RC . 89 . 111

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Niklaus Wirth

From Modula to Oberon

The Programming Language Oberon

September 1989

89. 10. 15

111

[illegible]

Authors' address:

Institut für Computersysteme
ETH-Zentrum
CH-8092 Zürich, Switzerland

From Modula to Oberon

N. Wirth

Abstract

The programming language Oberon is the result of a concentrated effort to increase the power of Modula-2 and simultaneously to reduce its complexity. Several features were eliminated, and a few were added in order to increase the expressive power and flexibility of the language. This paper describes and motivates the changes. The language is defined in a concise report.

Introduction

The programming language Oberon evolved from a project whose goal was the design of a modern, flexible, and efficient operating system for a single-user workstation. A principal guideline was to concentrate on properties that are genuinely essential and – as a consequence – to omit ephemeral issues. It is the best way to keep a system in hand, to make it understandable, explicable, reliable, and efficiently implementable.

Initially, it was planned to express the system in Modula-2 [1] (subsequently called Modula), as that language supports the notion of modular design quite effectively, and because an operating system has to be designed in terms of separately compilable parts with conscientiously chosen interfaces. In fact, an operating system should be no more than a set of basic modules, and the design of an application must be considered as a goal-oriented extension of that basic set: Programming is always extending a given system.

Whereas modern languages, such as Modula, support the notion of extensibility in the procedural realm, the notion is less well established in the domain of data types. In particular, Modula does not allow the definition of new data types as extensions of other, programmer-defined types in an adequate manner. An additional feature was called for, thereby giving rise to an *extension* of Modula.

The concept of the planned operating system also called for a highly dynamic, centralized storage management relying on the technique of garbage collection. Although Modula does not prevent the incorporation of a garbage collector in principle, its variant record feature constitutes a genuine obstacle. As the new facility for extending types would make the variant record feature superfluous, the removal of this stumbling block was a logical decision. This step, however, gave rise to a *restriction* (subset) of Modula.

It soon became clear that the rule to concentrate on the essential and to eliminate the inessential should not only be applied to the design of the new system, but equally stringently to the language in which the system is formulated. The application of the principle thus led from Modula to a new language. However, the adjective "new" has to be understood in proper context: Oberon evolved from Modula by very few additions and several subtractions. In relying on evolution rather than revolution we remain in the tradition of a long development that led from Algol to Pascal, then to Modula-2, and eventually to Oberon. The common traits of these languages are their procedural rather than functional model and the strict typing of data. Even more fundamental, perhaps, is the idea of abstraction: the language must be defined in terms of mathematical, abstract concepts without reference to any computing mechanism. Only if a language satisfies this criterion, can it be called "higher-level". No syntactic coating whatsoever can earn a language this attribute alone.

The definition of a language must be coherent and concise. This can only be achieved by a careful choice of the underlying abstractions and an appropriate structure combining them. The language manual must be reasonably short, avoiding the explanation of individual cases derivable from the general rules. The power of a formalism must not be measured by the length of its description. To the contrary, an overly lengthy definition is a sure symptom of inadequacy. In this respect, not complexity but simplicity must be the goal.

In spite of its brevity, a description must be complete. Completeness is to be achieved within the framework of the chosen abstractions. Limitations imposed by particular implementations do not belong to a language definition proper. Examples of such restrictions are the maximum values of numbers, rounding and truncation errors in arithmetic, and actions taken when a program violates the stated rules. It should not be necessary to supplement a language definition with a voluminous standards document to cover "unforeseen" cases.

But neither should a programming language be a mathematical theory only. It must be a practical tool. This imposes certain limits on the terseness of the formalism. Several features of Oberon are superfluous from a purely theoretical point of view. They are nevertheless retained for practical reasons, either for programmers' convenience or to allow for efficient code generation without the necessity of complex, "optimizing" pattern matching algorithms in compilers. Examples of such features are the presence of several forms of repetitive statements, and of standard procedures such as INC, DEC, and ODD. They complicate neither the language conceptually nor the compiler to any significant degree.

These underlying premises must be kept in mind when comparing Oberon with other languages. Neither the language nor its defining document reach the ideal; but Oberon approximates these goals much better than its predecessors.

A compiler for Oberon has been implemented for the NS32000 processor family and is embedded in the Oberon operating environment [8]. The compiler requires less than 50 KByte of memory, consists of 6 modules with a total of about 4000 lines of source code, and compiles itself in about 15 seconds on a workstation with a 25MHz NS32532 processor.

After extensive experience in programming with Oberon, a revision was defined and implemented. The differences between the two versions are summarised towards the end of the paper. Subsequently, we present a brief introduction to (revised) Oberon assuming familiarity with Modula (or Pascal), concentrating on the added features and listing the eliminated ones. In order to be able to start with a clean slate, the latter are taken first.

Features omitted from Modula

Data types

Variant records are eliminated, because they constitute a genuine difficulty for the implementation of a reliable storage management system based on automatic garbage collection. The functionality of variant records is preserved by the introduction of extensible data types.

Opaque types cater for the concept of abstract data type and information hiding. They are eliminated as such, because again the concept is covered by the new facility of extended record types.

Enumeration types appear to be a simple enough feature to be uncontroversial. However, they defy extensibility over module boundaries. Either a facility to extend given enumeration types has to be introduced, or they have to be dropped. A reason in favour of the latter, radical solution was the observation that in a growing number of programs the indiscriminate use of enumerations (and subranges) had led to a type explosion that contributed not to program clarity but rather to verbosity. In connection with import and export, enumerations give rise to the exceptional rule that the import of a type identifier also causes the (automatic) import of all associated constant identifiers. This exceptional rule defies conceptual simplicity and causes unpleasant problems for the implementor.

Subrange types were introduced in Pascal (and adopted in Modula) for two reasons: (1) to indicate that a variable accepts a limited range of values of the base type and to allow a compiler to generate appropriate guards for assignments, and (2) to allow a compiler to allocate the minimal storage space needed to store values of the indicated subrange. This appeared desirable in connection with packed records. Very few implementations have taken advantage of this space saving facility, because the additional compiler complexity is very considerable. Reason 1 alone, however, did not appear to provide sufficient justification to retain the subrange facility in Oberon.

With the absence of enumeration and subrange types, the general possibility of defining *set types* based on given element types appeared as redundant. Instead, a single, basic type SET is introduced, whose values are sets of integers from 0 to an implementation-defined maximum.

The basic type *CARDINAL* had been introduced in Modula in order to allow address arithmetic with values from 0 to 2^{16} on 16-bit computers. With the prevalence of 32-bit addresses in modern processors, the need for unsigned arithmetic has practically vanished, and therefore the type CARDINAL has been eliminated. With it, the bothersome incompatibilities of operands of types CARDINAL and INTEGER have disappeared.

Pointer types are restricted to be bound to a record type or to an array type.

The notion of a definable index type of arrays has also been abandoned: All indices are by default integers. Furthermore, the lower bound is fixed to 0; array declarations specify a number of elements (length) rather than a pair of bounds. This break with a long standing tradition since Algol 60 clearly demonstrates the principle of eliminating the inessential. The specification of an arbitrary lower bound hardly provides any additional expressive power. It represents a rather limited kind of mapping of indices which introduces a hidden computational effort that is incommensurate with the supposed gain in convenience. This effort is particularly heavy in connection with bound checking and with dynamic arrays.

Modules and import/export rules

Experience with Modula over the last eight years has shown that *local modules* were rarely used. Considering the additional complexity of the compiler required to handle them, and the additional complications in the visiblity rules of the language definition, the elimination of local modules appears justified.

The *qualification* of an imported object's identifier x by the exporting module's name M, viz. M.x, can be circumvented in Modula by the use of the import clause FROM M IMPORT x. This facility has also been discarded. Experience in programming systems involving many modules has taught that the explicit qualification of each occurrence of x is actually preferable. A simplification of the compiler is a welcome side-effect.

The dual role of the main module in Modula is conceptually confusing. It constitutes a *module* in the sense of a package of data and procedures enclosed by a scope of visibility, and at the same time it constitutes a single *procedure* called main program. A module is composed of two textual pieces, called the definition part and the implementation part. The former is missing in the case of a main program module.

By contrast, a module in Oberon is in itself complete and constitutes a unit of compilation. Definition and implementation parts are merged; names to be visible in client modules, i.e. exported identifiers, are marked, and they typically precede the declarations of objects not exported. A compilation generates in general a changed object file and a new symbol file. The latter contains information about exported objects for use in the compilation of client modules. The generation of a new symbol file must, however, be specifically enabled by a compiler option, because it will invalidate previous compilations of clients.

The notion of a main program has been abandoned. Instead, the set of modules linked through imports typically contains (parameterless) procedures. They are to be considered as individually activatable, and they are called *commands*. Such an activation has the form M.P, where P denotes the command and M the module containing it. The effect of a command is considered – not like that of a main program as accepting input and transforming it to output – as a change of state represented by global data.

Statements

The *with statement* of Modula has been discarded. Like in the case of imported identifiers, the explicit qualification of field identifiers is to be preferred. Another form of with statement is introduced; it has a different function and is called a regional guard (see below).

The elimination of the for statement constitutes a break with another long standing tradition. The baroque mechanism of Algol 60's for statement had been trimmed significantly in Pascal (and Modula). Its marginal value in practice has led to its absence from Oberon.

Low-level facilities

Modula makes access to machine-specific facilities possible through low-level constructs, such as the data types ADDRESS and WORD, absolute addressing of variables, and type casting functions. Most of them are packaged in a module called SYSTEM. These features were supposed to be rarely used and easily visible through the presence of the identifier SYSTEM in a module's import list. Experience has revealed, however, that a significant number of programmers import this module quite indiscriminately. A particularly seductive trap is the use of Modula's type transfer functions.

It appears preferable to drop the pretense of portability of programs that import a "standard", yet system-specific module. *Type transfer functions* denoted by type identifiers are therefore eliminated, and the module SYSTEM is restricted to providing a few machine-specific functions that typically are compiled into inline code. In particular, it does not contain any data types, such as *ADDRESS* and *WORD*. Individual implementations are free to provide appropriate versions of the module SYSTEM, but their facilities do not belong to the language definition. The use of SYSTEM declares a program to be patently implementation-specific and thereby non-portable.

Concurrency

The system Oberon does not require any language facilities for expressing concurrent processes. The pertinent rudimentary features of Modula, in particular the coroutine, were therefore not retained. This exclusion is merely a reflection of our actual needs within the concrete project, but not on the general relevance of concurrency in programming.

Features introduced in Oberon

In contrast to the number of eliminated features, there are only a few new ones. The important new concepts are type extension and type inclusion. Furthermore, open arrays may have several dimensions (indices), whereas in Modula they were confined to a single dimension.

Type extension

The most important addition is the facility of extended record types. It permits the construction of new types on the basis of existing types, and establishes a certain degree of compatibility between the new and old types. Assuming a given type

```
T    = RECORD x, y: INTEGER END
```

extensions may be defined which contain certain fields in addition to the existing ones. For example

```
T0   = RECORD (T)  z: REAL END
T1   = RECORD (T)  w: LONGREAL END
```

define types with fields x, y, z and x, y, w respectively. We define a type declared by

```
T'   = RECORD (T) <field definitions> END
```

to be a *(direct) extension* of T, and conversely T to be the *(direct) base type* of T'. Extended types may be extended again, giving rise to the following definitions:

A type T' is an *extension* of T, if T' = T or T' is a direct extension of an extension of T. Conversely, T is a *base type* of T', if T = T' or T is the direct base type of a base type of T'. We denote this relationship by T' → T.

The rule of assignment compatibility states that values of an extended type are assignable to variables of their base types. For example, a record of type T0 can be assigned to a variable of the base type T. This assignment involves the fields x and y only, and in fact constitutes a *projection* of the value onto the space spanned by the base type.

It is important to allow modules which import a base type to be able to declare extended types. In fact, this is probably the normal usage.

This concept of extensible data type gains importance when extended to pointers. It is appropriate to say that a pointer type P' bound to T' extends a pointer type P, if P is bound to a base type T of T', and to extend the assignment rule to cover this case. It is now possible to form data structures whose nodes are of different types, i.e. inhomogeneous data structures. The inhomogeneity is automatically (and most sensibly) bounded by the fact that the nodes are linked by pointers of a common base type.

Typically, the pointer fields establishing the structure are contained in the base type T, and the procedures manipulating the structure are defined in the same (base) module as T. Individual extensions (variants) are defined in client modules together with procedures operating on nodes of the extended type. This scheme is in full accordance with the notion of system extensibility: new modules defining new extensions may be added to a system without requiring a change of the base modules, not even their recompilation.

As access to an individual node via a pointer bound to a base type provides a projected view of the node data only, a facility to widen the view is necessary. It depends on the ability to determine the actual type of the referenced node. This is achieved by a *type test*, a Boolean expression of the form

```
    t IS T'        (or  p IS P')
```

If the test is affirmative, an assignment t' := t (t' of type T') or p' := p (p' of type P') should be possible. The static view of types, however, prohibits this. Note that both assignments violate the rule of assignment compatibility. The desired assignment is made possible by providing a *type guard* of the form

```
    t' := t(T')     (p' := p(P'))
```

and by the same token access to the field z of a T0 (see previous examples) is made possible by a type guard in the designator t(T0).z. Here the guard asserts that t is (currently) of type T0. In analogy to array bound checks and case selectors, a failing guard leads to program abortion.

Whereas a guard of the form t(T) asserts that t is of type T for the designator (starting with) t only, a *regional type guard* maintains the assertion over an entire sequence of statements. It has the form

```
    WITH t: T DO StatementSequence END
```

and specifies that t is to be regarded as of type T within the entire statement sequence. Typically, T is an extension of the declared type of t. Note that assignments to t within the region therefore require the assigned value to be (an extension) of type T. The regional guard serves to reduce the number of guard evaluations.

As an example of the use of type tests and guards, consider the following types Node and Object defined in a module M:

```
    TYPE  Node =   POINTER TO Object;
          Object = RECORD key, x, y: INTEGER;
                     left, right: Node
                   END
```

Elements in a tree structure anchored in a variable called root (of type Node) are searched by the procedure *element* defined in M.

```
    PROCEDURE element(k: INTEGER): Node;
       VAR p: Node;
```

```
BEGIN p := root;
   WHILE (p # NIL) & (p.key # k) DO
      IF p.key < k THEN p := p.left ELSE p := p.right END
   END ;
   RETURN p
END element
```

Let extensions of the type Object be defined (together with their pointer types) in a module M1 which is a client of M:

```
TYPE  Rectangle =     POINTER TO RectObject;
      RectObject =    RECORD (Object) w, h: REAL END ;
      Circle =        POINTER TO CircleObject;
      CircleObject =  RECORD (Object) rad: REAL; shaded: BOOLEAN END
```

After the search of an element, the type test is used to discriminate between the different extensions, and the type guard to access extension fields. For example:

```
p := M.element(K);
IF p # NIL THEN
     IF p IS Rectangle THEN ... p(Rectangle).w ...
     ELSIF (p IS Circle) & ~p(Circle).shaded THEN ... p(Circle).rad ...
     ELSIF ...
```

The extensibility of a system rests upon the premise that new modules defining new extensions may be added without requiring adaptations nor even recompilation of the existing parts, although components of the new types are included in already existing data structures.

The type extension facility not only replaces Modula's variant records, but represents a type-safe alternative. Equally important is its effect of relating types in a type hierarchy. We compare, for example, the Modula types

```
T0'  =  RECORD t: T;  z: REAL END ;
T1'  =  RECORD t: T;  w: LONGREAL END
```

which refer to the definition of T given above, with the extended Oberon types T0 and T1 defined above. First, the Oberon types refrain from introducing a new naming scope. Given a variable r0 of type T0, we write r0.x instead of r0.t.x as in Modula. Second, the types T, T0', and T1' are distinct and unrelated. In contrast, T0 and T1 are related to T as extensions. This becomes manifest through the type test, which asserts that variable r0 is not only of type T0, but also of base type T.

The declaration of extended record types, the type test, and the type guard are the only additional features introduced in this context. A more extensive discussion is provided in [2]. The concept is very similar to the class notion of Simula 67 [3], Smalltalk [4], Object Pascal [5], C++ [6], and others, where the properties of the base class are said to be *inherited* by the derived classes. The class facility stipulates that all procedures applicable to objects of the class be defined together with the data definition. This dogma stems from the notion of abstract data type, but it is a serious obstacle in the development of large systems, where the possibility to add further procedures defined in additional modules is highly desirable. It is awkward to be obliged to redefine a class solely because a method (procedure) has been added or changed, particularly when this change requires a recompilation of the class definition and of all its client modules.

We emphasise that the type extension facility – although gaining its major role in connection with pointers to build heterogeneous, dynamic data structures as shown in the example above – also applies to statically declared objects used as variable parameters. Such objects are allocated in a workspace organized as a stack of procedure activation records, and therefore take advantage of an extremely efficient allocation and deallocation scheme.

In Oberon, procedure *types* rather than procedures (methods) are connected with objects in the program text. The binding of actual methods (specific procedures) to objects (instances) is delayed until the program is executed. The association of a procedure type with a data type occurs through the declaration of a record field. This field is given a procedure type. The association of a method – to use Smalltalk terminology – with an object occurs through the assignment of a specific procedure as value to the field, and not through a static declaration in the extended type's definition which then "overrides" the declaration given in the base type. Such a procedure is called a *handler*. Using type tests, the handler is capable of discriminating among different extensions of the record's (object's) base type. In Smalltalk, the compatibility rules between a class and its subclasses are confined to pointers, thereby intertwining the concepts of access method and data type in an undesirable way. In Oberon, the relationship between a type and its extensions is based on the established mathematical concept of projection.

In Modula, it is possible to declare a pointer type within an implementation module, and to export it as an opaque type by listing the same identifier in the corresponding definition module. The net effect is that the type is exported while all its properties remain hidden (invisible to clients). In Oberon, this facility is generalized in the sense that the selection of the record fields to be exported is arbitrary and includes the cases all and none. The collection of exported fields defines a partial view – a *public projection* – to clients.

In client modules as well as in the module itself, it is possible to define extensions of the base type (e.g. TextViewers or GraphViewers). Of importance is also the fact that non-exported components (fields) may have types that are not exported either. Hence, it is possible to hide certain data types effectively, although components of (opaquely) exported types refer to them.

Type inclusion

Modern processors feature arithmetic operations on several number formats. It is desirable to have all these formats reflected in the language as basic types. Oberon features five of them:

LONGINT, INTEGER, SHORTINT	(integer types)
LONGREAL, REAL	(real types)

With the proliferation of basic types, a relaxation of compatibility rules among them becomes almost mandatory. (Note that in Modula the numeric types INTEGER, CARDINAL, and REAL are incompatible). To this end, the notion of *type inclusion* is introduced: a type T includes a type T', if the values of type T' are also values of type T. Oberon postulates the following hierarchy:

LONGREAL ⊇ REAL ⊇ LONGINT ⊇ INTEGER ⊇ SHORTINT

The assignment rule is relaxed accordingly: A value of type T' can be assigned to a variable of type T, if T' is included in T (or if T' extends T), i.e. if T ⊇ T' or T' → T. In this respect, we return to (and extend) the flexibility of Algol 60. For example, given variables

```
i: INTEGER;  k: LONGINT;  x: REAL
```

the assignments

```
k := i;  x := k;  x := 1;  k := k+i;  x := x*10 + i
```

conform to the rules, whereas the statements i := k; k := x are not acceptable. x := k may involve truncation.

The presence of several numeric types is evidently a concession to implementations which can allocate different amounts of storage to variables of the different types, and which thereby offer an opportunity for storage economization. This practical aspect should – with due respect for mathematical abstraction – not be ignored. The notion of type inclusion minimises the consequences for the programmer and requires only few implicit instructions for changing the data representation, such as sign extensions and integer to floating-point conversions.

Differences between Oberon and Revised Oberon

A revision of Oberon was defined after extensive experience in the use and implementation of the language. Again, it is characterized by the desire to simplify and integrate. The differences between the original version [7] and the revised version [9] are the following:

1. Definition and implementation parts of a module are merged. It appeared as desirable to have a module's specification contained in a single document, both from the view of the programmer and the compiler. A specification of its interface to clients (the definition part) can be derived automatically. Objects previously declared in the definition part (and repeated in the implementation part), are specially marked for export. The need for a structural comparison of two texts by the compiler thereby vanishes.
2. The syntax of lists of parameter types in the declaration of a procedure type is the same as that for regular procedure headings. This implies that dummy identifiers are introduced; they may be useful as comments.
3. The rule that type declarations must follow constant declarations, and that variable declarations must follow type declarations is relaxed.
4. The apostrophe is eliminated as a string delimiter.
5. The relaxed parameter compatibility rule for the formal type ARRAY OF BYTE is applicable for variable parameters only.

Summary

The language Oberon has evolved from Modula-2 and incorporates the experiences of many years of programming in Modula. A significant number of features have been eliminated. They appear to have contributed more to language and compiler complexity than to genuine power and flexibility of expression. A small number of features have been added, the most significant one being the concept of type extension.

The evolution of a new language that is smaller, yet more powerful than its ancestor is contrary to common practices and trends, but has inestimable advantages. Apart from simpler compilers, it results in a concise defining document [9], an indispensible prerequisite for any tool that must serve in the construction of sophisticated and reliable systems.

Acknowlegement

It is impossible to explicitly acknowledge all contributions of ideas that ultimately simmered down to what is now Oberon. Most came from the use or study of existing languages, such as Modula-2, Ada, Smalltalk, and Cedar, which often taught us how *not* to do it. Of particular value was the contribution of Oberon's first user, J. Gutknecht. The author is grateful for his insistence on the elimination of dead wood and on basing the remaining features on a sound mathematical foundation. And last, thanks go to the anonymous referee who very carefully read the manuscript and contributed many valuable suggestions for improvement.

References

1. N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1982.
2. N. Wirth. Type Extensions. *ACM Trans. on Prog. Languages and Systems, 10,* 2 (April 1988) 204-214.
3. G. Birtwistle, et al. *Simula Begin.* Auerbach, 1973.
4. A. Goldberg, D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.
5. L. Tesler. Object Pascal Report. *Structured Language World, 9,* 3 (1985), 10-14.
6. B. Stroustrup. *The Programming Language C++.* Addison-Wesley, 1986.
7. N. Wirth. The programming language Oberon. *Software - Practice and Experience, 18,* 7 (July 1988), 671-690.
8. J. Gutknecht and N. Wirth. The Oberon System. *Software - Practice and Experience, 19,* (1989)
9. N. Wirth. The programming language Oberon (Revised Report). (companion paper)

The Programming Language Oberon
(Revised Report)

N.Wirth

Make it as simple as possible, but not simpler.
A. Einstein

1. Introduction

Oberon is a general–purpose programming language that evolved from Modula–2. Its principal new feature is the concept of *type extension*. It permits the construction of new data types on the basis of existing ones and to relate them.

This report is not intended as a programmer's tutorial. It is intentionally kept concise. Its function is to serve as a reference for programmers, implementors, and manual writers. What remains unsaid is mostly left so intentionally, either because it is derivable from stated rules of the language, or because it would require to commit the definition when a general commitment appears as unwise.

2. Syntax

A language is an infinite set of sentences, namely the sentences well formed according to its syntax. In Oberon, these sentences are called compilation units. Each unit is a finite sequence of symbols from a finite vocabulary. The vocabulary of Oberon consists of identifiers, numbers, strings, operators, delimiters, and comments. They are called lexical symbols and are composed of sequences of characters. (Note the distinction between symbols and characters.)

To describe the syntax, an extended Backus–Naur Formalism called EBNF is used. Brackets [and] denote optionality of the enclosed sentential form, and braces { and } denote its repetition (possibly 0 times). Syntactic entities (non–terminal symbols) are denoted by English words expressing their intuitive meaning. Symbols of the language vocabulary (terminal symbols) are denoted by strings enclosed in quote marks or words written in capital letters, so–called reserved words. Syntactic rules (productions) are marked by a $ sign at the left margin of the line.

3. Vocabulary and representation

The representation of symbols in terms of characters is defined using the ASCII set. Symbols are identifiers, numbers, strings, operators, delimiters, and comments. The following lexical rules must be observed. Blanks and line breaks must not occur within symbols (except in comments, and blanks in strings). They are ignored unless they are essential to separate two consecutive symbols. Capital and lower–case letters are considered as being distinct.

1. *Identifiers* are sequences of letters and digits. The first character must be a letter.

```
$     ident = letter {letter | digit}.
```

Examples:

```
x  scan  Oberon  GetSymbol  firstLetter
```

2. *Numbers* are (unsigned) integers or real numbers. Integers are sequences of digits and may be followed by a suffix letter. The type is the minimal type to which the number belongs (see 6.1.). If no suffix is specified, the representation is decimal. The suffix H indicates hexadecimal representation.

A real number always contains a decimal point. Optionally it may also contain a decimal scale factor. The letter E (or D) is pronounced as "times ten to the power of". A real number is of type REAL, unless it has a scale factor containing the letter D; in this case it is of type LONGREAL.

```
$    number = integer | real.
$    integer = digit {digit} | digit {hexDigit} "H" .
$    real = digit {digit} "." {digit} [ScaleFactor].
$    ScaleFactor = ("E" | "D") ["+" | "-"] digit {digit}.
$    hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
$    digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Examples:

```
1987
100H              = 256
12.3
4.567E8           = 456700000
0.57712566D-6     = 0.00000057712566
```

3. *Character constants* are either denoted by a single character enclosed in quote marks or by the ordinal number of the character in hexadecimal notation followed by the letter X.

```
$    CharConstant = """" character """" | digit {hexDigit} "X".
```

4. *Strings* are sequences of characters enclosed in quote marks ("). A string cannot contain a quote mark. The number of characters in a string is called the *length* of the string. Strings can be assigned to and compared with arrays of characters (see 9.1 and 8.2.4).

```
$    string = """" {character} """" .
```

Examples:

```
"OBERON"   "Don't worry!"
```

5. *Operators and delimiters* are the special characters, character pairs, or *reserved words* listed below. These reserved words consist exclusively of capital letters and cannot be used in the role of identifiers.

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

6. *Comments* may be inserted between any two symbols in a program. They are arbitrary character sequences opened by the bracket (* and closed by *). Comments do not affect the meaning of a program.

4. Declarations and scope rules

Every identifier occurring in a program must be introduced by a declaration, unless it is a predefined identifier. Declarations also serve to specify certain permanent properties of an object, such as whether it is a constant, a type, a variable, or a procedure.

The identifier is then used to refer to the associated object. This is possible in those parts of a program only which are within the *scope* of the declaration. No identifier may denote more than one object within a given scope. The scope extends textually from the point of the declaration to the end of the block (procedure or module) to which the declaration belongs and hence to which the object is *local.* The scope rule has the following amendments:

1. If a type T is defined as POINTER TO T1 (see 6.4), the identifier T1 can be declared textually following the declaration of T, but it must lie within the same scope.

2. Field identifiers of a record declaration (see 6.3) are valid in field designators only.

In its declaration, an identifier in the global scope may be followed by an export mark (*) to indicate that it be exported from its declaring module. In this case, the identifier may be used in other modules, if they import the declaring module. The identifier is then prefixed by the identifier designating its module (see Ch. 11). The prefix and the identifier are separated by a period and together are called a *qualified identifier.*

```
$       qualident = [ident "."] ident.
$       identdef = ident ["*"].
```

The following identifiers are predefined; their meaning is defined in the indicated sections:

ABS	(10.2)	LEN	(10.2)
ASH	(10.2)	LONG	(10.2)
BOOLEAN	(6.1)	LONGINT	(6.1)
BYTE	(6.1)	LONGREAL	(6.1)
CAP	(10.2)	MAX	(10.2)
CHAR	(6.1)	MIN	(10.2)
CHR	(10.2)	NEW	(6.4)
DEC	(10.2)	ODD	(10.2)
ENTIER	(10.2)	ORD	(10.2)
EXCL	(10.2)	REAL	(6.1)
FALSE	(6.1)	SET	(6.1)
HALT	(10.2)	SHORT	(10.2)
INC	(10.2)	SHORTINT	(6.1)
INCL	(10.2)	TRUE	(6.1)
INTEGER	(6.1)		

5. Constant declarations

A constant declaration associates an identifier with a constant value.

```
$       ConstantDeclaration = identdef "=" ConstExpression.
$       ConstExpression = expression.
```

A constant expression can be evaluated by a mere textual scan without actually executing the program. Its operands are constants (see Ch. 8). Examples of constant declarations are

```
N     =    100
limit =    2*N -1
all   =    {0 .. WordSize-1}
```

6. Type declarations

A data type determines the set of values which variables of that type may assume, and the operators that are applicable. A type declaration is used to associate an identifier with the type. Such association may be with unstructured (basic) types, or it may be with structured types, in which case it defines the structure of variables of this type and, by implication, the operators that are applicable to the components. There are two different structures, namely arrays and records, with different component selectors.

```
$    TypeDeclaration = identdef "=" type.
$    type = qualident | ArrayType | RecordType | PointerType | ProcedureType.
```

Examples:

```
Table        = ARRAY N OF REAL

Tree         = POINTER TO Node

Node         = RECORD key: INTEGER;
                  left, right: Tree
               END

CenterNode   = RECORD (Node)
                 name: ARRAY 32 OF CHAR;
                 subnode: Tree
               END

Function*    = PROCEDURE (x: INTEGER): INTEGER
```

6.1. Basic types

The following basic types are denoted by predeclared identifiers. The associated operators are defined in 8.2, and the predeclared function procedures in 10.2. The values of a given basic type are the following:

1.	BOOLEAN	the truth values TRUE and FALSE.
2.	CHAR	the characters of the ASCII set (0X ... 0FFX).
3.	SHORTINT	the integers between MIN(SHORTINT) and MAX(SHORTINT).
4.	INTEGER	the integers between MIN(INTEGER) and MAX(INTEGER).
5.	LONGINT	the integers between MIN(LONGINT) and MAX(LONGINT).
6.	REAL	real numbers between MIN(REAL) and MAX(REAL).
7.	LONGREAL	real numbers between MIN(LONGREAL) and MAX(LONGREAL).
8.	SET	the sets of integers between 0 and MAX(SET).
9.	BYTE	(see 9.1 and 10.1)

Types 3 to 5 are *integer* types, 6 and 7 are *real* types, and together they are called *numeric* types. They form a hierarchy; the larger type *includes* (the values of) the smaller type:

```
LONGREAL ⊇ REAL ⊇ LONGINT ⊇ INTEGER ⊇ SHORTINT
```

6.2. Array types

An array is a structure consisting of a fixed number of elements which are all of the same type, called the *element type.* The number of elements of an array is called its *length.* The elements of the array are designated by indices, which are integers between 0 and the length minus 1.

```
$    ArrayType = ARRAY length {"," length} OF type.
$    length = ConstExpression.
```

A declaration of the form

```
ARRAY N0, N1, ... , Nk OF T
```

is understood as an abbreviation of the declaration

```
ARRAY N0 OF
    ARRAY N1 OF
    ...
        ARRAY Nk OF T
```

Examples of array types:

```
ARRAY N OF INTEGER
ARRAY 10, 20 OF REAL
```

6.3. Record types

A record type is a structure consisting of a fixed number of elements of possibly different types. The record type declaration specifies for each element, called *field,* its type and an identifier which denotes the field. The scope of these field identifiers is the record definition itself, but they are also visible within field designators (see 8.1) referring to elements of record variables.

```
$    RecordType = RECORD ["(" BaseType ")"] FieldListSequence END.
$    BaseType = qualident.
$    FieldListSequence = FieldList {";" FieldList}.
$    FieldList = [IdentList ":" type].
$    IdentList = identdef {"," identdef}.
```

If a record type is exported, field identifiers that are to be visible outside the declaring module must be marked. They are called *public fields*; unmarked fields are called *private fields.*

Record types are extensible, i.e. a record type can be defined as an extension of another record type. In the examples above, CenterNode *(directly) extends* Node, which is the *(direct) base type* of CenterNode. More specifically, CenterNode extends Node with the fields *name* and *subnode.*

Definition: A type T0 *extends* a type T, if it equals T, or if it directly extends an extension of T. Conversely, a type T is a *base type* of T0, if it equals T0, or if it is the direct base type of a base type of T0.

Examples of record types:

```
RECORD day, month, year: INTEGER
END

RECORD
  name, firstname: ARRAY 32 OF CHAR;
  age: INTEGER;
  salary: REAL
END
```

6.4. Pointer types

Variables of a pointer type P assume as values pointers to variables of some type T. The pointer type P is said to be *bound* to T, and T is the *pointer base type* of P. T must be a record or array type. Pointer types inherit the extension relation of their base types. If a type T0 is an extension of T and P0 is a pointer type bound to T0, then P0 is also an extension of P.

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

If p is a variable of type P = POINTER TO T, then a call of the predefined procedure NEW(p) has the following effect (see 10.2): A variable of type T is allocated in free storage, and a pointer to it is assigned to p. This pointer p is of type P; the *referenced* variable p↑ is of type T. Failure of allocation results in p obtaining the value NIL. Any pointer variable may be assigned the value NIL, which points to no variable at all.

6.5. Procedure types

Variables of a procedure type T have a procedure as value. If a procedure P is assigned to a procedure variable of type T, the (types of the) formal parameters of P must be the same as those indicated in the formal type list of T. The same holds for the result type in the case of a function procedure (see 10.1). P must not be declared local to another procedure, and neither can it be a predefined procedure.

```
$    ProcedureType = PROCEDURE [FormalParameters].
```

7. Variable declarations

Variable declarations serve to introduce variables and associate them with identifiers that must be unique within the given scope. They also serve to associate fixed data types with the variables.

```
$    VariableDeclaration = IdentList ":" type.
```

Variables whose identifiers appear in the same list are all of the same type. Examples of variable declarations (refer to examples in Ch. 6):

```
i, j, k:   INTEGER
x, y:      REAL
p, q:      BOOLEAN
s:         SET
f:         Function
a:         ARRAY 100 OF REAL
w:         ARRAY 16 OF
             RECORD ch: CHAR;
               count: INTEGER
```

t:	Tree

8. Expressions

Expressions are constructs denoting rules of computation whereby constants and current values of variables are combined to derive other values by the application of operators and function procedures. Expressions consist of operands and operators. Parentheses may be used to express specific associations of operators and operands.

8.1. Operands

With the exception of sets and literal constants, i.e. numbers and character strings, operands are denoted by *designators.* A designator consists of an identifier referring to the constant, variable, or procedure to be designated. This identifier may possibly be qualified by module identifiers (see Ch. 4 and 11), and it may be followed by selectors, if the designated object is an element of a structure.

If A designates an array, then A[E] denotes that element of A whose index is the current value of the expression E. The type of E must be an integer type. A designator of the form A[E1, E2, ... , En] stands for A[E1][E2] ... [En]. If p designates a pointer variable, p↑ denotes the variable which is referenced by p. If r designates a record, then r.f denotes the field f of r. If p designates a pointer, p.f denotes the field f of the record p↑, i.e. the dot implies dereferencing and p.f stands for p↑.f, and p[E] denotes the element of p↑ with index E.

The *typeguard* v(T0) asserts that v is of type T0, i.e. it aborts program execution, if it is not of type T0. The guard is applicable, if

1. T0 is an extension of the declared type T of v, and if
2. v is a variable parameter of record type or v is a pointer. In the latter case, condition 1. applies to the pointer base types of T and T0 rather than to T and T0 themselves.

```
$    designator = qualident {"." ident | "[" ExpList "]" | "(" qualident ")" | "↑" }.
$    ExpList = expression {"," expression}.
```

If the designated object is a variable, then the designator refers to the variable's current value. If the object is a procedure, a designator without parameter list refers to that procedure. If it is followed by a (possibly empty) parameter list, the designator implies an activation of the procedure and stands for the value resulting from its execution. The (types of the) actual parameters must correspond to the formal parameters as specified in the procedure's declaration (see Ch. 10).

Examples of designators (see examples in Ch. 7):

i	(INTEGER)
a[i]	(REAL)
w[3].ch	(CHAR)
t.key	(INTEGER)
t.left.right	(Tree)
t(CenterNode).subnode	(Tree)

8.2. Operators

The syntax of expressions distinguishes between four classes of operators with different precedences (binding strengths). The operator ~ has the highest precedence, followed by multiplication operators, addition operators, and relations. Operators of the same precedence associate from left to right. For example, x–y–z stands for (x–y)–z.

```
$    expression = SimpleExpression [relation SimpleExpression].
$    relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
$    SimpleExpression = ["+"|"–"] term {AddOperator term}.
$    AddOperator = "+" | "–" | OR .
$    term = factor {MulOperator factor}.
$    MulOperator = "*" | "/" | DIV | MOD | "&" .
$    factor = number | CharConstant | string | NIL | set |
$         designator [ActualParameters] | "(" expression ")" | "~" factor.
$    set = "{" [element {"," element}] "}".
$    element = expression [".." expression].
$    ActualParameters = "(" [ExpList] ")" .
```

The available operators are listed in the following tables. In some instances, several different operations are designated by the same operator symbol. In these cases, the actual operation is identified by the type of the operands.

8.2.1. Logical operators

symbol	result
OR	logical disjunction
&	logical conjunction
~	negation

These operators apply to BOOLEAN operands and yield a BOOLEAN result.

p OR q	stands for	"if p then TRUE, else q"
p & q	stands for	"if p then q, else FALSE"
~ p	stands for	"not p"

8.2.2. Arithmetic operators

symbol	result
+	sum
–	difference
*	product
/	quotient
DIV	integer quotient
MOD	modulus

The operators +, –, *, and / apply to operands of numeric types. The type of the result is that operand's type which includes the other operand's type, except for division (/), where the result is the real type which includes both operand types. When used as operators with a single operand, – denotes sign inversion and + denotes the identity operation.

The operators DIV and MOD apply to integer operands only. They are related by the following formulas defined for any x and y:

$x = (x \text{ DIV } y) * y + (x \text{ MOD } y)$

$0 <= (x \text{ MOD } y) < y$ or $y < (x \text{ MOD } y) <= 0$

8.2.3. Set operators

symbol	result
+	union
–	difference
*	intersection
/	symmetric set difference

The monadic minus sign denotes the complement of x, i.e. –x denotes the set of integers between 0 and MAX(SET) which are not elements of x.

```
x – y    = x * (-y)
x / y    = (x–y) + (y–x)
```

8.2.4. Relations

symbol	relation
=	equal
#	unequal
<	less
<=	less or equal
>	greater
>=	greater or equal
IN	set membership
IS	type test

Relations are Boolean. The ordering relations <, <=, >, and >= apply to the numeric types, CHAR, and character arrays (strings). The relations = and # also apply to the type BOOLEAN and to set, pointer, and procedure types. *x IN s* stands for "x is an element of s". x must be of an integer type, and s of type SET. *v IS T* stands for "v is of type T" and is called a *type test*. It is applicable, if

1. T is an extension of the declared type T0 of v, and if
2. v is a variable parameter of record type or v is a pointer. In the latter case, condition 1. applies to the pointer base types of T and T0 rather than to T and T0 themselves.

Assuming, for instance, that T is an extension of T0 and that v is a designator declared of type T0, then the test "v IS T" determines whether the actually designated variable is (not only a T0, but also) a T.

Examples of expressions (refer to examples in Ch. 7):

1987	(INTEGER)
i DIV 3	(INTEGER)
~p OR q	(BOOLEAN)
(i+j) * (i–j)	(INTEGER)
s – {8, 9, 13}	(SET)
i + x	(REAL)
a[i+j] * a[i–j]	(REAL)
(0<=i) & (i<100)	(BOOLEAN)
t.key = 0	(BOOLEAN)
k IN {i .. j–1}	(BOOLEAN)
t IS CenterNode	(BOOLEAN)

9. Statements

Statements denote actions. There are elementary and structured statements. Elementary statements are not composed of any parts that are themselves statements. They are the assignment, the procedure call, and the return and exit statements. Structured statements are composed of parts that are themselves statements. They are used to express sequencing and conditional, selective, and repetitive execution. A statement may also be empty, in which case it denotes no action. The empty statement is included in order to relax punctuation rules in statement sequences.

```
$    statement = [assignment | ProcedureCall |
$         IfStatement | CaseStatement | WhileStatement | RepeatStatement |
$         LoopStatement | WithStatement | EXIT | RETURN [expression] ].
```

9.1. Assignments

The assignment serves to replace the current value of a variable by a new value specified by an expression. The assignment operator is written as ":=" and pronounced as *becomes.*

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

The type of the expression must be included by the type of the variable, or it must extend the type of the variable. The following exceptions hold:

1. The constant NIL can be assigned to variables of any pointer type.

2. Strings can be assigned to any variable whose type is an array of characters, provided the length of the string is less than that of the array. If a string s of length n is assigned to an array a , the result is a[i] = si for i = 0 ... n−1, and a[n] = 0X.

3. Values of the types CHAR and SHORTINT can be assigned to variables of type BYTE.

Examples of assignments (see examples in Ch. 7):

```
i := 0
p := i = j
x := i + 1
k := log2(i+j)
F := log2
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y) * (x−y)
t.key := i
w[i+1].ch := "A"
```

9.2. Procedure calls

A procedure call serves to activate a procedure. The procedure call may contain a list of actual parameters which are substituted in place of their corresponding formal parameters defined in the procedure declaration (see Ch. 10). The correspondence is established by the positions of the parameters in the lists of actual and formal parameters respectively. There exist two kinds of parameters: *variable* and *value parameters.*

In the case of variable parameters, the actual parameter must be a designator denoting a variable. If it designates an element of a structured variable, the selector is evaluated when the formal/actual parameter substitution takes place, i.e. before the execution of the procedure. If the parameter is a value parameter, the corresponding actual parameter must be an expression. This expression is evaluated prior to the procedure activation, and the resulting value is assigned to the formal parameter which now constitutes a local variable (see also 10.1.).

```
$    ProcedureCall = designator [ActualParameters].
```

Examples of procedure calls:

```
ReadInt(i)          (see Ch. 10)
WriteInt(j*2+1, 6)
INC(w[k].count)
```

9.3. Statement sequences

Statement sequences denote the sequence of actions specified by the component statements which are separated by semicolons.

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

9.4. If statements

```
$    IfStatement =      IF expression THEN StatementSequence
$                       {ELSIF expression THEN StatementSequence}
$                       [ELSE StatementSequence]
$                       END.
```

If statements specify the conditional execution of guarded statements. The Boolean expression preceding a statement is called its *guard*. The guards are evaluated in sequence of occurrence, until one evaluates to TRUE, whereafter its associated statement sequence is executed. If no guard is satisfied, the statement sequence following the symbol ELSE is executed, if there is one.

Example:

```
IF (ch >= "A") & (ch <= "Z") THEN ReadIdentifier
ELSIF (ch >= "0") & (ch <= "9") THEN ReadNumber
ELSIF ch = 22X THEN ReadString
ELSE SpecialCharacter
END
```

9.5. Case statements

Case statements specify the selection and execution of a statement sequence according to the value of an expression. First the case expression is evaluated, then the statement sequence is executed whose case label list contains the obtained value. The case expression and all labels must be of the same type, which must be an integer type or CHAR. Case labels are constants, and no value must occur more than once. If the value of the expression does not occur as a label of any case, the statement sequence following the symbol ELSE is selected, if there is one. Otherwise it is considered as an error.

```
$    CaseStatement =    CASE expression OF case {"|" case} [ELSE StatementSequence] END.
$    case =             [CaseLabelList ":" StatementSequence].
$    CaseLabelList =    CaseLabels {"," CaseLabels}.
$    CaseLabels =       ConstExpression [".." ConstExpression].
```

Example:

```
CASE ch OF
   "A" .. "Z":    ReadIdentifier
 | "0" .. "9":    ReadNumber
 | 22X :          ReadString
ELSE              SpecialCharacter
END
```

9.6. While statements

While statements specify repetition. If the Boolean expression (guard) yields TRUE, the statement sequence is executed. The expression evaluation and the statement execution are repeated as long as the Boolean expression yields TRUE.

```
$    WhileStatement =   WHILE expression DO StatementSequence END.
```

Examples:

```
WHILE j > 0 DO
   j := j DIV 2; i := i+1
END

WHILE (t # NIL) & (t.key # i) DO
   t := t.left
END
```

9.7. Repeat Statements

A repeat statement specifies the repeated execution of a statement sequence until a condition is satisfied. The statement sequence is executed at least once.

```
$    RepeatStatement =   REPEAT StatementSequence UNTIL expression.
```

9.8. Loop statements

A loop statement specifies the repeated execution of a statement sequence. It is terminated by the execution of any exit statement within that sequence (see 9.9).

```
$     LoopStatement = LOOP StatementSequence END.
```

Example:

```
LOOP
  IF t1 = NIL THEN EXIT END ;
  IF k < t1.key THEN t2 := t1.left; p := TRUE
  ELSIF k > t1.key THEN t2 := t1.right; p := FALSE
  ELSE EXIT
  END ;
  t1 := t2
END
```

Although while and repeat statements can be expressed by loop statements containing a single exit statement, the use of while and repeat statements is recommended in the most frequently occurring situations, where termination depends on a single condition determined either at the beginning or the end of the repeated statement sequence. The loop statement is useful to express cases with several termination conditions and points.

9.9. Return and exit statements

A return statement consists of the symbol RETURN, possibly followed by an expression. It indicates the termination of a procedure, and the expression specifies the result of a function procedure. Its type must be identical to the result type specified in the procedure heading (see Ch. 10).

Function procedures require the presence of a return statement indicating the result value. There may be several, although only one will be executed. In proper procedures, a return statement is implied by the end of the procedure body. An explicit return statement therefore appears as an additional (probably exceptional) termination point.

An exit statement consists of the symbol EXIT. It specifies termination of the enclosing loop statement and continuation with the statement following that loop statement. Exit statements are contextually, although not syntactically bound to the loop statement which contains them.

9.10. With statements

If a pointer variable or a variable parameter with record structure is of a type T0, it may be designated in the heading of a with clause together with a type T that is an extension of T0. Then this variable is treated within the with statement as if it had been declared of type T. The with statement assumes a role similar to the type guard, extending the guard over an entire statement sequence. It may be regarded as a *regional type guard.*

```
$      WithStatement = WITH qualident ":" qualident DO StatementSequence END .
```

Example:

```
WITH t: CenterNode DO name := t.name; L := t.subnode END
```

10. Procedure declarations

Procedure declarations consist of a *procedure heading* and a *procedure body.* The heading specifies the procedure identifier, the *formal parameters,* and the result type (if any). The body contains declarations and statements. The procedure identifier is repeated at the end of the procedure declaration.

There are two kinds of procedures, namely *proper procedures* and *function procedures.* The latter are activated by a function designator as a constituent of an expression, and yield a result that is an operand in the expression. Proper procedures are activated by a procedure call. The function procedure is distinguished in the declaration by indication of the type of its result following the parameter list. Its body must contain a RETURN statement which defines the result of the function procedure.

All constants, variables, types, and procedures declared within a procedure body are *local* to the procedure. The values of local variables are undefined upon entry to the procedure. Since procedures may be declared as local objects too, procedure declarations may be nested.

In addition to its formal parameters and locally declared objects, the objects declared in the environment of the procedure are also visible in the procedure (with the exception of those objects that have the same name as an object declared locally).

The use of the procedure identifier in a call within its declaration implies recursive activation of the procedure.

```
$	ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
$	ProcedureHeading = PROCEDURE ["*"] identdef [FormalParameters].
$	ProcedureBody = DeclarationSequence [BEGIN StatementSequence] END.
$	ForwardDeclaration = PROCEDURE "↑" identdef [FormalParameters].
$	DeclarationSequence = {CONST {ConstantDeclaration ";"} |
$	  TYPE {TypeDeclaration ";"} | VAR {VariableDeclaration ";"}}
$	  {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
```

A *forward declaration* serves to allow forward references to a procedure that appears later in the text in full. The actual declaration – which specifies the body – must indicate the same parameters and result type (if any) as the forward declaration, and it must be within the same scope. An asterisk following the symbol PROCEDURE is a hint to the compiler and specifies that the procedure is to be usable as parameter and assignable to variables of a compatible procedure type.

10.1. Formal parameters

Formal parameters are identifiers which denote actual parameters specified in the procedure call. The correspondence between formal and actual parameters is established when the procedure is called. There are two kinds of parameters, namely *value* and *variable parameters*. The kind is indicated in the formal parameter list. Value parameters stand for local variables to which the result of the evaluation of the corresponding actual parameter is assigned as initial value. Variable parameters correspond to actual parameters that are variables, and they stand for these variables. Variable parameters are indicated by the symbol VAR, value parameters by the absence of the symbol VAR. A function procedure without parameters must have an empty parameter list. It must be called by a function designator whose actual parameter list is empty too.

Formal parameters are local to the procedure, i.e. their scope is the program text which constitutes the procedure declaration.

```
$	FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" qualident].
$	FPSection = [VAR] ident {"," ident} ":" FormalType.
$	FormalType = {ARRAY OF} qualident.
```

The type of each formal parameter is specified in the parameter list. For variable parameters, it must be identical to the corresponding actual parameter's type, except in the case of a record, where it must be a base type of the corresponding actual parameter's type. For value parameters, the rule of assignment holds (see 9.1). If the formal parameter's type is specified as

```
ARRAY OF T
```

the parameter is said to be an *open array parameter*, and the corresponding actual parameter may be any array with the element type T.

In the case of a parameter with formal type BYTE, the corresponding actual parameter may be of type CHAR or SHORTINT. If the formal type of a variable parameter is ARRAY OF BYTE, any actual parameter type is permitted.

If a formal parameter specifies a procedure type, then the corresponding actual parameter must be either a procedure declared at level 0 or a variable (or parameter) of that procedure type. It cannot be a predefined procedure. The result type of a procedure can be neither a record nor an array.

Examples of procedure declarations:

```
PROCEDURE ReadInt(VAR x: INTEGER);
  VAR i : INTEGER; ch: CHAR;
BEGIN i := 0; Read(ch);
  WHILE ("0" <= ch) & (ch <= "9") DO
     i := 10*i + (ORD(ch)-ORD("0")); Read(ch)
  END ;
  x := i
END ReadInt

PROCEDURE WriteInt(x: INTEGER);  (* 0 <= x < 10↑5 *)
  VAR i: INTEGER;
     buf: ARRAY 5 OF INTEGER;
BEGIN i := 0;
  REPEAT buf[i] := x MOD 10;  x := x DIV 10;  INC(i) UNTIL x = 0;
  REPEAT DEC(i); Write(CHR(buf[i] + ORD("0"))) UNTIL i = 0
END WriteInt

PROCEDURE log2(x: INTEGER): INTEGER;
  VAR y: INTEGER;  (*assume x>0*)
BEGIN y := 0;
  WHILE x > 1 DO x := x DIV 2; INC(y) END ;
  RETURN y
END log2
```

10.2. Predefined procedures

The following table lists the predefined procedures. Some are *generic* procedures, i.e. they apply to several types of operands. v stands for a variable, x and n for expressions, and T for a type.

Function procedures:

Name	Argument type	Result type	Function
ABS(x)	numeric type	type of x	absolute value
ODD(x)	integer type	BOOLEAN	x MOD 2 = 1
CAP(x)	CHAR	CHAR	corresponding capital letter
ASH(x, n)	x, n: integer type	LONGINT	$x * 2^n$, arithmetic shift
LEN(v, n)	v: array n: integer type	LONGINT	the length of v in dimension n
LEN(v)	is equivalent with LEN(v, 0)		

MAX(T)	T = basic type
	T = SET
		T
		INTEGER
			maximum value of type T
			maximum element of sets

MIN(T)	T = basic type
	T = SET
		T
		INTEGER
			minimum value of type T
			0

Type conversion procedures:

Name	Argument type	Result type	Function
ORD(x)	CHAR, BYTE	INTEGER	ordinal number of x
CHR(x)	integer type, BYTE	CHAR	character with ordinal number x
SHORT(x)	LONGINT INTEGER LONGREAL	INTEGER SHORTINT REAL	identity (truncation possible)
LONG(x)	SHORTINT INTEGER REAL	INTEGER LONGINT LONGREAL	identity
ENTIER(x)	real type	LONGINT	largest integer not greater than x

Note that ENTIER(i/j) = i DIV j

Proper procedures:

Name	Argument types	Function
INC(v)	integer type	v := v+1
INC(v, x)	integer type	v := v+x
DEC(v)	integer type	v := v−1
DEC(v, x)	integer type	v := v−x
INCL(v, x)	v: SET; x: integer type	v := v + {x}
EXCL(v, x)	v: SET; x: integer type	v := v − {x}
COPY(x, v)	x: character array, string v: character array	v := x
NEW(v)	pointer type	allocate v↑
HALT(x)	integer constant	terminate program execution

The second parameter of INC and DEC may be omitted, in which case its default value is 1. In HALT(x), x is a parameter whose interpretation is left to the underlying system implementation.

11. Modules

A module is a collection of declarations of constants, types, variables, and procedures, and a sequence of statements for the purpose of assigning initial values to the variables. A module typically constitutes a text that is compilable as a unit.

```
$    module = MODULE ident ";" [ImportList] DeclarationSequence
$        [BEGIN StatementSequence] END ident "." .
$      ImportList = IMPORT import {"," import} ";" .
$      import = identdef [":" ident].
```

The import list specifies the modules of which the module is a client. If an identifier x is exported from a module M, and if M is listed in a module's import list, then x is referred to as M.x. If the form "M : M1" is used in the import list, that object declared within M1 is referenced as M.x .

Identifiers that are to be visible in client modules, i.e. outside the declaring module, must be marked by an export mark in their declaration. If a type imported from a module M is used in the specification of an exported object, (e.g. in its type or in its heading, but not in a procedure body), then also M must be marked in the import list.

The statement sequence following the symbol BEGIN is executed when the module is added to a system (loaded). Individual (parameterless) procedures can thereafter be activated from the system, and these procedures serve as *commands.*

Example:

```
MODULE Out;
   (*exported procedures:  Write, WriteInt, WriteLn*)
   IMPORT Texts, Oberon;

   VAR W: Texts.Writer;

   PROCEDURE Write*(ch: CHAR);
   BEGIN Texts.Write(W, ch)
   END ;

   PROCEDURE WriteInt*(x, n: LONGINT);
      VAR i: INTEGER; a: ARRAY 16 OF CHAR;
   BEGIN i := 0;
      IF x < 0 THEN Texts.Write(W, "-"); x := -x END ;
      REPEAT a[i] := CHR(x MOD 10 + ORD("0")); x := x DIV 10; INC(i) UNTIL x = 0;
      REPEAT Texts.Write(W, " "); DEC(n) UNTIL n <= i;
      REPEAT DEC(i); Texts.Write(W, a[i]) UNTIL i = 0
   END WriteInt;

   PROCEDURE WriteLn*;
   BEGIN Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
   END WriteLn;

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

Appendix: The Module SYSTEM

The module SYSTEM contains certain procedures that are necessary to program *low-level* operations referring directly to objects particular to a given computer and/or implementation. These include for example facilities for accessing devices that are controlled by the computer, and facilities to break the data type compatibility rules otherwise imposed by the language definition. It is recommended to restrict their use to specific modules (called *low-level* modules). Such modules are inherently non-portable, but easily recognized due to the identifier SYSTEM appearing in their import lists. The following specifications hold for the ETH implementation for the NS32000 processor.

The procedures contained in module SYSTEM are listed in the following tables. They correspond to single instructions compiled as in–line code. For details, the reader is referred to the processor manual. v stands for a variable, x, y, a, and n for expressions, and T for a type.

Function procedures:

Name	Argument types	Result type	Function
ADR(v)	any	LONGINT	address of variable v
BIT(a, n)	a: LONGINT n: integer type	BOOLEAN	Mem[a][n]
CC(n)	integer constant	BOOLEAN	Condition n (0 <= n < 16)
LSH(x, n)	x, n: integer type	LONGINT	logical shift
ROT(x, n)	x, n: integer type	LONGINT	rotation
SIZE(T)	any type	integer type	number of bytes required by T
VAL(T, x)	T, x: any type	T	x interpreted as of type T

Proper procedures:

Name	Argument types	Function
GET(a, v)	a: LONGINT; v: any basic type	v := Mem[a]
PUT(a, x)	a: LONGINT; x: any basic type	Mem[a] := x
MOVE(v0, v1, n)	v0, v1: any type; n: integer type	assign first n bytes of v0 to v1
NEW(v, n)	v: any pointer type n: integer type	allocate storage block of n bytes assign its address to v

File: Oberon2.Report.Doc / NW 30.8.89