# The Programming Language Oberon

**Revision 8.8.2007**

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.

This document describes the language defined in 1988/90 as revised in July 2007. Changes and additions appear in **blue**.

## 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.

## 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 is pronounced as "times ten to the power of". A real number is of type REAL.

```
number = integer | real.
integer = digit {digit} | digit {hexDigit} "H" .
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = "E" ["+" | "-"] 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 |

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.

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

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. They may be nested.

## 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) |
| BOOLEAN | (6.1) | LONGINT | (6.1) |
| BYTE | (6.1) | LONGREAL | (6.1) |
| CHAR | (6.1) | NEW | (6.4) |
| CHR | (10.2) | ODD | (10.2) |
| DEC | (10.2) | ORD | (10.2) |
| INC | (10.2) | REAL | (6.1) |
| INTEGER | (6.1) | SET | (6.1) |
| ASR | (10.2) | LSL | (10.2) |
| ASSERT | (10.2) | LSR | (10.2) |
| FLOOR | (10.2) | PACK | (10.2) |
| FLT | (10.2) | UNPK | (10.2) |

## 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 "=" StrucType.
```

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

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 Latin-1 set. |
| 3. | INTEGER | the integers between $-2^{31}$ and $+2^{31}-1$. |
| 4. | LONGINT | integers, range includes that of type INTEGER. |
| 5. | REAL | real numbers (IEEE Standard, 32 bits). |
| 6. | LONGREAL | long real numbers (IEEE Standard, 64 bits). |
| 5. | SET | the sets of integers between 0 and 31. |

### 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.* 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* and 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 (or NIL) 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 parameters 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 standard 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
              END
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*.

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.

```
designator  =  qualident {selector}.
selector  =  "." 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 | CharConst | string | NIL | TRUE | FALSE |
                     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. Let q = x DIV y, and r = x MOD y. Then quotient *q* and remainder *r* are defined by the equations

$x = q*y + r \qquad 0 \le r < y$

*8.2.3. Set operators*

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

When used with a single operand of type SET, the minus sign denotes the set complement.

*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 <, <=, >, >= 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. The relations <= and >= denote inclusion when applied to sets. *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.

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. The value of *NIL IS T* is undefined.

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 | ForStatement | WithStatement].
```

## 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 or procedure type.
2. Arrays must have the same element type, and the length of the destination array must not be less than the length of the source array.
3. Strings can be assigned to any array of characters, provided the length of the string is less than that of the array. (A string is automatically terminated by a 0X character).
4. In the case of records, the type of the destination must be an extension of the type of the source.

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(2*j + 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
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 must be of type INTEGER, and all labels must be integers. The labels range from 0 to *n*-1, where *n* is specified as a constant in the case clause.

```
CaseStatement =   CASE expression “:” ConstExpression OF {"|" case} END.
case          =   [CaseLabelList ":" StatementSequence].
CaseLabelList =   labels {"," labels}.
labels        =   integer [“..” integer].
```

Example:

```
CASE k : 4 OF
    | 0: x := x + y
    | 1: x := x – y
    | 2: x := x * y
    | 3: x := x / y
END
```

### 9.6. While statements

While statements specify repetition. If any of the Boolean expressions (guards) yields TRUE, the corresponding statement sequence is executed. The expression evaluation and the statement execution are repeated until none of the Boolean expressions yields TRUE.

```
WhileStatement  =  WHILE expression DO StatementSequence
         {ELSIF 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
```

```
WHILE m > n DO m := m – n
ELSIF n > m DO n := n – m
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. For statements

A for statement specifies the repeated execution of a statement sequence for a given number of times, while a progression of values is assigned to an integer variable called the *control variable* of the for statement.

```
ForStatement =
  FOR ident “:=“ expression TO expression [BY ConstExpression] DO
  StatementSequence END .
```

The for statement

```
FOR v := beg TO end BY inc DO S END
```

is, if *inc* > 0, equivalent to

```
v := beg; lim := end;
WHILE v <= lim DO S; v := v + inc END
```

and if *inc* < 0 it is equivalent to

```
v := beg; lim := end;
WHILE v >= lim DO S; v := v + inc END
```

*beg* and *end* must be of type INTEGER, and *inc* must be an integer (constant expression). If the step is not specified, it is assumed to be 1.

### 9.9. 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 the variable is guarded 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 end with a RETURN clause 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 | IntSpex].
ProcedureBody = DeclarationSequence [BEGIN StatementSequence]
          [ ";" RETURN expression] END.
DeclarationSequence = [CONST {ConstantDeclaration ";"}]
          [TYPE {TypeDeclaration ";"}] [VAR {VariableDeclaration ";"}]
          {ProcedureDeclaration ";"}.
```

### 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 three kinds of parameters, namely *value, constant,* and *variable* parameters. A variable parameter corresponds to an actual parameter that is a variable, and it stands for this variable. A constant parameter corresponds to an actual parameter that is an expression, and it stands for its value, which cannot be changed by assignment. A value parameter represents a local variable to which the value of the actual expression is assigned. The kind of a parameter is indicated in the formal parameter list: Variable parameters are denoted by the symbol VAR, constant parameters by the symbol CONST, and value parameters by the absence of such a prefix.

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 = [CONST | 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. Value parameters must not be of an array or record type.

If the formal parameter's type is specified as

```
ARRAY OF T
```

the parameter is said to be an *open array*, and the corresponding actual parameter may be of arbitrary length.

If a formal parameter specifies a procedure type, then the corresponding actual parameter must be either a procedure declared globally, 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) | INTEGER or REAL | type of x | absolute value |
| | SET | INTEGER | no. of elements |
| ODD(x) | INTEGER | BOOLEAN | x MOD 2 = 1 |
| LEN(v) | v: array | | the length of v |
| LSL(x, n) | x: INTEGER | type of x | logical shift left, x * $2^n$ |
| LSR(x, n) | x: INTEGER | type of x | unsigned shift right, x * $2^{-n}$ |
| ASR(x, n) | x: INTEGER | type of x | signed shift right, x * $2^{-n}$ |

*Type conversion procedures:*

| Name | Argument type | Result type | Function |
|---|---|---|---|
| FLOOR(x) | REAL | INTEGER | x with fractional part removed |
| FLT(x) | INTEGER | REAL | identity, type transfer |
| ORD(x) | CHAR | INTEGER | ordinal number of x |
| CHR(x) | INTEGER | CHAR | character with ordinal number x |

*Proper procedures:*

| Name | Argument types | Function |
|---|---|---|
| INC(v) | INTEGER | v := v + 1 |
| INC(v, n) | INTEGER | v := v + n |
| DEC(v) | INTEGER | v := v - 1 |
| DEC(v, n) | INTEGER | v := v - n |
| NEW(v) | pointer type | allocate v^ |
| ASSERT(b) | BOOLEAN | abort, if ~b |
| ASSERT(b, n) | BOOLEAN, INTEGER | |
| PACK(x, y) | REAL, INTEGER | pack x and y into x |
| UNPK(x, y) | REAL, INTEGER | unpack x into x and y |

Procedures INC and DEC may have an explicit increment or decrement. It must be a constant. The second parameter *n* of ASSERT is a value transmitted to the system as an abort parameter.

The parameter y of PACK represents the exponent of x. PACK(x, y) is equivalent to x := x * $2^y$. UNPK is the reverse operation of PACK. The resulting x is normalized, i.e. 1.0 <= x < 2.0.

### 10.3. Leaf procedures

If a procedure is not calling any other procedure, it is called a *leaf procedure*. This may be indicated by following the symbol *procedure* by an asterisk. It serves as a compiler directive, and it causes the compiler to let parameters remain allocated in registers (of which there is only a small number). Furthermore, the variables declared first in the list are also allocated in registers, if possible. Only variables of type INTEGER or SET are considered. This facility serves to speed up execution.

### 10.4. Interrupt procedures

Interrupt procedures are those that are activated by an interrupt signal. They are denoted by an additional specification in place of a parameter list. They have neither parameters nor a result type, and they act as a command (see Ch. 11).

```
IntSpex  =  “[“ integer “]”.
```

The IntSpex are processor-dependent. For the ARM processor, for example, the integer is the offset in the return instruction. Here, the asterisk after the symbol PROCEDURE has the additional effect that no registers are saved. This option is used in the case of the “fast interrupt”, where the ARM-processor uses a different set of registers R8 – R15 for fast interrupts.

## 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      =  ident [":=" 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, an exported object x declared within M1 is referenced in the importing module as M.x .

Identifiers that are to be visible in client modules, i.e. which are to be exported, must be marked by an asterisk (export mark) in their declaration.

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.
```

## 12. The Module SYSTEM

The module SYSTEM contains definitions that are necessary to program low-level operations referring directly to resources 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 low-level modules. Such modules are inherently non-portable, but easily recognized due to the identifier SYSTEM appearing in their import lists. The subsequent definitions are applicable to most modern computers; however, individual implementations may include in this module definitions that are particular to the specific, underlying computer. The following applies to the current implementation for the ARM processor.

Module SYSTEM exports the data type BYTE. No representation of values is specified. It is used to relax the type compatibility rules for procedure parameters. If a formal parameter is of type ARRAY OF SYSTEM.BYTE, then the corresponding actual parameter may be of any type.

The procedures contained in module SYSTEM are listed in the following tables. They correspond to short instruction sequences compiled as in-line code. v stands for a variable, x, y, a, and n for expressions.

*Function procedures:*

| Name | Argument types | Result type | Function |
|---|---|---|---|
| ADR(v) | any | INTEGER | address of variable v |
| SIZE(T) | any type | INTEGER | size in bytes |
| BIT(a, n) | a, n: INTEGER | BOOLEAN | bit n of Mem[a] |
| ROR(x, n) | x: INTEGER | type of x | rotate right |
| VAL(T, x) | T, x: any type | T | x cast into type T |

*Proper procedures:*

| Name | Argument types | Function |
|---|---|---|
| GET(a, v) | a: INTEGER; v: any basic type | v := Mem[a] |

| | | |
|---|---|---|
| PUT(a, x) | a: INTEGER; x: any basic type | Mem[a] := x |
| LDPSR(b, x) | b, x: INTEGER | load status register with x |
| STPSR(b, v) | b, v: INTEGER | store status register in v |
| LDCPR(p, n, x) | p, n, x: INTEGER | load register n of coprocessor p with x |
| STCPR(p, n, v) | p, n, v: INTEGER | load register n of coprocessor p in v |

## Appendix

## The Syntax of Oberon-07

```
letter  =  ...
digit  =  ...
hexdigit  =  ...
ident  =  letter {letter | digit}.
integer  =  digit {digit} | digit {hexDigit} "H".
real  =  digit {digit} "." {digit} [ScaleFactor].
ScaleFactor  =  "E" ["+" | "-"] digit {digit}.
number  =  integer | real.
charConst  =  """ character """ | digit {hexDigit} "X".
string  =  """ {character} """.
qualident  =  ident ["." ident].
identdef = ident ["*"].

ConstantDeclaration  =  identdef "=" ConstExpression.
ConstExpression  =  expression.
TypeDeclaration  =  identdef "=" StrucType.
StrucType  =  ArrayType | RecordType | PointerType | ProcedureType.
type  =  ident | StrucType.
ArrayType  =  "ARRAY" length {"," length} "OF" type.
Length  =  ConstExpression.
RecordType  =  "RECORD" ["(" BaseType ")"] [FieldListSequence] "END".
BaseType  =  qualident.
FieldListSequence  =  FieldList {";" FieldList}.
FieldList  =  IdentList ":" type.
IdentList  =  identdef {"," identdef}.
PointerType  =  "POINTER" "TO" type.
ProcedureType  =  "PROCEDURE" [FormalParameters].
VariableDeclaration  =  IdentList ":" type.

designator  =  qualident {selector}.
selector  =  "." ident | "[" ExpList "]" | "^" |  "(" qualident ")".
ExpList  =  expression {"," expression}.
factor  =  number | CharConst | string | "NIL" | "TRUE" | "FALSE" |
   set | designator [ActualParameters] | "(" expression ")" | "~" factor.
ActualParameters  =  "(" [ExpList] ")" .
term  =  factor {MulOp factor}.
MulOp  =  "*" | "/" | "DIV" | "MOD" | "&".
SimpleExpression  =  ["+" | "-"] term {AddOp term}.
AddOp  =  "+" | "-" | "OR".
expression  =  SimpleExpression [relation SimpleExpression].
relation  =  "=" | "#" | "<" | "<=" | ">" | ">=" | "IN" | "IS".
set  =  "{" [element {"," element}] "}".
element  =  expression [".." expression].

statement  =  [assignment | ProcedureCall | IfStatement | CaseStatement |
   WhileStatement | RepeatStatement | ForStatement | WithStatement].
assignment  =  designator ":=" expression.
ProcedureCall  =  designator [ActualParameters].
ActualParameters  =  "(" [expression {"," expression}] ")" .
StatementSequence  =  statement {";" statement}.
IfStatement  =  "IF" expression "THEN" StatementSequence
   {"ELSIF" expression "THEN" StatementSequence}
   ["ELSE" StatementSequence] "END".
CaseStatement  =  "CASE" expression ":" ConstExpression "OF" {"|" case} "END".
Case  =  CaseLabelList ":"  StatementSequence.
CaseLabelList  = labels {","labels}.
```

```
labels  =   integer [“..” integer].
WhileStatement  =  "WHILE" expression "DO" StatementSequence
    {“ELSIF” expression "DO" StatementSequence} "END".
RepeatStatement  =  "REPEAT" StatementSequence "UNTIL" expression.
ForStatement  =  "FOR" ident ":=" expression "TO" expression ["BY" ConstExpression] "DO"
    StatementSequence "END".
WithStatement  =  “WITH” ident “:” qualident “DO” StatementSequence "END".

ProcedureDeclaration  =  ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading  =  "PROCEDURE" ["*"] identdef [FormalParameters | IntSpex].
ProcedureBody  =  DeclarationSequence ["BEGIN" StatementSequence]
    ["RETURN" expression] "END".
DeclarationSequence  =  ["CONST" {ConstDeclaration ";"}]
    ["TYPE" {TypeDeclaration ";"}]
    ["VAR" {VariableDeclaration ";"}]
    {ProcedureDeclaration ";"}.
FormalParameters  =  "(" [FPSection {";" FPSection}] ")" [":" qualident].
FPSection  =  ["CONST" | "VAR"] ident {"," ident} ":" FormalType.
FormalType  =  ["ARRAY" "OF"] qualident.
IntSpex  =  "[" integer "]".

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