteString(W, " 0."); r := 0; (*remainder*)
        FOR j := 1 TO exp-1 DO
          r := 10*r + f[j]; f[j] := r DIV 2:
          r := r MOD 2; Texts.Write(W, CHR(f[j]+ORD("0")))
        END;
        f[exp] := 5; Texts.Write(W, "5"); Texts.WriteLn(W)
      END;
      Texts.Append(Oberon.Log, W.buf)
    END PowersOf2.
```

Output of program PowersOf2:

```
        2    1   0.5
        4    2   0.25
        8    3   0.125
       16    4   0.0625
       32    5   0.03125
       64    6   0.015625
      128    7   0.0078125
      256    8   0.00390625
      512    9   0.001953125
     1024   10   0.0009765625
     2048   11   0.00048828125
     4096   12   0.000244140625
     8192   13   0.0001220703125
    16384   14   0.00006103515625
```

32768	15	0.000030517578125
65536	16	0.0000152587890625
131072	17	0.00000762939453125
262144	18	0.000003814697265625
524288	19	0.0000019073486328125
1048576	20	0.00000095367431640625
2097152	21	0.000000476837158203125
4194304	22	0.0000002384185791015625
8388608	23	0.00000011920928955078125
16777216	24	0.000000059604644775390625
33554432	25	0.0000000298023223876953125
67108864	26	0.00000001490116119384765625
134217728	27	0.000000007450580596923828125
268435456	28	0.0000000037252902984619140625
536870912	29	0.00000000186264514923095703125
1073741824	30	0.000000000931322574615478515625
2147483648	31	0.0000000004656612873077392578125
4294967296	32	0.00000000023283064365386962890625

Our 2nd example is similar in nature. Its task is to compute the fractions $d = 1/i$ exactly. The difficulty lies, of course, in the representation of those fractions that are infinite sequences of digits, e.g. $1/3 = 0.333\ldots$ Fortunately, all fractions have a repeating period, and a reasonable and useful solution is to mark the beginning of the period and to terminate at its end. How do we find the beginning and the end of the period? Let us first consider the algorithm for computing the digits of the fraction.

Starting out with *rem* = 1, we repeat multiplying by 10 and dividing the product by i. The integer quotient is the next digit and the remainder is the new value of *rem*. This is precisely the conventional method of division, as illustrated

by the following piece of program and the example with $i = 7$:

```
1.000000 / 7 = 0.142857
1 0
  30
   20
    60
     40
      50
       1


rem := 1;
REPEAT rem := 10 * rem; nextDigit := rem DIV i;
       rem := rem MOD i UNTIL ...
```

We know that the period has ended as soon as a remainder occurs which had been encountered previously. Hence, our recipe is to remember all remainders and their indices. The latter designate the place where the period had started. We denote these indices by x and give elements of x initial value 0. In the above explained division by 7, the values of x are

```
x[1] = 1, x[2] = 3, x[3] = 2,
x[4] = 6, x[5] = 4, x[6] = 5


PROC Fractions*;
  CONST Base = 10; N = 32;
  VAR i, j, m, rem: INT;
      d: ARRAY N OF INT; (*digits*)
      x: ARRAY N+1 OF INT; (*index"*)
BEGIN
  FOR i := 2 TO N DO
```

```
     FOR j := 0 TO i-1 DO x[j] := 0 END;
     m := 0; rem := 1;
     REPEAT m := m+1; x[rem] := m;
       rem := Base * rem; d[m] := rem DIV i;
       rem := rem MOD i
     UNTIL x[rem] # 0;
     Texts.WriteInt(W, i, 6);
     Texts.WriteString(W, " 0.");
     FOR j := 1 TO x[rem]-1 DO
       Texts. Write(W, CHR(d[j]J+ORD("0")))
     END;
     Texts.VWrite(W, "");
     FOR j := x[rem] TO m DO
       Texts.Write(W, CHR(d[j]+ORD("0")))
     END;
     Texts.WriteLn(W)
   END;
   Texts.Append(Oberon.Log, W.buf)
 END Fractions.
```

Output of program Fractions:

```
 2 0.5'0
 3 0.'3
 4 0.25'0
 5 0.2'0
 6 0.1'6
 7 0.142857
 8 0.125'0
 9 0.1
10 0.1'0
11 0.'09
12 0.08'3
```

```
13 0.'076923
14 0.0'714285
15 0.0'6
16 0.0625'0
17 0.'0588235294117647
18 0.0'5
19 0.'052631578947368421
20 0.05'0
21 0.'047619
22 0.0'45
23 0.'0434782608695652173913
24 0.041'6
25 0.04'0
26 0.0'384615
27 0.'037
28 0.03'571428
29 0.'0344827586206896551724137931
30 0.0'3
31 0.'032258064516129
32 0.03125'0
```

Our last example of a program computes a list of prime
numbers. It is based on the idea of inspecting the divisibility
of successive integers. The tested integers are obtained by
incrementing alternatively by 2 and 4, thereby avoiding
multiples of 2 and 3 ab initio. Divisibility needs to be
tested for prime divisors only, which are obtained by storing
previously computed results.

```
PROC Primes∗:
  CONST N = 252; M = 16; (∗M ~ sqrt(N)∗)
      LL = 10; (∗no. of primes placed on a line∗)
  VAR i, k, x, inc, lim, square, L: INT;
```

```
      prime: BOOL; P, V: ARRAY M+1 OF INT;
 BEGIN L := 0; x := 1; inc := 4; lim := 1; square := 9;
   FOR i := 3 TO N DO
     (* find next prime number P[i] *)
     REPEAT x := x + inc; inc := 6 - inc;
       IF square <= x THEN
         INC(lim); V[lim] := square;
         square := P[lim+1] * P[lim+1]
       END;
       k := 2; prime := TRUE:
       WHILE prime & (k < lim) DO INC(k);
         IF V[k] < x THEN V[k] := V[k] + 2*P[k] END;
         prime := x # V[k]
       END
     UNTIL prime;
     IF i<= M THEN P[i] := x END;
     Texts.WriteInt(W, x, 6); L := L+1;
     IF L= LL THEN Texts.WriteLn(W); L := 0 END
   END;
   Texts.Append(Oberon.Log, W.buf)
 END Primes.
```

Output of program Primes:

```
  5    7   11   13   17   19   23   29   31   37
 41   43   47   53   59   61   67   71   73   79
 83   89   97  101  103  107  109  113  127  131
137  139  149  151  157  163  167  173  179  181
191  193  197  199  211  223  227  229  233  239
241  251  257  263  269  271  277  281  283  293
307  311  313  317  331  337  347  349  353  359
367  373  379  383  389  397  401  409  419  421
431  433  439  443  449  457  461  463  467  479
```

487	491	499	503	509	521	523	541	547	557
563	569	S71	577	587	593	599	601	607	613
617	619	631	641	643	647	653	659	661	673
677	683	691	701	709	719	727	733	739	743
751	757	761	769	773	787	797	809	811	821
823	827	829	839	853	857	859	863	877	881
883	887	907	911	919	929	937	941	947	953
967	971	977	983	991	997	1009	1013	1019	1021
1031	1033	1039	1049	1051	1061	1063	1069	1087	1091
1093	1097	1103	1109	1117	1123	1129	1151	1153	1163
1171	1181	1187	1193	1201	1213	1217	1223	1229	1231
1237	1249	1259	1277	1279	1283	1289	1291	1297	1301
1303	1307	1319	1321	1327	1361	1367	1373	1381	1399
1409	1423	1427	1429	1433	1439	1447	1451	1453	1459
1471	1481	1483	1487	1489	1493	1499	1511	1523	1531
1543	1549	1553	1559	1567	1571	1579	1583	1597	1601

These examples show that arrays are a fundamental feature used in most programs. There exist hardly any programs of relevance outside the classroom which do not employ repetitions and arrays (or analogous data structures).

Part II

PRO

PROCEDURES

Consider the task of processing a set of data consisting of a header and a sequence of *N* similar individual units. It might be generally described as

```
ReadHeader;
ProcessHeader;
WriteHeader;
FOR i := 1 TO N DO
  ReadUnit; ProcessUnit:
  Write(i); WriteUnit
END
```

Clearly, the description of the original task has been made in terms of subtasks, emphasizing the dominant structure and supressing details. Of course, the subtasks ReadHeader, ProcessHeader, etc. must now be further described with all the necessary details. Instead of replacing these descriptive English words with elaborate Oberon programs, we may consider these words as identifiers and define the details of the subtasks by textually separate pieces of program, called procedures (or subroutines). These definitions are called procedure declarations, because they define the actions of the procedure and give it a name. The identifiers in the main program referring to these declarations are said to be

procedure calls, and their action is to invoke the procedure. Syntactically, the procedure call is a statement.

Procedures play a fundamental role in program design. They aid in displaying the algorithm's structure and in decomposing a program into logically coherent units. This is particularly important in the case of complex algorithms, i.e. of long programs. In the above example, it might be considered somewhat extravagant to declare separate procedures instead of merely substituting the refined program texts for the identifiers. Nevertheless, the gain in clarity of program structure often recommends the use of explicit procedures even in such a simple case. But of course procedures become particularly useful, if the same procedure is to be invoked at several points of the program.

A procedure declaration consists of the symbol PROC followed by the identifier (together they form the procedure heading), followed by the symbol BEGIN and the statements for which the procedure identifier stands and which are therefore called the procedure body. The declaration is terminated by the symbol END and the repetition of the identifier. The latter enables a compiler to detect mismatched endings of statements and declarations. The general syntax of procedure declarations will be given later. A simple example is the following, which computes the sum of a[0], a[1], ... a[N-1].

```
  PROC Add;
  BEGIN sum := 0.0;
    FOR i := 0 TO N-1 DO sum := a{i} + sum END
  END Add
```

The procedure concept becomes even much more useful due to two additional features that are coupled with it, namely the concepts of parameters and of locality of names. Parameters make it possible to invoke the same procedures at different points of the program applying the procedure to different values and variables as determined at the point of the call. Locality of names and objects is a concept considerably enhancing the procedure's role in structuring a program and compartmentalizing its parts. We shall first discuss the concept of locality.

Summarizing, we repeat the following essential points:

1. The procedure aids in exhibiting the inherent structure of a program and in decomposing a programming task.

2. If a procedure is called from two or more points, it reduces the length of the program and therefore the programming task and the potential for programming error. A further economic advantage is the reduction in the size of the compiled code.

THE CONCEPT OF LOCALITY

If we inspect the preceding example of procedure Add, we notice that the role of the variable *i* is strictly confined to the procedure body. This inherent locality should be expressed explicitly, and can be done by declaring *i* inside the procedure declaration. *i* thereby becomes a local variable.

```
PROC Add;
  VAR i: INT;
BEGIN sum := 0.0;
  FOR i:=0 TO N-1 DO
    sum := a[i] + sum
  END
END Add
```

In some sense, the procedure declaration assumes the form of a separate program. In fact, any declaration possible in a program, such as constant, type, variable, or procedure declaration, may also occur in a procedure declaration. This implies that procedure declarations may be nested and are defined recursively.

It is good programming practice to declare objects local, i.e. to confine the existence of an object to that procedure in which it has meaning. The procedure, i.e. the section of program text in which a name is declared, is called its scope.

Since declarations can be nested, scopes are also nestable. The possibility of having objects local to some scope has several consequences. For example, the same name can be used to denote different objects. In fact, this is a most useful consequence, because a programmer is free to choose local identifiers without knowledge of those existing in the surrounding scope, as long as they do not denote objects used in the local scope (in which case he must obviously be aware of them anyhow). This decoupling of knowledge about different program parts is particularly useful and perhaps even vital in the case of large programs.

The rules of scope (validity of identifiers) are as follows:

1. The scope of an identifier is the procedure in which its declaration occurs, and all procedures enclosed by that procedure, subject to rule 2.

2. If an identifier i declared in a procedure P is redeclared in some inner procedure Q enclosed in P, then procedure Q and all procedures enclosed in Q are excluded from the scope of i declared in P.

3. The standard identifiers of Oberon are considered to be declared in an imaginary scope enclosing the program.

These rules may also be remembered by the algorithm in which the declaration of a given identifier i is searched:

First, search the declarations of the procedure P in whose body i occurs; if the declaration of i is not among them, continue the search in the procedure surrounding P;

Then repeat this same rule until the declaration is encountered.

```
VAR a: INT;
PROC P;
  VAR b: INT;
  PROC Q;
    VAR b, c: BOOL;
  BEGIN
    (* here a, b(BOOL), c are visible *)
  END Q;
BEGIN
  (* here a, b(INT) are visible *)
END P
```

A consequence of the locality concept and of the rule that a variable does not exist outside its scope is that its value is lost when its declaring procedure is terminated. This implies that, when the same procedure is later called again, this value is unknown. The values of local variables are undefined when the procedure is (re)entered. Hence, if a variable must retain its value between two calls, it must be declared outside the procedure. The "lifetime" of a variable is the time during which its declaring procedure is active.

The use of local declarations has 3 significant advantages:

1. It makes clear that an object is confined to a procedure, usually a small part of the entire program.

2. It ensures that inadvertent use of a local object by other parts of the program is detected by the compiler.

3. It enables the implementation to minimize storage, because a variable's storage is released when the procedure to which the variable is local is terminated. This storage can then be reused for other variables.

PARAMETERS

Procedures may be given parameters. They are the essential feature that make procedures so useful. Consider again the previous example of procedure Add. Very likely a program contains several arrays to which the procedure should be applicable. Redefining it for each such array would be cumbersome, inelegant, and can be avoided by introducing its operand as a parameter as follows.

```
PROC Add(VAR x: Vector);
  VAR i: INT;
BEGIN sum := 0:
  FOR i := 0 TO N-1 DO sum := x[i] + sum END
END Add
```

The parameter x is introduced in the parameter list in the procedure heading. It thereby automatically becomes a local object, in fact is a place-holder for the actual array that is specified in the procedure calls

```
Add(a); ... ; Add(b)
```

The arrays a and b are called actual parameters which are substituted for x, which is called formal parameter. The formal parameter's specification must contain its type. This enables a compiler to check whether or not an appropriate

actual parameter is supplied. We say that *a* and *b*, the actual parameters, must be compatible with the formal parameter *x*. In the example above, its type is Vector, presumably declared in the environment of Add as

```
TYPE Vector = ARRAY N OF REAL;
VAR a, b: Vector
```

A better version of Add would include not only the array, but also the result sum as a parameter. We shall later return to this example. But first we need explain that there exist 2 kinds of formal parameters, namely variable and value parameters. The former are characterized by the symbol VAR, the latter by its absence.

We conclude this chapter by giving the syntax of procedure declarations and procedure calls:

```
proc = prop | func
prop = PROC id [params]; decls [BEGIN ss] END id
func = PROC id [params]: type; decls [BEGIN ss]
                                [RETURN expr] END id
params = ([param{; param}])
param  = [VAR] ident{, ident}: {ARRAY OF} type
```

Apart from the declaration of modules, we now have also encountered all forms of declarations.

```
decls = [CONST {const;}] [TYPE {typdef;}]
        [VAR {field;}] {proc;}
```

As its name indicates, the actual parameter corresponding to a formal variable parameter (specified by the symbol VAR) must be a variable. The formal identifier then stands for that variable.

```
PROC exchange(VAR x, y: INT);
   VAR z: INT;
BEGIN z := x; x := y; y := z
END exchange
```

The procedure calls

```
exchange(a, b); exchange(A[i], A[i+1])
```

then have the effect of the above 3 assignments, each with appropriate substitutions made upon the call. The following points should be remembered:

1. Variable parameters may serve to transmit a computed result outside the procedure.

2. The formal parameter acts as a place-holder for the substituted actual parameter.

3. The actual parameter cannot be an expression, and therefore not a constant either, even if no assignment to its formal correspondent is made.

4. If the actual parameter involves indices, these are evaluated when the formal-actual substitution is made.

5. The types of corresponding formal and actual parameters must be the same.

Value parameters serve to pass a value from the calling side into the procedure and constitute the predominant case of parameters. The corresponding actual parameter is an expression, of which a variable or a constant are a particular and simple case. The formal value parameter must be considered as a local variable of the indicated type. Upon call, the actual expression is evaluated and the result is assigned to that local variable. As a consequence, the formal parameter may later be assigned new values without affecting any part of the expression. In a sense, actual expression and formal parameter become decoupled as soon as the procedure is entered. As an illustration, we formulate the previously shown program to compute the power $z = x^i$ as a procedure. Note: Assignments to value parameters are prohibited, if they are structured.

```
PROC ComputePower(VAR z: REAL; x: REAL; i: INT);
BEGIN z := 1.0:
  WHILE i > 0 DO
    IF ODD(i) THEN z := z*x END;
    x := x*x; i := i DIV 2
  END
END ComputePower
```

Possible calls are, for example

procedure call	for
ComputePower(u , 2.5, 3)	$u := 2.5^3$
ComputePower(A[i], B[i], 2)	$A_i := B_i^2$

Note:

1. If a formal parameter is an array or a record and not specified as a variable parameter, then assignments to elements are prohibited.

2. In the example above z and x must be declared in distinct param, because "VAR z, x: REAL" would classify x as a variable parameter too, making it impossible to place a general expression in its corresponding actual parameter place.

12.3   OPEN ARRAY PARAMETERS

lf a formal parameter type denotes an array structure, its corresponding actual parameter must be an array of the identical type. This implies that it must have elements of identical type and the same bounds of the index range. Often this restriction is rather severe, and more flexibility is highly desirable. It is provided by the facility of the so-called *open array* which requires that the types of the elements of the formal and actual arrays be the same, but leaves the index range of the formal array open. In this case, arrays of any size may be substituted as actual parameters. An open array is specified by the element type preceded by ARRAY OF. For example, a procedure declared as

```
PROC P(s: ARRAY OF CHAR)
```

allows calls with character arrays of arbitrary length. The actual length (number of elements) is obtained by calling the standard function LEN(s).

FUNCTION PROCEDURES

So far we have encountered 2 possibilities to pass a result from a procedure body to its calling place: the result is either assigned to a non-local variable or to a variable parameter. There exists a 3rd method: the function procedure. It permits the use of the computed result (as an intermediate value) in an expression. The function procedure identifier stands for a computation as well as for the computed result. The procedure declaration is characterized by the indication of the type of the result following the parameter list. As an example, we rephrase the power computation given above.

```
PROC power(x: REAL; i: INT): REAL;
  VAR z: REAL;
BEGIN z := 1.0:
  WHILE i > 0 DO
    IF ODD(i) THEN z := z*x END;
    x := x*x; i := 1 DIV 2
  END;
  RETURN z
END power
```

Possible calls are

```
u    := power(2.5, 3)
A[i] := power(B[i], 2)
```

```
u     := x + power(y, i+1) / power(z, i-1)
```

The result of the function evaluation is specified as an expression following the symbol RETURN and immediately preceding the symbpl END.

Calls inside an expression are called *function designators*. Their syntax is the same as that of procedure calls. However, a parameter list is mandatory, although it may be empty.

```
ReturnStatement = "RETURN" [expression].
```

We now revisit the previous example of adding the elements of an array and formulate it as a function procedure:

```
PROC sum(VAR a: Vector; n: INT): REAL;
  VAR i: INT; s: REAL;
BEGIN s := 0.0:
  FOR i := 0 TO n-1 DO s := a[i]+s END;
  RETURN s
END sum
```

This procedure, as previously specified, sums the elements $a[0] \ldots a[n-1]$, where n is given as a value parameter, and may be different from (but not larger than!) the number of elements N. A more elegant solution specifies a as an open array, omitting the explicit indication of the array's size.

```
PROC sum(VAR x: ARRAY OF REAL): REAL;
  VAR i: INT; s: REAL;
BEGIN s := 0.0:
  FOR i := 0 TOLEN(x)-1 DO's := x{i] +s END;
```

```
    RETURN s
END sum
```

Obviously, procedures are capable of generating more than
one result by making assignments to several variables. Only
one value, however, can be returned as the result of a func-
tion. This value, moreover, cannot be of a structured type.
Therefore, the other results must be passed to the caller via
VAR parameter or assignment to variables that are not local
to the function procedure. Consider for example the follow-
ing procedure which computes a primary result denoted as
the function's value and a secondary result used to count
the number of times the procedure is called.

```
PROC square(x: INT): INT;
BEGIN INC(n); RETURN x∗x
END square
```

There is nothing remarkable about this example as long
as the secondary result is used for its indicated purpose.
However, it might be misused as follows:

```
m := square(m) +n
```

Here the secondary result occurs as an argument of the ex-
pression containing the function designator itself. The conse-
quence is that e.g. the values square(m)+n and n+square(m)
differ, seemingly defying the basic law of commutativity of
addition.

Assignments of values from within function procedures to
non-local variables are called *side-effects*. The programmer

should be fully aware of their capability of producing un-expected results when the function is used inappropriately. We summarize:

1. A function procedure specifies a result which is used at its place of call as an argument of an expression.

2. The result of a function procedure cannot be structured.

3. If a function procedure generates secondary results, it is said to have side-effects. These must be used with care. It is advisable to use a regular procedure instead, which passes its results via VAR parameters.

4. We recommend to choose function identifiers which are nouns rather than verbs. The noun then denotes the function's result. Boolean functions are appropriately labelled by an adjective. In contrast, regular procedures should be designated by a verb describing their action.

RECURSION

Procedures may not only be called, but can call procedures themselves. Since any procedure that is visible can be called, a procedure may call itself. This self-reactivation is called *recursion*. Its use is appropriate when an algorithm is recursively defined and in particular when applied to a recursively defined data structure.

Consider as an example the task of listing all possible permutations of n distinct objects $a[0]\ldots a[n-1]$. Calling this operation Permute(n), we can formulate its algorithm as follows:

First, keep $a[n-1]$ in its place and generate all permutations of $a[0]\ldots a[n-2]$ by calling Permute(n-1);

Then repeat the same process after having exchanged $a[n-1]$ with $a[0]$.

Continue repeating for all values $i=1\ldots n-2$.

This recipe is formulated as a program as follows, using characters as permuted objects.

```
MODULE Permute;
  IMPORT Texts, Oberon;
  CONST M = 20;
  VAR a: ARRAY M OF CHAR; W: Texts.Writer;
```

```
PROC permute(k: INT);
  VAR i: INT;
  PROC swap(i: INT)
    VAR c: CHAR
  BEGIN c := a[i]; a[i] := a[k]; a[k] := c
  END swap;
BEGIN
  IF k = 0 THEN Texts.WriteString(W, a)
  ELSE permute(k-1);
    FOR i := 0 TO k-1 DO swap(i);
      permute(k-1);      swap(i);
    END
  END
END permute;


PROC Go*; (*command*)
  VAR S: Texts.Scanner;
  PROC len(): INT
    VAR n: INT;
  BEGIN n := 0;
    WHILE a[n] > 0X DO INC(n) END;
    RETURN n
  END len;
BEGIN
  Texts.OpenScanner(S, Oberon.Par.text,
                       Oberon.Par.pos);
  Texts.Scan(S); COPY(S.s, a);
  permute(len()); Texts.WriteLn(W);
  Texts.Append(Oberon.Log, W.buf)
END Go;

BEGIN Texts.OpenWriter(W)
```

```
END Permute.
```

The data generated from the command `Permute.Go ABC` are:

```
 ABC BAC CBA BCA ACB CAB
```

Every chain of recursive calls must terminate at some time, and hence every recursive procedure must place the recursive call within a conditional statement. In the example given above, the recursion terminates when the number of the objects to be permuted is 1. (Concerning input and output conventions, see **??** and **??**).

The number of possible permutations can easily be derived from the algorithm's recursive definition. We express this number appropriately as a function $np(n)$. Given n elements, there are n choices for the element $a[n-1]$, and with each fixed $a[n-1]$ we obtain $np(n-1)$ permutations. Hence the total number $np(n) = n * np(n-1)$. Evidently $np(1) = 1$. The computation of np is now expressible as a recursive function procedure.

```
PROC np(n: INT): INT;
BEGIN
  IF n <= 1 THEN n := 1
  ELSE RETURN n := n * np(n-1)
  END;
  RETURN n
END np
```

We recognize np as the factorial function, defined as $f(n) = 1 * 2 * 3 * \cdots * n$. This formula suggests to program the algorithm using repetition instead of recursion:

```
PROC np(n: INT): INT;
  VAR p: INT;
BEGIN p := 1;
  WHILE n > 1 DO p := n*p; DEC(n) END;
  RETURN p
END np
```

This formulation will compute the result more efficiently
than the recursive version. The reason is that every call re-
quires some "administrative" instructions whose execution
costs time. The instructions representing repetition are less
time-consuming. Although the difference may not be too rel-
evant, it is recommended to employ a repetitive formulation
in place of the recursive one, whenever this is easily possi-
ble. It is always possible in principle; however, the repetitive
version may complicate and obscure the algorithm to such a
degree that the advantages turn into disadvantages. For ex-
ample, a repetitive form of the procedure permute is much
less simple and obvious than the one shown above. To il-
lustrate the usefulness of recursion, 2 additional examples
follow. They typically stem from problems whose solution
is naturally found and explained using recursion.

The 1st example belongs to the class of algorithms which
operate on data whose structure is also defined recursively.
The specific problem consists of converting simple expres-
sions into their corresponding postfix form, i.e. a form in
which the operator follows its operands. An expression shall
here be defined using EBNF as follows:

```
expression = term {(+|-) term}
term = factor {(*|/) factor}
factor = letter | ( expression ) |"[" expression "]"
```

Denoting terms as `T0`, `T1`, and factors as `F0`, `F1`, the rules of conversion are

```
T0 + T1 --> T0 T1 +
To - T1 --> T0 T1 -
F0 * F1 -> F0 F1 *
F0 / F1 -> F0 F1 /
   (E) -> E
   [E] -> E
```

The following program truthfully mirrors the structure of the syntax of accepted expressions. As the syntax is recursive, so is the program. This close mirroring is the best guarantee for the program's correctness. Note also that similarly repetition in the syntax, expressed by the curly brackets, yields repetition in the program, expressed by while statements. No procedure in this program calls itself directly. Instead, recursion occurs indirectly by a call of expression on term, which calls factor, which calls expression. Indirect recursion is obviously much less visible than direct recursion.

This example also illustrates a case of local procedures. Notably, factor is declared local to term, and term local to expression, following the rule that objects should preferrably be declared local to the scope in which they are used. This rule may not only be advisable, but even crucial, as demonstrated by the variables addop (local to term) and mulop (local to factor). If these variables were declared globally, the program would fail. To find the explanation, we must recall the rule that local variables exist (and are given storage) during the time in which their procedure is active. An immediate consequence is that in the case of a recursive call, new incarnations of the local variables are created. Hence,

there exist as many as there are levels of recursion. This also implies that a programmer must ensure that the depth of recursion never becomes inordinately large.

```
MODULE Postfix;
  IMPORT Texts, Oberon:
  VAR c: CHAR; W: Texts.Writer;
               R: Texts.Reader;

  PROC expression;
    VAR addop: CHAR;
    PROC term;
      VAR mulop: CHAR;

      PROC factor;
      BEGIN
        IF c = "(" THEN
          Texts.Read(R, c); expression;
          WHILE c # ")" DO Texts.Read(R, c) END
        ELSIF c = "[" THEN
          Texts.Read(R, c); expression;
          WHILE c # "]" DO Texts.Read(R, c) END
        ELSE
          WHILE (c<"a") OR (c>"z") DO
            Texts.Read(R,c)
          END;
          Texts.Write(W, c)
        END;
        Texts.Read(R, c)
      END factor;
    BEGIN (*term*) factor;
      WHILE (c = "*") OR (c = "/") DO
        mulop := c; Texts.Read(R, c);
```

```
            factor;  Texts.Write(W, mulop)
        END
      END term;
    BEGIN (*expression*) term;
      WHILE (c = "+") OR (c = "-") DO
        addop := c; Texts.Read(R, c);
        term;    Texts.Write(W, addop)
      END
    END expression;


    PROC Parse*;
    BEGIN
      Texts.OpenReader(R, Oberon.Par.text,
                          Oberon.Par.pos);
      Texts.Read(R, c);
      WHILE c # "~" DO
        expression; Texts.WriteLn(W);
        Texts.Append(Oberon.Log, W.buf);
        Texts.Read(R, c)
      END
    END Parse;
  BEGIN Texts.OpenWriter(W);
  END Postfix.
```

A sample of data processed and generated by the command
Postfix.Parse is shown below:


```
a + b                 a b +
a * b + c             a b * c +
a + b * c             a b c * +
a * (b / [c - d])     a b c d - / *
```

The next program example demonstrating recursion belongs to the class of problems search for a solution by trying and testing. A partial "solution" which is once "posted" may, after testing had shown its invalidity, have to be retracted. This kind of approach is therefore also called *backtracking*. Recursion is often very convenient for the formulation of such algorithms.

Our specific example is supposed to find all possible placement of 8 queens on a chess board in such a fashion that none is checking any other piece, i.e. each row, column, and diagonal must contain at most one piece. The approach consists of trying to place a queen in column i, starting with $i = 0$, and, proceeding to the right, assuming that each column to the left contains a correctly placed queen already. If no place is free in column i, the next column to its left has to be reconsidered. The information necessary to deduce whether or not a given square is still free, is represented by the 3 global variables called row, d1, d2 such that:

```
row[j] & d1[i+j] & d2[N-1+i-j] =
"the square in column i and row j is free"
```

Recursion occurs directly in procedure TryCol. The auxiliary procedures PlaceQueen and RemoveQueen could in principle be declared local to TryCol. However, there exists a single chess board only (represented by row, d1, d2), and these procedures are appropriately considered as belonging to these global data, and hence not as local to (each incarnation of) TryCol.

```
MODULE Queens;
  IMPORT Texts, Oberon;
```

```
CONST N = 8; (*no. of rows and columns*)


(*x[i]=j: a queen is on field j in column i*)
VAR x: ARRAY N OF INT;
  row: ARRAY N OF BOOL; (*No queen on i-th: row*)
   d1,          (*diagonals: upleft to lowright*)
   d2: ARRAY 2*N-1 OF BOOL; (*upright to lowleft*)
    W: Texts.Writer;


PROC Clear;
  VAR i: INT;
BEGIN
  FOR i := 0 TO N-1 DO row[i] := TRUE END;
  FOR i := 0 TO 2*(N-1) DO d1[i] := TRUE;
                          d2[i] := TRUE END
END Clear;


PROC WriteSolution;
  VAR i: INT;
BEGIN
  FOR i := 0 TO N-1 DO Texts.WriteInt(W, x[i], 4)
  END; Texts.WriteLn(VW)
END WriteSolution;


PROC PlaceQueen(i, j: INT);
BEGIN
  x[i] := j; row[i] := FALSE;
  d1[i+j] := FALSE; d2[N-1+i-j] := FALSE
END PlaceQueen;


PROC RemoveQueen(i, j: INT);
BEGIN
  row[i] := TRUE; d1[i+j] := TRUE;
```

```
        d2[N-1+i-j] := TRUE
    END RemoveQueen;


    PROC TryCol(i: INT);
      VAR j: INT;
    BEGIN
      FOR j := 0 TO N-1 DO
        IF row[j] & d1[i+j] & d2[N-1+i-j] THEN
          PlaceQueen(i, j);
          IF i < N-1 THEN TryCol(i+1)
          ELSE WriteSolution END;
          RemoveQueen(i, j)
        END
      END
    END TryCol;


    PROC Find*; (*command*)
    BEGIN Clear; TryCol(0);
      Texts.Append(Oberon.Log, W.buf)
    END Find:


BEGIN Texts.OpenWriter(W)
END Queens.
```

TYPE DECLARATIONS

Every variable declaration specifies the variable's type as its constant property. The type can be one of the standard, primitive types, or it may be of a type declared in the program itself. Type declarations have the form

```
TypeDeclaration = identifier "=" type.
```

They are preceded by the symbol TYPE. Types are classified into unstructured and structured types. Each type essentially defines the set of values which a variable of this type may assume. A value of an unstructured type is an atomic unit, whereas a value of structured type has components (elements). For example, the type INT is unstructured; its elements are atomic. It does not make sense, e.g. to refer to the third bit of the value 13; the circumstance that a number may "have a third bit", or a second digit, is a characteristic of its (internal) representation, which intentionally is to remain unknown.

In the following sections we shall show how to declare structured types. We distinguish between various structuring methods of which we have so far encountered the array only. In addition, there exists the record type. A facility to introduce structures that vary dynamically during program

execution is based on the concept of pointers and will be discussed in a separate chapter.

```
type = qualident | ArrayType | RecordType
             | PointerType | ProcedureType.
```

Before proceeding to the various kinds of types, we note that in general, if a type T is declared by the declaration

```
TYPE T = someType
```

and a variable t is declared as

```
VAR t: T
```

then these two declarations can always be merged into the single declaration

```
VAR t: someType
```

However, in this case fs type has no explicit name and therefore remains anonymous. Typically, record types are given explicit names.

The concept of type is important, because it divides a program's set of variables into disjoint classes. Inadvertent assignments among members of different classes can therefore be detected by a mere inspection of the program text without executing the program. Given, for example, the declarations

```
VAR b: BOOL; i: INT; x: REAL
```

the assignments b := i and i := x are illegal, because the types of the variables are incompatible.

RECORD TYPES

In an array all elements are of the same type. In contrast to the array, the record structure offers the possibility to declare a collection of elements as a unit even if the elements are of different types. The following examples are typical cases where the record is the appropriate choice of structuring method. A date consists of three elements, namely day, month, and year. A description of a person may consist of the person's names, sex, identification number, and birthdate. This is expressed by the following type declarations

```
Date = RECORD day,month,year: INT END
Person = RECORD
  firstName,lastName: ARRAY 32 OF CHAR;
  male: BOOL;
  idno: INT;
  birth: Date
END
```

The record structure makes it possible to refer either to the entire collection of data or to individual elements. Elements of a record are also called record fields, and their names are called field identifiers. This stems from the habit of looking at such data as forms or tables drawn on paper with individual fields delineated as rectangles and labelled

with field names. Similar to arrays, where we denote the *i*-th element of an array *a* by $a[i]$, i.e. by the array identifier followed by an index, we denote the field *f* of a record *r* by *r.f*, i.e. by the record identifier followed by the field's name. For example, given the variables

```
d1, d2: Date;
p1, p2: Person;
students: ARRAY 100 OF Person
```

we can construct, for example, the following variable designators:

```
d1.day
d2.month
p1.firstName
p1.birth
```

These examples show that fields may themselves be structured. Similarly, records may be elements of array or record structures, i.e. there exists the possibility to construct hierarchies of structures. As a consequence, selectors of elements can be sequenced, as shown by the following examples of designators. The multi-dimensional array discussed in the chapter on arrays now appears as a particular case of these structuring hierarchies.

```
p1.lastName[0]
p2.birth.year
students[23].idno
students[k].firstName[0]
```

At first sight the record may appear as a generalized array, because it relaxes the restriction that all elements be of the same type. However, in another aspect it is more restrictive than the array: the selector of the element must be a fixed field identifier, whereas the index selecting an array element may be an expression, i.e. a result of previous computations.

It is important to note that a record may assume arbitrary combinations of its field's values. Hence, in the example of the type Date, a value day = 31 may coexist with month = 2, although this is not an actual date.

The syntax of a record declaration is defined as follows.

```
RecordType = RECORD FieldListSequence END.
FieldListSequence = FieldList {; FieldList}.
FieldList = [IdentList: type].
```

and that of a designator is

```
designator = qualident {selector}.
selector = "." identifier | "[" ExpList "]" | "^".
```

Note: Oberon's concept of record type extension is not discussed in this book.

DYNAMIC DATA STRUCTURES AND POINTERS

Array and record structures share the common property that they are static. This implies that variables of such a structure maintain the same structure during the whole time of their existence. In many applications, this is an intolerable restriction; they require data which do not only change their value, but also their composition, size, and structure. Typical examples are lists and trees that grow and shrink dynamically. Instead of providing list and tree structures, a collection that for some applications would again not suffice, Oberon offers a basic tool to construct arbitrary structures. This is the pointer type.

Every complex data structure ultimately consists of elements whose structure is static. Pointers, i.e. values of pointer types, are themselves not structured, but rather are used to establish relationships among those static elements, usually called *nodes*. We also say that pointers link elements or point to elements. Evidently, different pointer variables may point to the same element, hence providing the possibility to compose arbitrarily complex structures, and at the same time opening many pitfalls for programming mistakes that are difficult to pinpoint. Operating with pointers indeed requires utmost care.

Pointers in Oberon cannot point to arbitrary variables. The type of variable to which they point must be specified in the pointer type's declaration, and the pointer type is said to be bound to the referenced object's type. Example:

```
TYPE Node = POINTER TO NodeDescriptor
VAR p0, p1: Node
```

Here Node (and thereby also variables p0 and p1) are bound to the type NodeDescriptor, i.e. they can point to variables of type NodeDescriptor only. These variables, typically of record type, are not created by the declaration of p0 and p1. Instead, they are created by an allocation procedure call. In Oberon, it is represented by a predefined operator called NEW. The statement NEW(p0) then creates a (record) variable of type NodeDescriptor and assigns a pointer referring to that variable (i.e. a value of type Node) to p0. The created variable is said to be dynamically created (allocated); it has no name, is anonymous, and can be accessed only via a pointer using the dereferencing operator ^. The said variable is denoted by the designator p0^. If the referenced variable is of record type with, say, fields x and y, then the fileds are denoted, for example, by p0^.x, p1^.y. Since in this case it is clear that field selection applies to the referenced record rather than to the pointer, the designators may be abbreviated into p0.x and p1.y.

```
PointerType = POINTER TO type.
```

What really makes pointers such a powerful tool is the circumstance that they may point to variables which themselves contain pointers. This reminds us of procedures that

call procedures and thereby introduce recursion. In fact, pointers are the tool to implement recursively defined data structures (such as lists and trees). The nature of the recursive data structure is evident from the declaration of the type of its elements.

Just as every recursion of procedure activation must terminate at some time, so must every recursion in referencing terminate at some point. The role of the if statement to terminate procedural recursion is here taken by the special pointer value NIL terminating referencing recursion. NIL points to no object. It is an obvious consequence that a designator of the form p^ (p^.x, p.x) must never be evaluated, if p = NIL. We summarize the following essential points:

1. Every pointer type is bound to a type; its values are pointers which point to variables of that type.

2. The referenced variables are anonymous and can be accessed via pointers only.

3. The referenced variables are dynamically created by an allocation procedure which assigns the variable's pointer to p.

4. The pointer constant NIL belongs to every pointer type and points to no object.

5. The variable referenced by a pointer p is denoted by the designator p^. In order for p^ to be meaningful, p must not have the value NIL.

6. Field designators of the form p^.x may be abbreviated to p.x.

Lists, also called linear lists or chains, are characterized by consisting of record typed nodes that each have exactly 1 element being a pointer to a record of the same type as itself. This implies recursion. A list pointer declaration then assumes the characteristic form

```
List = POINTER TO ListDesc;
ListDesc = RECORD key: INT;
                 Data: ...
                 next: List
          END
```

"Data" actually stands for any number of fields representing data pertaining to the listed node. Key is part of these data; it is mentioned separately here because it is quite common to associate with each element a unique identifying key, and also because it will be used in subsequent examples of operations on lists. The essential ingredient here, however, is the field next, so labelled because it evidently is the pointer to the next element in the list. Direct recursion in data type declarations is not permitted for the obvious reason that there would be no evident termination. The declaration given above can NOT be abbreviated into

```
List = RECORD key: INT; next: List END
```

Assume now that a list is accessible in a program via its 1st element, denoted by the pointer variable

```
first: List
```

The empty list is represented by first = NIL. A non-empty list is most conveniently constructed by inserting new ele-

ments at its front. The following assignments are needed to insert one element (let *p* be an auxiliary variable of type `List`)

```
NEW(p); (*assign values to p.key and p.data*)
p.next := first; first := p
```

Having constructed a list by repeated insertion of nodes, we may wish to search the list for a node with key value equal to a given *x*. We evidently use a repetition; the while statement is appropriate, because we do not know the number of nodes (and hence repetitions) beforehand. It is wise to include the case of the empty list!

```
p := first;
WHILE (p # NIL) & (p.key # x) DO
  p := p.next
END;
IF p # NIL THEN found END
```

We draw attention to the fact that here we make use of the rule that the term *b* is not evaluated, if in the expression a & b, the factor *a* is found to be `FALSE`. If this rule would not hold, the factor `p.key # x` might be evaluated with `p = NIL`, which is illegal.

The 2nd frequently encountered dynamic data structure is the tree. It is characterized by its nodes having *n* pointer fields each, where *n* is the degree of the tree. The common and in some sense optimal case is the binary tree with $n = 2$. Lists now appear as degenerate trees of degree 1. The respective declarations are

```
Tree = POINTER TO TreeNode;
TreeNode = RECORD key: INT:
                 data: ...
                 left,
                 right: Tree
          END
```

The place of the variable `first` in the case of lists is taken by a variable to be

```
root: Tree
```

with `root = NIL` denoting the empty tree. Trees are commonly used to represent collections of data in order of ascending key values, making retrieval very efficient. The following statements represent a search in an ordered binary tree, whose similarity to the binary search in an ordered array is remarkable. Again, p is an auxiliary variable (of type `Tree`).

```
p := root:
WHILE (p # NIL) & (p.key # x) DO
  IF p.key <x THEN p := p.right
              ELSE p := p.left END
END;
IF p # NIL THEN found END
```

This example is a repetitive version of the tree search. Next we show the recursive version. It is, in addition, extended such that a new node is created and inserted at the appropriate place, whenever no node with key value x exists.

```
PROC search(VAR p: Tree; x: INT): Tree;
```

```
        VAR q: Tree;
    BEGIN
      IF p # NIL THEN
        IF p.key < x THEN
          q := search(p.right, x)
        ELSIF p.key > x THEN
          q := search(p^.left, x)
        ELSE
          q := p
        END
      ELSE (*not found, hence insert*)
        NEW(q); q.key := x;
        q.left:= NIL; q.right := NIL
      END;
      RETURN q
    END search
```

The call search(root, x) now stands for a search of x in the tree represented by the variable root.

And this concludes our examples of operations on lists and trees to illustrate pointer handling. Lists and trees have nodes which are all of the same type. We draw attention to the fact that the pointer facility admits the construction of even more general data structures consisting of nodes of various types. Typical for all these structures is that all nodes are declared as record types. Hence, the record emerges as a particularly useful data structure in conjunction with pointers.

Creation of nodes is expressed by the standard procedure NEW, which is part of a system's storage management. We assume that retrieval of storage is performed automatically

by a so-called storage reclamation mechanism, also called *garbage collector*. It relies on the fact that records that are no longer referenced by any pointer may be recollected, i.e. their storage can be recycled.

PROCEDURE TYPES

So far, we have regarded procedures exclusively as program parts, i.e. as texts that specify actions to be performed on variables whose values are numbers, logical values, characters, etc. However, we may take the view that procedures themselves are objects that can be assigned to variables. In this light, a procedure declaration appears as a special kind of constant declaration, the value of this constant being a procedure. If we allow variables in addition to constants, it must be possible to declare types whose values are procedures. These are called *procedure types*.

A procedure type declaration specifies the number and the types of parameters and, if it is to be a function procedure, the type of the result. For example, a procedure type with a REAL argument and a result of the same type is declared by

```
Function = PROC(x: REAL): REAL
```

Mathematical functions, sine, cosine, square root, exponential and logarithm are all of this type. The general syntax is

```
ProcedureType = PROC [TypeListl]
TypeList = ([[VAR] Type{, [VAR] Type}])[: returnType]
```

If we declare a variable `f: Function`, we can then assign
`f := Math.sin`. Thus, the call `f(x)` is equivalent to `Math.sin(x)`.

It is now also possible to declare functions and procedures
which themselves have functions and procedures as param-
eters. For example, a function procedure to integrate any
function over a certain interval can be expressed in the fol-
lowing way. It adds the values of the given parametric
function over the given interval a to b in n steps.

```
PROC integral(f: Function; a, b: REAL; n: INT): REAL;
  VAR i: INT; sum, dx: REAL;
BEGIN dx := (b-a)/n; sum := 0;
  FOR i:= 0 TO n-1 DO
    sum := f(a + i*dx + 0.5*dx) + sum
  END;
  RETURN sum * dx
END integral
```

The integrations of, for example, the sine function over the
interval 0 to π, and that of the exponential function from 0
to 1 are expressed simply as

integration call	yield	truth
integral(Math.sin, 0, 3.14159, 20)	2.0020	2
integral(Math.exp, 0, 1 , 20)	1.7181	$e - 1$

To conclude, we emphasize the restriction of Oberon: Proce-
dures assigned to variables or passed as parameters,

- must NOT be declared local to any other procedures.

- Neither can they be standard procedures (such as `ODD`,
  `INCL`, etc.).

Part III

MOD

MODULES

Modules are the most important feature distinguishing Oberon from its ancestor Pascal. We have already encountered modules, simply because every program is a module. However, most systems are not only a single module, but rather a set of several modules, each module containing declared objects such as constants, variables, procedures, and types. Objects declared in a module M0 can be referenced in another module M1, if they are explicitly made to be known in M1, i.e. if they are imported by M1. In the examples of the preceding chapters, we have typically imported procedures for input and output from modules containing collections of frequently used procedures. These procedures are actually part of our program, even if we have not programmed them and they are textually disjoint.

The key point is that modules can be kept in a program *library* and are automatically referenced when a programmer's program is loaded and executed. Hence it is possible to prepare collections of frequently used operations (such as for input and output), and to avoid reprogramming them each time a program needs such operations. Modern implementations go even one step further and offer what is called separate compilation. This signifies that such modules are not stored in the program library in source form as Oberon texts, but rather in compiled form. Upon program

loading, the compiled (main) module is joined (linked) with the precompiled modules from which it imports objects. In this case, the compiler must also have access to descriptions of the objects of the previously compiled, imported modules, when the importing program is compiled. This facility distinguishes separate compilation from independent compilation as it exists in typical implementations of Fortran, Pascal, and assemblers.

Every subsidiary module may again import objects from other modules. A program therefore constitutes an entire hierarchy of modules. The main program is said to be at the highest level, those modules which do not import objects at all being at the lowest level. Usually, a programmer is not even aware of this hierarchy, because his modules import objects from modules that he has not programmed himself; therefore he is unaware of their imports and of the module hierarchy below them. In principle, however, his program is the text written by himself, extended with the texts of the imported modules.

These extensions are usually quite large (even if the direct imports consist of a few output procedures only). In principle, the indirect imports constitute the entire environment or OS. In a single-user computer system, there is virtually no need for any parts that are neither directly nor indirectly imported by the main program. However, some modules, such as the basic I/O and file system, may be required by all programs and therefore become de facto resident and may therefore be regarded as the OS.

The principal motivation behind the partitioning of a program into modules is - beside the use of modules provided by other programmers - the establishment of a hierarchy of

abstractions. For example, in the previously encountered cases of imported I/O procedures, we merely wish to have them available, but do not need to know - or rather do not wish to bother to learn - how these procedures function in detail. To abstract means to "take away" from the essentials and thereby to ignore certain details. Each module constitutes an abstraction, if we regard it from "the outside". We even wish to go one step further: we wish not only to ignore the details of its innards, but to hide them. The primary reason for this wish is that if the innards are protected from outside access, we can guarantee their correct functioning, thereby being able to limit the area of error search in the case of a malfunctioning program.

The 2nd, but not less important reason is to make it possible to change (improve) the innards of imported modules without having to change (and/or recompile) the importing modules. This effective *decoupling* of modules is indispensible for the development of large programs, in particular, if modules are developed by different people, and if we regard the OS as the low section of a program's module hierarchy. Without decoupling, any change or correction in an OS or in library modules would become virtually impossible.

A direct consequence of this need for decoupling is the necessity for a textual separation of the essentials from the details. The essentials of a module are the properties of objects that are importable into other modules; the details are those parts that are to be hidden and protected. In Oberon, objects to be visible in other modules are specially marked with an asterisk following their decalaration identifiers.

An importer of a module needs to know the properties of those objects only which are exported by the imported mod-

ule. It is customary and convenient to provide an extract of the module containing those relevant declarations only. We call it the definition of a module. The definition forms a contract between the client (importer) and the designer of a module. It is therefore also called the *interface* of the module. The implementation part remains the property of the module's designer. As long as the designer alters details without affecting the definition, he need not report his activity to the clients of his module. Modules are compiled separately, and are therefore called *compilation units*.

Concluding this introduction to the module concept, we postulate that adequate implementations provide full type compatibility checking between objects, independent of whether these objects are declared in the same or in different modules, i.e. the checking mechanism of the compiler functions also across module boundaries. However, the programmer must realize that this checking provides no absolute safeguard against mistakes. After all, it concerns formal, syntactic aspects only; it does not cover the *semantics*. It would not detect, for instance, the replacement of the algorithm for the sine function by that of the cosine function. However, it must not be regarded as the duty of a compiler to protect programmers against malicious "colleagues".

DEFINITIONS AND IMPLEMENTATIONS

A definition or interface specifies those properties of a module which are relevant to its clients (importers, users). It consists of the declarations of the exported identifiers. We will here consider definitions and implementations as disjoint texts. However, in Oberon there is only one text, namely the module, in which the exported objects are explicitly marked (by an asterisk). The definition may here be considered as an extract of the module text.

Variables declared in a definition are global in the sense that they exist during the entire lifetime of the program, although they are visible and accessible only in those modules (clients) which import the module of their declaration. In other modules they remain invisible. Such variables are typically state variables. But in general, export of variables should be used *rarely*, and if so, preferrably imported variables should be *read only*.

In definitions, procedures are specified by their headings only. A heading specifies the procedure's parameters, i.e. their types, and in the case of function procedures the result type. This information constitutes the procedure's *signature*.

If a type is declared in a definition module, the full details of its declaration are visible in importing modules. In the case

of a record type, the definition specifies only those fields, which are visisble by clients.

The following simple example exhibits the essential characteristics of modules, although typically modules are considerably larger pieces of program and contain a longer list of declarations. This example exports 2 procedures, put and get, adding data to and fetching data from a buffer which is hidden. Effectively, the buffer can only be accessed through these 2 procedures; it is therefore possible to guarantee the buffer's proper functioning, i.e. sequential access without loss of elements.

```
DEFINITION Buffer;
  VAR nonempty, nonfull: BOOL:
  PROC put(x: INT);
  PROC get(VAR x: INT);
END Buffer.
```

This definition part contains all the information about the buffer that a client is supposed to know. The details of its operation, its realization, are contained in the corresponding implementation.

```
MODULE Buffer; (*cyclic buffer of integers*)
  CONST N = 100;
  VAR nonempty*, nonfull*: BOOL;
  in, out, n: INT;
  buf: ARRAY N OF INT;

  PROC put*(x: INT);
  BEGIN
    IF n < N THEN
```

```
      buf[in] := x; in := (in+1) MOD N;
      INC(n); nonfull := n < N; nonempty := TRUE
    END
  END put;


  PROC get*(VAR x: INT);
  BEGIN
    IF n > O THEN
      x := buf[out]; out := (out+1) MOD N;
      DEC(n); nonempty := n > 0; nonfull := TRUE
    END
  END get:


BEGIN (*initialize*) n := 0; in := 0; out := 0;
  nonempty := FALSE; nonfull := TRUE
END Buffer.
```

The definition is the relevant extract from the module listed
here, implementing a fifo (first in first out) queue. This fact,
however, is not evident from the definition alone; normally
the semantics are mentioned in the form of a comment or
other documentation. Such comments will usually explain
what the module performs, but not how this is achieved.
Therefore, different implementations may be provided for
the same definition. The differences may lie in the details of
the mechanism, for example, the buffer might be represented
as a linked list instead of an array (allocating buffer portions
as needed, hence not limiting the buffer's size). Or, the
differences may even lie in the semantics. The following
program implements a stack (i.e. a lifo queue) instead of a
fifo queue, nevertheless fitting the same definition part. Any
change in a module's semantics necessitates corresponding

adjustments in the module's clients, and must therefore be made with utmost care.

```
MODULE Buffer; (*stack of integers*)
  CONST N = 100;
  VAR n: INT;
  nonempty*, nonfull*: BOOL:
  buf: ARRAY N OF INT;

  PROC put*(x: INT);
  BEGIN
    IF n < N THEN
      buf[n] := x; INC(n); nonfull := n < N;
      nonempty := TRUE
    END
  END put:

  PROC get*(VAR x: INT);
  BEGIN
    IF n>O THEN
      DEC(n); x := buf[n]; nonempty := n > 0;
      nonfull := TRUE
    END
  END get:

BEGIN n := 0; nonempty := FALSE; nonfull := TRUE
END Buffer.
```

Evidently, nonempty is the precondition for get, and so is nonfull for put. Let's conclude this example here.

The syntax of modules and definitions:

```
   Definition = DEFINITION identifier:
                {import} {declaration}
             END identifier.
 declaration = CONST {ConstantDeclaration;} |
               TYPE  {identifier[ = type];} |
               VAR   {VariableDeclaration;} |
               ProcedureHeading;
 Module = MODULE identifier;
           [IMPORT IdList] block
         END identifier.
```

The import list names the modules which are imported. If
a client module M1 is a client of M0, i.e. imports M0, then
the objects exported from M0, say *a*, *b*, are referenced by
qualified identifiers of the form M.x, for example M0.a, M0.b.
This facility permits to import different objects with the
same name from different modules and to avoid conflicts of
names. Standard identifiers are automatically imported into
all modules.

The possibility to publicize a module in the form of its
definition and at the same time to retain its operational
details hidden in its implementation specification, is partic-
ularly convenient for the establishment of program libraries.
Such collections of standard routines belong to every pro-
gramming environment. They typically include routines
for IO operations, for file handling, and for the computa-
tion of mathematical functions. Although there exists no
rigid standard for Oberon, the modules Files and Texts
can be considered as standard modules available in all im-
plementations of Oberon (see Chapter **??**). Here we present
the definition of module Math as the 1st example, featuring
standard mathematical functions:

```
DEFINITION Math;
  PROC sqrt  (x: REAL): REAL;
  PROC exp   (x: REAL): REAL;
  PROC ln    (x: REAL): REAL;
  PROC sin   (x: REAL): REAL;
  PROC cos   (x: REAL): REAL;
  PROC arctan(x: REAL): REAL;
  PROC real  (x: INT ): REAL;
  PROC entier(x: REAL): INT;
END Math.
```

DECOMPOSITION INTO MODULES

The quality of a program has many aspects and is an elusive property. A user of a program may judge it according to its efficiency, reliability, or convenience of user dialog. Whereas efficiency can be expressed in terms of numbers, convenience of usage is rather a matter of personal judgement, and all too often a program's usage is called convenient as long as it is conventional. An engineer of a program may judge its quality according to its clarity and perspicuity, again rather elusive and subjective properties. However, if a property cannot be expressed in terms of precise numbers, this is no reason for classifying it as irrelevant. In fact, program clarity is enormously important, and to demonstrate the correctness of a program is ultimately a matter of convincing a person that the program is trustworthy. How can we approach this goal? After all, complicated tasks usually do inherently require complex algorithms, and this implies a myriad of details. And the details are the jungle in which the devil hides.

The only salvation lies in structure. A program must be decomposed into partitions which can be considered one at a time without too much regard for the remaining parts. At the lowest level the elements of the structure are statements, at the next level procedures, and at the highest level modules. In parallel with program structuring proceeds the structur-

ing of data into arrays, records, etc. at the lower levels, and through association of variables with procedures and modules at the higher levels. The essence of programming is finding the right (or at least an appropriate) structure, and the experienced programmer is the person who has the intuition to find it at the stage of initial conception instead of during a gradual process of improvements and modifications. Yet, the programmer who has the courage to restructure when a better solution emerges is still much better off than the one who resigns and elaborates a program on the basis of a clearly inadequate structure, for this leads to those products that no one else, and ultimately not even the originator himself can "understand".

Even if there does not exist a recipe to determine the optimal structure of a program, there have emerged some criteria for guidance in the process of finding good and avoiding bad structure. A basic rule is that decomposition should be such that the connections between partitions are simple or "thin". A perhaps oversimplifying criterion for the thickness of a connection - also called interface - between 2 parts is the number of items that take part in it. Specifically, the interface of 2 modules is sketched in terms of the module's import lists, and a measure for the interface's thickness is the number of imported items. Hence, we must find a modularization which makes the import lists short. Naturally, it is difficult to find an optimum, for, the import lists would be shortest, i.e. they would disappear if the entire program were collapsed into a single module: a clearly undesirable solution.

The distinctive property of the module as the largest structuring unit is its capability to hide details and thereby to

establish a new level of abstraction. This property is used in several forms; we can distinguish between the following cases:

1. The module separates 2 kinds of data representation and contains the collection of procedures that perform the data conversion between the 2 levels. The typical example is a module for conversion of numbers from their abstract, atomic representation into sequences of decimal digits and vice-versa. Such modules contain no data of their own, they are typically packages of procedures.

2. The essence of a module is a set of data. It hides the details of the data representation by granting access to these data through calls of its exported procedures only. An example of this case is a module which contains a data set storing individual items organized in a way that items are found quickly. Another is a module whose hidden data set is a disk store; it hides the peculiar details necessary to operate the disk drive.

3. The module exports a data type and exports its associated operations. Typically, in Oberon such a module exports one or several types in opaque mode (sometimes these are also called private types). It thereby hides the details of the data type's structure and also the details of the operations. By hiding them, it is possible to guarantee the validity of postulated invariant properties of each variable of such a private type. The difference to modules of class 2 is that here variables of the private types are declared in the client modules, whereas in class 2 modules the variable is itself hidden. Typical examples are the queue and stack types, and

perhaps the most successful such data abstraction is the sequential file, also known as a stream.

This classification is not absolute. It cannot be, because all kinds share the common goal of hiding details. Nevertheless, we can formulate a few rules that serve as guidelines in the design of modules:

1. Keep the number of imported modules small.

2. Keep the number of imported modules in definitions even smaller.

3. Export of variables should be considered as the exception, and imported variables should be treated as "read-only" objects.

We conclude this chapter with an example that essentially falls into category 3. Let our stated goal be the design of a cross reference generator for Oberon programs. More precisely, the program's purpose is to read a text and to generate (1) a listing of the text with added line numbers and (2) a table of all encountered words (identifiers) in alphabetical order, each with a list of the numbers of the lines in which the word occurs. Moreover, comments and strings are to be skipped (i.e. their words are not to be listed), and Oberon key symbols are not to be listed either.

We quickly recognize the task as being divisible into the scanning of the source text (eliminating the parts that are to be skipped and ignored), and the recording and subsequent tabulating of the words. The first part is conveniently performed by the main program module, the latter by a subsidiary module which hides the data set and makes it accessible through 2 procedures: Insert (i.e. include a word)

and List (i.e. generate the requested table). A 3rd module
is used to generate the representation of numbers as se-
quences of decimal digits. The 3 principal modules involved
are called XREF, TableHandler, and Texts.

We begin by presenting the main program XREF that scans
the source text. A binary search is used to recognize key
words. The data set is of the type Table, imported from the
TableHandler.

```
DEFINITION TableHandler;
  CONST WordLength; TYPE Word;
  PROC Init (VAR w: Word);
  (*enter pair s,In in structure w *)
  PROC Insert(VAR s: ARRAY OF CHAR; In: INT;
              VAR w: Word);
  PROC List(w: Word)
END TableHandler.

MODULE XREF;
  IMPORT Texts, Oberon, TH := TableHandler;
  CONST N = 32; L = 8; (*keywords #, maxlen*)
  VAR ln: INT; (*current line number")
      Tab: TableHandler.Word; W: Texts.Writer;
      key: ARRAY N, L OF CHAR; (*table of keys*)

  PROC heading;
  BEGIN INC(ln); Texts.WriteInt(W, ln, 5);
                 Texts.Write(W, 20X)
  END heading;

  PROC Scan*; (*commana'*)
    VAR beg, end, time: LONGINT;
```

```
               k, m, l, r: INT; c: CHAR;
               id: ARRAY TH.WordLength OF CHAR;
               T: Texts. Text; R: Texts.Reader;
          PROC copy;
          BEGIN Texts.Write(W, c); Texts.Read(R, c);
          END copy;
        BEGIN TableHandler.Init(Tab);
          Oberon.GetSelection(T, beg, end, time);
          IF time >= 0 THEN ln := 0; heading;
            Texts.OpenReader(R, T, beg);
            Texts.Read(R, c);
            REPEAT
              IF (CAP(c)>="A") & (CAP(c)<="Z") THEN (*word*)
                k := 0;
                REPEAT id[k] := c; INC(k); copy
                UNTIL ~(("A"<=CAP(c)) & (CAP(c)<="Z")
                      OR ("a" <= c) & (c <= "z"));
                id[k] := OX;
                l := 0; r := N; (*binary search for key*)
                REPEAT m := (l+r) DIV 2;
                  IF key[m] < id THEN l := m + 1
                                 ELSE r := m END;
                UNTIL l >= r;
                IF (r = N) OR (id # key[r]) THEN
                  TableHandler.Insert(id, ln, Tab)
                END
              ELSIF (c >= "0") & (c <= "9") THEN (*number*)
                REPEAT copy UNTIL ~("0"<=c) & (c<="9")
              ELSIF c = "(" THEN copy;
                IF c="*" THEN (*skip comment*)
                  REPEAT
                    REPEAT IF c=0DX THEN heading END; copy
                    UNTIL c = "*"; copy
```

```
                UNTIL c = ")"; copy
              END
          ELSIF c = 22X THEN (*string*)
            REPEAT copy UNTIL c = '"'; copy
          ELSIF c = 0DX THEN (*end of line*)
            copy; heading
          ELSE copy
          END
        UNTIL R.eot; Texts.WriteLn(W);
        Texts.Append(Oberon.Log, W.buf);
        TableHandler.List(Tab)
      END
    END Scan;


  BEGIN Texts.OpenWriter(W);
    key[ 0] := "AND"   ; key[ 1] := "ARRAY"  ;
    key[ 2] := "BEGIN" ; key[ 3] := "BOOL"   ;
    key[ 4] := "BY"    ; key[ 5] := "CASE"   ;
    key[ 6] := "CONST" ; key[ 7] := "DIV"    ;
    key[ 8] := "DO"    ; key[ 9] := "ELSE"   ;
    key[10] := "ELSIF" ; key[11] := "END"    ;
    key[12] := "FOR"   ; key[13] := "IF"     ;
    key[14] := "IMPORT"; key[15] := "IN"     ;
    key[16] := "MOD"   ; key[17] := "MODULE" ;
    key[18] := "NOT"   ; key[19] := "OF"     ;
    key[20] := "OR"    ; key[21] := "POINTER";
    key[22] := "PROC"  ; key[23] := "RECORD" ;
    key[24] := "REPEAT"; key[25] := "RETURN" ;
    key[26] := "THEN"  ; key[27] := "TO"     ;
    key[28] := "TYPE"  ; key[29] := "UNTIL"  ;
    key[30] := "VAR"   ; key[31] := "WHILE"
  END XREE.
```

Next we present the table handler. As seen from its definition part, it exports the private type Table and its operations Insert and List. Notably the structure of the tables, and thereby also the access and search algorithms remain hidden. The 2 most likely choices are the organizations of binary trees and of a hash table. Here we opt for the former. The example is therefore a further illustration of the use of pointers and dynamic data structures. The module contains a procedure to search and insert a tree element, and a procedure that traverses the tree for the required tabulation (see also **??** on dynamic data structures). Each tree node is a record with fields for the key, the left and right descendants, and (the head of) a list of the line numbers. Another form of representation might be chosen, for example a balanced tree, and such a new implementaion might be provided without clients being aware of the change, because modules can be compiled separately. This is the key of modularization and constructing large systems. It requires, however, that the modules' interfaces are wisely chosen.

```
MODULE TableHandler;
  IMPORT Texts, Oberon;
  CONST WordLength* = 32;
  TYPE Item = POINTER TO RECORD
                num: INT; next: Item
              END;
      Word* = POINTER TO RECORD
                 key: ARRAY WordLength OF CHAR;
                 first: Item; ("list head*)
                 left, right: Word
              END;
  VAR W: Texts.Writer;
```

```
PROC Init∗(VAR w: Word);
BEGIN w := NIL
END Init;


(“search node with name s∗)
PROC Insert∗(VAR s: ARRAY OF CHAR; ln: INT; VAR w: Word);
  VAR t: Item;
BEGIN
  IF w = NIL THEN (∗insert new word∗)
    NEW(t); t.num := ln; t.next := NIL;
    NEW(w); w.first := t; w.left  := NIL;
    COPY(s, w.key);       w.right := NIL;
  ELSIF s < w.key THEN Insert(s, ln, w.left)
  ELSIF s > w.key THEN Insert(s, ln, w.right)
  ELSE NEW(t); t.num := ln;
    t.next := w.first; w.first := t
  END
END Insert;


PROC write(w: Word);
  VAR t: Item;
BEGIN Texts.WriteString(W, w.key);
  t := w.first;
  REPEAT Texts.VWriteInt(W, t.num, 5); t := t.next
  UNTIL t = NIL; Texts.WriteLn(W)
END write;


PROC List∗(w: Word);
BEGIN IF w # NIL THEN List(w.left); write(w);
                    List(w.right) END;
  Texts.Append(Oberon.Log, W.buf)
END List;
```

```
BEGIN Texts.OpenWriter(W)
END TableHandler.
```

THE CONCEPT OF A SEQUENCE

22.1 ABOUT IO

The usefulness and the success of high-level programming languages rests on the principle of abstraction, the hiding of details that pertain to the computer which is used to execute the program rather than to the algorithm expressed by the program. The domain that has most persistently relied on abstraction is that of IO operations. This is not surprising, because IO inherently involve the activation of devices that are peripheral to the computer, and whose structure, function, and operation differ strongly among various kinds and brands. Many programming languages have typically incorporated statements for reading and writing data in sequential form without reference to specific devices or storage media. Such an abstraction has many advantages, but there exist always applications where some property of some device is to be utilized that is poorly, if at all, available through the standard statements of the language. Also, generality is usually costly, and consequently operations that are conveniently implemented for some devices may be inefficient if applied to other devices. Hence, there also exists a genuine desire to make transparent the properties of some devices for applications that require their efficient utilization. Simplification and generalization by suppres-

sion of details is then in direct conflict with the desire for transparency for efficient use.

Oberon has resolved (or rather circumvented) this intrinsic dilemma, by not including any IO statements at all. This somehow extreme approach is acceptable for 2 facilities:

1st, there is the module structure allowing the construction of a hierarchy of (library) modules representing various levels of abstraction.

2nd, Oberon permits the expression of computer specific operations, such as communication with peripheral interfaces.

These operations are typically contained in modules at the lowest level of this hierarchy, and are therefore counted among the so-called low-level facilities.

- A program wishing to ignore the details of device handling imports procedures from the standard modules at the higher levels of this hierarchy.

- When desiring utmost efficiency or requiring access to specific properties of specific devices, one either uses low-level modules, so-called device drivers, or uses the primitives themselves.

In the latter case, the programmer pays the price of intransportability, for his programs refer directly to particulars of either his computer or its OS.

It is impossible to present in this context any operations of devices at the low levels of the module hierarchy, because there exists a wide variety of such devices. We restrict the

following material to the presentation of 2 typical modules used in performing conventional IO operations. Module `Texts` we have already encountered in examples in preceding chapters. The main concept to be presented is the file, data considered as sequences of elements of the same type. For this purpose, we present the definition of module `Files`.

We must generally distinguish between legible and illegible IO. Legible IO serves to communicate between the computer and its user. Mostly the data elements are characters i.e. values of type `CHAR`; the exception is graphical IO. Legible data are input through keyboards, scanners, etc. Legible output is generated by displays and printers. Illegible IO is made from and to so-called peripheral storage media, such as disks and tapes, but also may originate from sensors - e.g. in laboratories or drawing offices - and to devices that are controlled by computers, such as plotters, factory assembly lines, traffic signals, and networks. Data for illegible IO can be of any type, and need not be of type `CHAR`.

The vast majority of IO operations of both the legible and illegible variety is appropriately considered as *sequential*. Their data are of a structure that does not exist as a basic data structuring method in Oberon, such as the array or the record. Sequences have the following characteristics:

1. All elements are of the same type.

2. The number of elements (called the *length* of the sequence) is not known a priori. The sequence is therefore (a simple case of) a dynamic structure.

3. A sequence can be modified only by appending elements at its end, called *writing*.

4. Only a single element of a stream is visible (accessible) at any one time. Accessing this element is called *reading*, and the sequence is read by advancing from one element to the next.

In passing we state that the sequence as described above is perhaps the most successful case of data abstraction encountered. It is certainly more widely used in actual practice than the often quoted examples of stacks and queues. The language Pascal had included it among its basic data structuring methods along with arrays, records, and sets.

Before proceeding with the postulation and explanation of modules for handling sequences, which we will call *files* and *texts*, we wish to point out an important separation of function performed for legible IO.

> On the one hand, there is the actual transport of data to and from the computer. This involves the activation and sensing of the state of the peripheral device, be it a keyboard, a display, or a printer.

> On the other hand, there is the function of transforming the representation of data.

If, for example, the value of an expression of type INT is to be transmitted to a display, the computer-internal representation must be transformed into the decimal representation as a sequence of digits. The display device then translates the character representation (usually consisting of 8 bits for each character) into a pattern of visible dots or lines. However, the former translation can be considered as device independent, and is therefore a prime candidate for separation from device specific operations. It can be performed by

the same routines without regard whether the sequence is to be stored on a disk or to be made visible on a display.

A 3rd class of functions that can well be separated from physical data transfer and from conversion of data representation, pertains to devices associated with more than a single sequence, the primary example being a disk store. We refer to the operations of allocating storage space and associating names with individual texts or files. Considering that texts and files are dynamic structures, storage allocation is of considerable complexity. The naming of individual files and in particular the management of directories (in order to quickly locate individual files) is another task requiring an elaborate mechanism. Both storage allocation and directory management are the tasks of individual service modules. There seem to be as many ways to manage these tasks as there are OSes. And this is precisely where diversity transcends the many levels of IO operations. We therefore strictly adhere to and confine ourselves to the abstract notion of a sequence.

22.2 FILES AND RIDERS

Elements of a sequence cannot be identified by an index, as in the case of arrays, nor by a field name, as in the case of a record. Elements are instead accessed one by one, advancing strictly sequentially. This notion itself implies the postulation of an access machanism. In the Oberon module `Files`, we call it `Rider`. A rider is a data structure, an object, which can be placed on a file at a given position, and then be used to write or read single elements. Hence, `Files` defines not only 1, but 2 types:

1st, `File` stands for the data, and

2nd, Rider, for the operations performed.

It now follows, that several riders may be positioned on a same file and be moved independently. Not the file but each rider has a reading/writing position. However, normally only 1 is used.

```
DEFINITION Files;
  TYPE File;
  Rider = RECORD eof: BOOL; res: LONGINT END;

  PROC Old(name: ARRAY OF CHAR): File;
  PROC New(name: ARRAY OF CHAR): File;
  PROC Register(f: File);
  PROC Close   (f: File);
  PROC Purge   (f: File);
  PROC Length  (f: File): LONGINT;
  PROC GetDate (f: File; VAR t, d: LONGINT);

  PROC Set (VAR r: Rider; f: File; pos: LONGINT);
  PROC Pos (VAR r: Rider): LONGINT;
  PROC Base(VAR r: Rider): File   ;
  PROC Read       (VAR r: Rider; VAR c: CHAR     );
  PROC Readint    (VAR r: Rider; VAR x: INT      );
  PROC ReadReal   (VAR r: Rider; VAR x: REAL     ):
  PROC ReadLReal  (VAR r: Rider; VAR x: LONGREAL);
  PROC ReadString(VAR r: Rider; VAR x: ARRAY OF CHAR);

  PROC Write       (VAR r: Rider; c: CHAR);
  PROC Writelnt    (VAR r: Rider; x: INT );
  PROC WriteReal   (VAR r: Rider; x: REAL);
  PROC WriteString(VAR r: Rider; x: ARRAY OF CHAR);
```

```
    PROC Delete(   name: ARRAY OF CHAR; VAR res: INT);
    PROC Rename(old,new: ARRAY OF CHAR; VAR res: INT);
  END Files.
```

The procedures are listed in 4 groups:

1st group operates on files,

$2^{nd}, 3^{rd}$ on riders for reading and writing, respectively, and

4th on the file directory only.

We assume that files are stored on a persistent medium, such as a magnetic disk or flash-rom, and that all files are listed in a directory with names. In the

1st group, procedure Old yields the file listed in the directory by the specified name. New generates a new (empty) file with specified name. Actual registration in the directory is performed by procedure Register, typically after the entire file had been generated. Close terminates the writing of a file (flushing buffers), and it is implied in Register. GetDate yields the date and time of a file's creation.

2nd group, Set places a rider on a file at a specified position (between 0 and the length of the file). Pos indicates the rider's current position, and Base the file on which it is placed. The Read procedures read a specified type of data element (consisting of one or several bytes) without any transformation of representation. They advance the rider by the appropriate amount. A string is assumed to be terminated by a 0X character.

3rd group, procedures operate similarly for generating (writing) a file. Note that the rider must not necessarily be positioned at the end of the file, although this is the normal, prevalent case.

We are now in a position to show, how files are typically written and read in Oberon. Let us assume the declarations

```
f: Files.File; r: Files.Rider
```

1st, the empty file is created,

then, a rider is associated with it.

We assume it is generated sequentially (here in a while-loop), and

finally (and optionally) it is registered in the directory.

```
f := Files.New("MyFile");
Files.Set(r, f, 0);
WHILE more DO compute(next);
              Files.Write(r, next) END;
Files.Register(f)
```

The file can thereafter be read by the following pattern:

1st, the specified name file is associated with variable f.

Then a rider r is placed at the start of f, and

then advanced by reading.

```
f := Files.Old("MyFile"); Files.Set(r, f, 0):
WHILE ~r.eof DO Files.Read(f,next); process(next) END
```

Note that the rider indicates that the end of the file had been reached after the 1st unsuccessful attempt of reading.

22.3 TEXTS, READERS AND WRITERS

`Texts`, basically, are sequences of characters. In the Oberon System, however, texts have some additional properties. As they can be displayed and printed, they must carry properties which determine their style and appearance. In particular, a font is specified. This is a style, the patterns by which characters are represented. Subsequences of charcaters in a text may have their individual fonts. Furthermore, Oberon texts specify color and vertical offset (allowing for negative offset for subscripting, and positive offset for superscripting). Texts are typically subjected to complicated editing operations, which require a flexible, internal data representation. Therefore `Texts` in Oberon differ substantiall from `Files` in many respect. However, in their essence they are also sequences, and the basic operations strongly resemble those of files.

Our special interest in texts is justified by the fact, that humans communicate with computers mostly via texts, be they typed on a keyboard or displayed on a screen. As a consequence, reading and writing of texts usually includes a change of data representation. For example, integers will have to be changed into sequences of decimal digits on output, and the reading of integers requires the reading of a sequence of digits and the computation of the represented value. These conversions are included in the read/write procedures of module `Texts`, which we have used in many examples in preceding chapters. The following is its definition, which evidently resembles that of `Files`. In place

of the single type `Rider` we find the 2 types `Reader` and
`Writer`.

```
DEFINITION Texts;
  IMPORT Files, Fonts:
  CONST replace = 0; insert = 1; delete = 2;
        Inval = 0; Name = 1; String = 2; Int = 3;
        Real = 4; LongReal = 5; Char = 6;
  TYPE Text = POINTER TO RECORD len: LONGINT END;
       Reader = RECORD eot: BOOL; fnt: Fonts.Font;
                       col, voff: SHORTINT END;
       Scanner = RECORD (Reader) nextCh: CHAR;
                   line, class: INT: i: LONGINT;
                   x: REAL; y: LONGREAL;
                   c: CHAR: len: SHORTINT;
                   s: ARRAY 32 OF CHAR;
                 END; Buffer;
       Writer = RECORD buf: Buffer; fnt: Fonts.Font;
                       col, voff: SHORTINT END;


  PROC Open       (t: Text; name: ARRAY OF CHAR);
  PROC Delete     (t: Text; beg, end: LONGINT);
  PROC Insert     (t: Text; pos:LONGINT; b:Buffer);
  PROC Append     (t: Text; b: Buffer);
  PROC ChangeLooks(t: Text; beg, end: LONGINT;
                   sel: SET; fnt: Fonts.Font;
                   col, voff: SHORTINT);


  PROC OpenReader(VAR r: Reader; t: Text; pos: LONGINT);
  PROC Read       (VAR r: Reader; VAR c: CHAR);
  PROC Pos        (VAR r: Reader): LONGINT;


  PROC OpenScanner(VAR S: Scanner; t: Text; pos: LONGINT);
```

```
    PROC Scan          (VAR S: Scanner);

  PROC OpenWriter    (VAR w: Writer);
  PROC SetFont       (VAR w: Writer; fnt: Fonts.Font);
  PROC SetColor      (VAR w: Writer; col: SHORTINT);
  PROC SetOffset     (VAR w: Writer; off: SHORTINT);
  PROC Write         (VAR w: Writer; c: CHAR);
  PROC WriteLn       (VAR w: Writer);
  PROC WriteString   (VAR w: Writer; s: ARRAY OF CHAR);
  PROC Writelnt      (VAR w: Writer; x, n: LONGINT);
  PROC WriteHex      (VAR w: Writer; x: LONGINT);
  PROC WriteReal     (VAR w: Writer; x: REAL;     n: INT);
  PROC WriteRealFix  (VAR w: Writer; x: REAL;  n, k: INT);
  PROC WriteLongReal (VAR w: Writer; x: LONGREAL; n: INT);
  PROC WriteDate     (VAR w: Writer; t, d: LONGINT);

  PROC Load (VAR r: Files.Rider; t: Text);
  PROC Store(VAR r: Files.Rider; t: Text)
END Texts.
```

Procedure OpenWriter uses as defaults a standard font, black color, and zero offset.

Simple, sequential reading and writing of a text now follow the patterns of reading and writing a file. Let us assume the following declarations of variables:

```
  T: Texts.Text;
  R: Texts.Reader;
  W: Texts.Writer;

  NEW(T); Texts.Open(T, "MyName");
  Texts.OpenReader(R, T, 0); Texts.Read(R, ch);
```

```
        WHILE ~R.eot DO process(ch); Texts.Read(R, ch) END;


        NEW(T); Texts.Open(T, "MyName");
        Texts.OpenWriter(W); generate(ch);
        WHILE more DO Texts.Write(W, ch); generate(ch) END;
        Texts.Append(T, W.buf)
```

Note that the Oberon paradigm is to write a text, or a piece
of text, first into a buffer, and thereafter insert or append
it to a text. This is done for reasons of efficiency, because
the possibly needed rendering of the text, for example on
a display, can be done once upon insertion of the buffered
piece of text rather than after generating each character.

Very often one wants to read a text consisting of a sequence
of items which are not all of the same type, such as numbers,
strings, names, etc. When using procedures reading items of
a fixed type, the programmer must know the exact sequence
in which the various items will appear in the text. Typically
one does not know, and even if a specific order is specified,
mistakes may occur. So one should like to have available a
reading mechanism that reads items one at a time, but lets
the program determine what type of item was read, and take
further steps accordingly. This facility is provided in the
Oberon text module by the machanism called *scanning*. In
place of a `Reader` we use a `Scanner`. Procedure `Scan(S)` then
reads the next item. Its kind can be determined by inspecting
the field `S.class`, and its value accordingly is given by `S.l`
in the case an integer was read, `S.x` if a real number was
read, `S.s` if a name or a string was read. This scheme has
proven to be most useful because of its flexibility. It defines
the following syntax for texts to be scanned. Blanks and line
ends serve to separate consecutive items.

```
item = name | integer | real | longreal | string
            | SpecialChar
name = letter {letter | digit}
sign = [-]
decimal = [sign] digit {digit}
hexdig = A | B | C | D | E | F
integer = decimal | digit {digit | hexdig} H
real = decimal . digit {digit} [(E|D) decimal]
```

A string is any sequence of any characters (except quotes) enclosed in quotes. Special characters are all characters except letters, digits, and the quote mark.

22.4 STANDARD INPUT AND OUTPUT (IO)

In order to simplify the description of IO of texts in simple cases, the Oberon system introduces some conventions and global variables. We call this source of input and sink of output *standard IO*.

Standard output sink is the text `Log` defined as global variable in module Oberon. Given a global writer `W`, the text, previously written by `Write` procedures into the writers buffer (`W.buf`) is sent to the displayed `Log` text by the statement

```
Texts.Append(Oberon.Log, W.buf)
```

The standard input is often assumed to be the text following the command which invoked execution of a given procedure. This input text is identified by the variable `Par` in module `Oberon`. It is considered as parameter of the invoked com-

mand, and it is typically read by the scanning mechanism described above. The necessary statements are:

```
Texts.OpenScanner(S, Oberon.Par.text,
                      Oberon.Par.pos);
Texts.Scan(S)
```

The 1st item thus read may be the 1st item of the desired text, or it may be the name of the text file to be scanned, or something else, according to the specification of the individual command. The following conventions have established themselves for the designation of input texts. Assume that the command (procedure) name is followed by:

- an identifier (possibly qualified). Then this is the name of the input text (input file),

- an asterisk (*). Then the marked viewer (window) contains the input text,

- an @ symbol. Then the most recent text selection is the beginning of the input text,

- an arrow (↑). Then the most recent selection denotes the file name of the input text.

These conventions are expressed by the following function yielding the designated input text:

```
PROC This*(): Texts.Text;
  VAR beg, end, time: LONGINT; S: Texts.Scanner;
      T: Texts.Text: v: Viewers.Viewer;
BEGIN Texts.OpenScanner(S, Oberon.Par.text,
                            Oberon.Par.pos);
```

```
    Texts.Scan(S);
    IF S.class = Texts.Char THEN
      IF S.c="*" THEN (*input in marked viewer*)
        v := Oberon.MarkedViewer();
        IF (v.dsc # NIL) &
           (v.dsc.next IS TextFrames.Frame) THEN
          T := v.dsc.next(TextFrames.Frame).text;
          beg := 0
        END
      ELSIF S.c="@" THEN (*input starts at selection*)
        Oberon.GetSelection(T, beg, end, time);
        IF time < 0 THEN T := NIL END
      ELSIF S.c = "↑" THEN (*selection is the file name*)
        Oberon.GetSelection(T, beg, end, time);
        IF time >= 0 THEN (*there is a selection*)
          Texts.OpenScanner(S, Oberon.Par.text,
                              Oberon.Par.pos);
          Texts.Scan(S);    (*input is named file*)
          IF S.class = Texts.Name THEN
            NEW(T); Texts.Open(T, S.s); pos := 0
          END
        END
      END
    ELSIF S.class = Texts.Name THEN (*input is named file*)
      NEW(T); Texts.Open(T, S.s); pos := 0
    END;
    RETURN T
  END This;
```

Let us assume that this function procedure is defined in
module Oberon, together with the global variable pos, indi-
cating the position in the text where input is to start. The
following program for copying a text may serve as a pat-

tern for selecting the input according to the conventions postulated above.

```
MODULE ProgramPattern;
  IMPORT Texts, Oberon:
  VAR W: Texts.Writer; (*global writer*)

  PROC copy(VAR R: Texts.Reader):
    VAR ch: CHAR;
  BEGIN Texts.Read(R, ch);
    WHILE ~R.eot DO Texts.Write(W, ch);
                    Texts.Read(R, ch) END
  END copy

  PROC SomeCommand*:
    VAR R: Texts.Reader; (*local reader*)
  BEGIN Texts.OpenReader(R, Oberon.This.text,
                            Oberon.Par.pos);
    copy(R, W); Texts.Append(Oberon.Log, W.buf)
  END SomeCommand;
BEGIN Texts.OpenWriter(W)
END ProgramPattern.
```

Module Oberon can be considered as the core of the Oberon system housing a small number of global resources. These include the previously encountered output Log, and the record Par, specifying the viewer, frame, and text in which the activated command lies, and hence providing access to its input parameters. Here we show only an excerpt of the definition of module Oberon:

```
TYPE ParList = POINTER TO RECORD
                   vwr: Viewers. Viewer;
```

```
                    frame: Display.Frame;
                    text: Texts. Text:
                    pos: LONGINT;
                END;
VAR Log: Texts.Text: Par: ParList;
PROC Time(): LONGINT;


(*provide coordinates X, Y
               for a new viewer to be allocated*)
PROC AllocateUserViewer  (DX: INT; VAR X,Y: INT);
PROC AllocateSystemViewer(DX: INT; VAR X,Y: INT);


PROC GetSelection(VAR            text: Texts.Text;
                     VAR beg, end, time: LONGINT);
```

To conclude this introduction to Oberon conventions about
IO, we show how a new viewer (window) is opened, and
how the output text is directed into this text viewer. Optimal
positioning of the new viewer is achieved by procedure
Oberon.AllocateViewer, with the specified set of standard,
frequently needed commands in the title bar.

```
PROC OpenViewer(T: Texts.Text);
  VAR V: MenuViewers.Viewer; X, Y: INT;
BEGIN
  Oberon.AllocateUserViewer(Oberon.Mouse.X, X, Y);
  V := MenuViewers.New(TextFrames.NewMenu("Name",
        "Sys.Close .Copy .Grow Ed.Search .Store"),
                      TextFrames.NewText(T, 0),
                      TextFrames.menuH, X, Y
  )
END OpenViewer;
```

Part IV

OOP

OBJECT-ORIENTED PROGRAMMING (OOP)

23.1 ORIGINS OF OOP

The term OOP originated around 1975. It expressed a shift in the paradigm of programming.

- In the conventional, *procedural* programming style, the algorithm stands in the foreground of the programmer's concerns, the algorithm operating on a set of data.

- In the OO view, the datum on which algorithms are applied stand in the foreground, They are grouped into what is called an *object*.

Programming is a notoriously difficult task, and the rapidly increasing power of modern computers made the situation worse. The tasks to be modeled and solved became more and more complicated. Their complexity often surpassed the capabilities of the human mind. In this situation, it is not surprising that the demand for better tools, for a new discipline, even for panaceas became urgent and ardent. The new paradigm of OOP was therefore welcomed, as their proponents raised hopes for an easier approach to complex problems. Partly, these hopes were exaggerated, partly the essence of OOP was driven to unwise extremes for the sake of unity of doctrine. For example, even simple integers were

to be regarded as objects in the new sense. What was this "new sense"?

Before we try to answer this question, we should point out that the notion, but not the term OOP was introduced at least 10 years earlier. In 1965 the language Simula was introduced as a variant of Algol adapted to and extended for the needs of simulation of systems governed by discrete events. (What later became objects were then called elements). A well-known example of such a system was a supermarket. Here customers, merchandise, cashiers, personnel, shelves, carts, were all modeled as objects, and they formed classes. For modeling such systems, this was the obvious approach. Only 10 years later, under the leadership of Smalltalk, was the new viewpoint extended to other applications. Of foremost interest were operating systems (OS), where resource pools, processors, windows, texts and devices became regarded as autonomous agents. Graphics editors were another suitable application. They are used to handle various figures such as lines, circles, rectangles, ellipses, text captions, each forming their own class.

In these examples, the objects constitute almost autonomous entities, in the simulated models as well as the operating systems. They not only feature attributes and properties common to all members of their class, but also a common behavior represented by associated procedures. These are typically activated by a "higher power" such as a scheduler in the case of simulation, or a human being in the case of an OS or a graphics editor. Both data and procedures together constitute and characterize objects.

The Smalltalk designers clearly wished to present not only a new language, but also a new paradigm, a new approach

to programming. To be effective and convincing in this endeavor, they provided also a new terminology to underscore a different quality of programming. In this effort, record-structured variables became objects, their associated procedures became methods, a data type became a class, and procedure calls are now termed sending messages to objects. This is denoted as

```
object.method (message)
```

In Simula, the ancestor of OOP, there was not a set of methods to be invoked by messages, but rather a single coroutine representing the algorithmic behavior of the simulated object, which was regarded as continuously alive. We may now consider the parts between breakpoints as methods, although the analogy is somewhat lacking. For example, a customer of the supermarket would be characterized by 1st entering the scene, picking up a cart, wandering along the shelves, picking up merchandise, passing a cashier, and finally returning the cart and leaving the scene. Objects (customers) would emerge (be allocated), proceed, and leave (be deallocated). Hence, typically a dynamic data structure, a linked list, is used to represent the set of currently involved objects.

It now seems that OOP merely provides (or dictates?) a different view of familiar programming concepts, but does not require any additional features in a language, except perhaps some notational conventions. The same old contents in a new form? Subsequently, we will investigate this question.

Let us consider a simple, rudimentary line drawing editor. It is typical for many applications in so far as it operates on a set of objects. In this case, the objects are straight lines, rectangles, circles, ellipses, text captions, and others. These objects are typically represented as a linked list, to which it is easy to add and delete new instances.

However, using a statically typed language, we immediately encounter a difficulty. Declaring each of the figures by its own record type, we are prevented from forming a single list. Declaring a separate list for each type of figure appears as artificial and cumbersome. On the other hand, integrating all figure types into a single declaration appears as equally contorted and cumbersome:

1. a discrimination field is necessary, subsequently giving rise to various IF statements.

2. a single declaration will contain record fields that pertain to only a few, but not to all figure types.

Whatever one chooses to do, the solution remains unsatisfactory.

We therefore introduce a new feature in the language Oberon. It is called type extension. Basically, in our example, we declare a base type called `Figure`, and then a derived type for each individual kind of figure, here called `Line`, `Rectangle`, `Circle`, etc. The key is that each instance of these subtypes is also considered as of its base type `Figure`. Such declarations of derived types have the form:

```
RecordType = RECORD (Base) FieldListSequence END
```

The Base specifies the base type of the new, derived record type, which thereby becomes a subtype. Evidently, it now becomes possible to define entire types hierarchies. For our example:

```
TYPE Figure = POINTER TO FigureDesc;
FigureDesc  = RECORD (*this is the base type*)
   x,y,w,h: INT; (*coordinates, width & height*)
      next: Figure (*next in list of figures*)
END;
LineDesc      = RECORD (FigureDesc) END;
RectangleDesc = RECORD (FigureDesc) lw : INT END;
CircleDesc    = RECORD (FigureDesc) r  : INT END;
EllipseDesc   = RECORD (FigureDesc) a,b: INT END;
CaptionDesc   = RECORD (FigureDesc) cap:
                                  ARRAY 32 OF INT END
```

Accordingly, a descriptor of a rectangle consists of the fields x, y, w, h, and lw (line width), the one for a circle of the fields x, y, w, h, r(adius), the one for an ellipse of x, y, w, h, a, b, and the one for a text caption of x, y, w, h, cap. The derived types retain all fields of the base type. In OO terminology, they are said to inherit the properties of the base. We will not adopt this anthropomorphic term.

We call the derived types extended types, because they extend the base type with additional properties. In OO terminology, the base type is called an abstract type, on which the concrete types, the extended types, rest. All of them are compatible also with their base type. Therefore, it is possible to link them into a single list through the base

type's field next, and therefore to build data structures that contain elements of different types, that is, inhomogeneous structures. For many applications, this is an important and necessary requirement.

When, for example, traversing such an inhomogeneous list, it may be necessary to determine the type of each encountered object. It is only known that every element in the list is of type Figure, but now we wish to determine the subtype of an individual element. This is possible through the new facility of the type test, which is classified as a Boolean expression, and has the form (syntax)

```
expression = variable IS identifier
```

Assume a variable *p* of type Figure, then the traversal of a list and the discriminated application of a drawing procedure can be expressed as follows:

```
WHILE p # NIL DO
      IF p IS Line   THEN DrawLine(p)
   ELSIF p IS Circle THEN DrawCircle(p)
   END ;
   p := p.next
END
```

We will discover in the next section that there is actually a better, shorter way of expressing this action.

The type test is a necessary feature, because Oberon has deviated from the dogma of strictly static data typing. Note, however, that the actual type T of a variable declared to be of base type T0 can only be a subtype of T0. A Figure

can be a `Line`, `Circle`, etc., but not, for example, an integer. Type tests are therefore not needed for most data accesses. Furthermore, type tests can be implemented very efficiently.

In addition to the type test, Oberon also offers the construct of type guard, which has the form of a selector in variable designators.

```
designator = qualident {. ident | "[" ExpList "]"
                              | (qualident) | ^ }
```

Consider the example of the designator

```
p(Circle).r
```

The simple designator `p.r` would not be acceptable, because *p* is of type `Figure`, which does not feature a field *r*. With the type guard, the programmer can ascertain that in this case *p* is also of type `Circle`, in which case the field *r* is indeed applicable. Whereas *p* is of base type `Figure`, `p(Circle)` is of type `Circle`. Of course, it must be assumed that the programmer has made sure that his claim is correct, and the system will have to check whether this is indeed so (at run time). The type guard in a sense resembles an array index, where a system must check whether the actual index lies within the specified array bounds.

23.3 METHODS

It has already been remarked that objects are characterized not only by their attributes (data), but also by their behavior, i.e. procedures. Just as attributes may differ among various subtypes, so may their behavior. Therefore, it is sensible

to associate the procedures directly with individual objects or at least their subtypes. In strongly OO languages this is achieved by letting class declarations specify associated procedure declarations. In Oberon, we use what is available and refrain from introducing new features. We simply add to a record's data fields other fields that represent procedures, that is, are of procedure types. We redefine our example of the type Figure accordingly. Let us assume the 2 operations of drawing and marking figures.

```
FigureDesc = RECORD
                 x, y, w, h: INT;
                 next: Figure;
                 draw: PROC(f: Figure);
                 mark: PROC(f: Figure; on: BOOL)
             END;
```

Whereas in strictly OO languages methods are automatically associated with every object upon its creation, in Oberon this must be achieved by explicit assignments. Such initial assignments are called installations (of procedures). For example:

```
VAR c: Circle; NEW(c); c.x := ...;
c.draw := DrawCircle;
c.mark := MarkCircle; ...
```

The call of a method, i.e. of an object-bound procedure, is expressed as in the following example:

```
c.mark(c, TRUE)
```

The association of type-specific procedures to every object has the substantial advantage that subclass-discriminating conditional statements can be avoided. For example, the above call automatically activates the appropriate `MarkCircle` procedure without having to execute a sequence of `if` statements distinguishing between lines, circles, rectangles etc. This contributes to efficiency. The drawing of all objects in a list is now expressed quite simply as

```
WHILE p # NIL DO p.draw(p); p := p.next END
```

and in each call automatically the appropriate drawing procedure is activated.

23.4 HANDLERS AND MESSAGES

In Part III of this text, the importance of proper decomposition (modularization) of systems was emphasized. A module restricts the visibility of declarations and incorporates the concept of information hiding. In OO languages a type definition, i.e. a class declaration, assumes the role of a module, also in the sense of unit of separate compilation. In Oberon, we prefer to keep the constructs expressing information hiding, and those for type and procedure definition separated and independent. Hence, every module may contain one or several type definitions, and perhaps none at all.

Nevertheless, considering a data type and its operators as a logical unit, and to hide details of implementation, are old and proven techniques manifest in the concept of the abstract data type. Particularly for complex objects it is highly desirable that the various subtypes can be defined

and implemented by separate teams of programmers, and this perhaps years after the definition of the basis was completed and had been published or distributed. It is desirable to be able to define every derived type in its own module. This demand also holds for base types, with operators applying to the set of objects (in contrast to individual objects) defined in the base module, as well as for those defining the derived type in client modules. Among such operations we mention the so-called broadcast, the traversal of the objects' data structure (list, tree) with application of a method to every element. Declarations of such broadcasts are thus inherently confined to the base module.

Now it may happen that a subtype introduces its specific operations, including some that are not shared by other subtypes. Obviously, it is desirable that also these operations can be broadcast with the respective procedure defined in the base module, even if the new operators were introduced years after the base had been established. The base, however, cannot be changed nor accessed, because in the meantime it might have been distributed to many customers. How can this dilemma be resolved?

The solution lies in replacing the set of methods by a single procedure, which discriminates between the various methods. Such a single procedure we call a handler. It has only 2 parameters: The object to which it is applied, and an identification of the operation with its actual parameters. The 2nd parameter has the form of a record, and we call it a message. For such a message type to be capable of denoting any operation, even those to be introduced in the future, the same feature is used as for adding new types derived from

old ones: *Type Extension*. Let us explain this solution by our example of the graphics editor.

First, the declaration of the base type now becomes

```
FigureDesc = RECORD
                  x, y, w, h: INT;
                        next: Figure;
                      handle: Handler
              END;
```

with the procedure type `Handler` being defined as

```
Handler = PROC(f: Figure; VAR msg: Message)
```

and the new type `Message` by the (extensible) null record

```
Message = RECORD END
```

If certain operations applying to all objects are already known at the time of defining the base type (which is usually the case), the respective message (sub)types are directly declared, as for example:

```
DrawMsg = RECORD (Message) END;
MarkMsg = RECORD (Message) on: BOOL END;
MoveMsg = RECORD (Message) dx, dy: INT END
```

As an aside, we note that the record field next need not be exported from the base module, thus keeping the structure of the set of objects (list, tree) and its operations, such as traversal, hidden and flexible. A broadcast can now be specified as follows:

```
    PROC Broadcast(VAR msg: Message);
      VAR f: Figure;
    BEGIN f := root; (*global variable in base module*)
      WHILE f # NIL DO
        f.handle(f, msg); f := f.next
      END
    END Broadcast
```

Whenever a concrete figure type is introduced, this will typically be done in a new client module importing the base module Figure. In addition to the new subtype, for example Rectangle, its associated handler is declared following the pattern shown below. This handler is installed in the field handle of every object of type Rectangle.

```
    PROC Handle(f: Figure; VAR msg: Message);
      VAR r: Rectangle;
    BEGIN r := f(Rectangle);
         IF msg IS DrawMsg THEN (*draw rectangle r*)
       ELSIF msg IS MarkMsg THEN
         MarkRectangle(r, msg(MarkMsg).on)
       ELSIF msg IS MoveMsg THEN
         INC(r.x, msg(MoveMsg).x);
         INC(r.y, msg(MoveMsg).y)
       ELSIF ...
       END
    END Handle
```

The statement for moving a single object f by displacements dX and dY is now somewhat complicated:

```
    VAR msg: MoveMsg; msg.dx := dX;
    msg.dy := dY;  f.handle(f, msg)
```

However, we must keep in mind that mostly messages are sent to objects indirectly, that is, when the recipient is not known by the sender. A good example is the broadcast:

```
msg.dx := dX; msg.dy := dY; Broadcast(msg)
```

The particular flexibility of this technique using handlers and messages - sometimes identified as polymorphism - as well as its extensibility stems from the fact that "unknown" messages are simply ignored. They "fall through" the cascade of if - elsif statements in the handlers of objects to which the message does not apply.

Let us, for example, add a new module, say `Blinker`, to our editor system. It contains the declaration of the derived type `Blinker` (`BlinkerDesc`). Let it require a new method, causing the object to blink in a certain rhythm. We represent it by the (derived) message `BlinkMsg`. Then the call of the handler of any other object in the global object list, through which a broadcast will propagate, will have no effect, because the case

```
ELSIF msg IS BlinkMsg THEN ...
```

does not exist in the handler. The technique using handlers guarantees optimal, unlimited extensibility of existing systems by merely adding new modules on top of an existing hierarchy.

We summarise the technique as follows:

1. An object is represented by a pointer to a record. The record features a single, procedural field called `handle`.

The procedure assigned to `handle` is called a *handler*. It features 2 parameters:

- the 1st designates the object to which the handler is applied;

- the 2nd identifies the operation to be selected. It is a `VAR` parameter with record structure, and is called a *message*.

2. The handler defines the behavior of an object. It is called with a message which is typically of an extension of type `Message` specific to the object type.

3. The message parameter specifies the action to be taken by its (sub)type. Its record fields constitute the parameters of the action. Their number and types are specific to the particular message type and action.

4. The handler consists of a single if - elsif cascade. Type tests discriminate between the various message subtypes and thereby actions.

5. Assigning the handler to an object is called *installation*.

6. An action is initiated by first setting up the message and then sending one to the object.

7. Messages may be broadcast to all objects of a heterogeneous data structure without knowing its nature. Inapplicable messages will simply be ignored.

This concludes our brief introduction to the OOP paradigm. We realize that almost no language features had to be added to Oberon to support it. Apart from the already present facilities of records and of procedural types, only the notion

of type extension is both necessary and crucial. It allows to construct hierarchies of types and to build inhomogeneous data structures. As a consequence of abandoning the rule of strictly static typing, the introduction of dynamic type tests became necessary. The further facility of the type guard is merely one of convenience.

The procedural style of programming, which is most appropriate in many applications, and the OO style can co-exist in Oberon quite happily. If one chooses to program in the conventional, procedural style, one can ignore the OO facilities, and they do not get in one's way.

LIB

A

SYNTAX OF OBERON

```
digit  = 0-9
hexdig = digit|A-F
letter = A-Z|a-z

id     = letter{letter|digit}
qid    = [id.]id
xid    = id[*]

dec    = digit{digit}
hex    = digit{hexdig}
int    = dec | hex H
sign   = +|-
real   = [sign]dec.{digit}[E[sign]dec]
number = int|real
string = " {char} " | hex X

const  = xid=expr

typdef = xid=type
type   = pre | arr | rec | ptr | pro | fun
pre    = BOOL|BYTE|CHAR|INT|REAL|SET
arr    = ARRAY len{, len} OF type
len    = expr
rec    = RECORD[ (base)][ fields] END
base   = qid
```

```
fields = field{; field}
field  = xids: type
xids   = xid{, xid}
ptr    = POINTER TO type
pro    = PROC[ params]
fun    = PROC[ params]: qid


expr   = exp [rel exp]
rel    = =|#|<|<=|>|>=|IN|IS
exp    = [sign]term {add term}
add    = sign|OR
term   = factor {mul factor}
mul    = *|/|DIV|MOD|&
factor = number | string | NIL | TRUE | FALSE
         | set | pc | (expr) | ~factor
des    = qid{sel}
sel    = .id|'['exprs']'|^|(qid)
set    = '{' [seg{, seg}] '}'
seg    = expr[..expr]
exprs  = expr{, expr}
args   = ([exprs])


s      = [assign|pc|if|cases|while|repeat|for]
assign = des:=expr
pc     = des[args]
ss     = s{; s}
if     =    IF expr THEN ss
         {ELSIF expr THEN ss}
                    [ELSE ss] END
cases  = CASE expr OF case {'|' case} END
case   = [ranges: ss]
ranges = range{, range}
range  = label[..label]
```

```
label  = int|string|qid
while  = WHILE expr DO ss {ELSIF expr DO ss} END
repeat = REPEAT ss UNTIL expr
for    = FOR id:=expr TO expr[ BY expr] DO ss END


proc   = prop | func
prop   = PROC xid [params]; decls
           [BEGIN ss] END id
func   = PROC xid [params]: qid; decls
           [BEGIN ss] [RETURN expr] END id
decls  = [CONST {const;}][TYPE {typdef;}][VAR {field;}]
           {proc;}
params = ([param{; param}])
param  = [VAR] id{, id}: {ARRAY OF} qid


module = MODULE id;[ IMPORT import{, import};] decls
           [BEGIN ss] END id.
import = id[:= id]
```

LIBRARY

B.1 SYMBOLS AND KEYWORDS

```
(  )    + - * /    <  =  >     .  ,  ;  :  :=  ..
[  ]    & | ~ ^    <= # >=   ARRAY    IMPORT
{  }    DIV     CASE      BEGIN    MODULE
        END     ELSE      CONST    RECORD
BY IS   FOR     PROC      ELSIF    REPEAT
DO OF   MOD     THEN      FALSE    RETURN
IF OR   NIL     TRUE      UNTIL
IN TO   VAR     TYPE      WHILE    POINTER
```

B.2 STANDARD FUNCTIONS

Name	ArgType	ResType	Function
ABS(x)	numeric	type(x)	absolute value
ODD(x)	INT	BOOL	x MOD 2 = 1
LEN(v)	array	INT	length
LSL(x,n)	INT	INT	logical shift left
ASR(x,n)	INT	INT	signed shift right
ROR(x,n)	INT	INT	rotated right
ORD(x)	CHAR | BOOL | SET	INT	ordinal # of
CHR(x)	INT	CHAR	character of
FLOOR(x)	REAL	INT	round down
FLT(x)	INT	REAL	identity

Name	ArgType	Procedure
INC(v)	INT	v := v + 1
INC(v, n)	INT	v := v + n
DEC(v)	INT	v := v - 1
DEC(v, n)	INT	v := v - n
INCL(v, x)	SET, INT	v := v + x
EXCL(v, x)	SET, INT	v := v - x
ASSERT(b)	BOOL	abort if ~b
NEW(v)	pointer type	allocate v^