The Programming Language Oberon

Revision 1.10.2013 / 4.3.2016

Niklaus Wirth

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

Table of Contents

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 unnecessarily restrict the freedom of implementors.

This document describes the language defined in 1988/90 as revised in 2007/11.

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 by words in capital letters.

3. Vocabulary

The following lexical rules must be observed when composing symbols. 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.

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

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

Examples:

```
x   scan   Oberon   GetSymbol   firstLetter
```

Numbers are (unsigned) integers or real numbers. Integers are sequences of digits and may be followed by a suffix letter. 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".

```
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
```

Strings are sequences of characters enclosed in quote marks ("). A string cannot contain the delimiting quote mark. Alternatively, a single-character string may be specified by the ordinal number of the character in hexadecimal notation followed by an "X". The number of characters in a string is called the *length* of the string.

```
string  =  """ {character} """ | digit {hexdigit} "X" .
```

Examples:

```
"OBERON"          "Don't worry!"          22X
```

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

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.

In its declaration, an identifier in the module's 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 section 6.1 (types) or 10.2 (procedures):

ABS	ASR	ASSERT	BOOLEAN	BYTE
CHAR	CHR	DEC	EXCL	FLOOR
FLT	INC	INCL	INTEGER	LEN
LSL	NEW	ODD	ORD	PACK
REAL	ROR	SET	UNPK	

5. Constant declarations

A constant declaration associates an identifier with a constant value.

```
ConstDeclaration  =  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}
name    =   "Oberon"
```

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 a type. The types define the structure of variables of this type and, by implication, the operators that are applicable to 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:

BOOLEAN	the truth values TRUE and FALSE
CHAR	the characters of a standard character set
INTEGER	the integers
REAL	real numbers
BYTE	the integers between 0 and 255
SET	the sets of integers between 0 and an implementation-dependent limit

The type BYTE is compatible with the type INTEGER, and vice-versa.

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

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*. It must be a record type. 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 there is any. If a type *T* is an extension of *T0* and *P* is a pointer type bound to *T*, then *P* is also an extension of *P0, the pointer type bound to T0*.

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

If a type *P* is defined as POINTER TO T, the identifier *T* can be declared textually following the declaration of *P*, but [if so] it must lie within the same scope.

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 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 of type INTEGER. 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 | string | NIL | TRUE | FALSE |
                    set | designator [ActualParameters] | "(" expression ")" | "~" factor.
set              =  "{" [element {"," element}] "}".
element          =  expression [".." expression].
ActualParameters    =  "(" [ExpList] ")" .
```

The set {m .. n} denotes {m, m+1, ... , n-1, n}, and if m > n, the empty set. 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. Both operands must be of the same type, which is also the type of the result. When used as unary operators, – 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 equation

x = q*y + r 0 <= 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. The relations = and # also apply to the types BOOLEAN, SET, and to pointer and procedure types.

x IN s stands for "x is an element of s". *x* must be of type INTEGER, 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)
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 and the procedure call. 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].

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.

If a value parameter is structured (of array or record type), no assignment to it or to its elements are permitted. Neither may assignments be made to imported variables.

The type of the expression must be the same as that of the designator. The following exceptions hold:

1. The constant NIL can be assigned to variables of any pointer or procedure type.
2. Strings can be assigned to any array of characters, provided the number of characters in the string is not greater than that of the array. If it is less, a null character (0X) is appended. Single-character strings can also be assigned to variables of type CHAR.
3. In the case of records, the type of the source must be an extension of the type of the destination.

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

```
i := 0
p := i = j
x := FLT(i + 1)
k := (i + j) DIV 2
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. If the case expression is of type INTEGER or CHAR, all labels must be integers or single-character strings, respectively.

```
CaseStatement =  CASE expression OF case {"|" case} END.
case          =  [CaseLabelList ":" StatementSequence].
CaseLabelList =  LabelRange {"," LabelRange}.
LabelRange    =  label [".." label].
label         =  integer | string | qualident.
```

Example:

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

The type *T* of the case expression (case variable) may also be a record or pointer type. Then the case labels must be extensions of *T*, and in the statements *Si* labelled by *Ti*, the case variable is considered as of type *Ti*.

Example:

```
TYPE R = RECORD a: INTEGER END ;
    R0 = RECORD (R) b: INTEGER END ;
    R1 = RECORD (R) b: REAL END ;
    R2 = RECORD (R) b: SET END ;
    P = POINTER TO R;
    P0 = POINTER TO R0;
    P1 = POINTER TO R1;
    P2 = POINTER TO R2;
VAR p: P;

CASE p OF
    P0: p.b := 10 |
    P1: p.b := 2.5 |
    P2: p.b := {0, 2}
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;
WHILE v <= end DO S; v := v + inc END
```

and if *inc* < 0 it is equivalent to

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

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

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. A 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 globally are also visible in the procedure.

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]
        [RETURN expression] END.
DeclarationSequence = [CONST {ConstDeclaration ";"}]
        [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 two kinds of parameters, namely *value* and *variable* parameters. A variable parameter corresponds to an actual parameter that is a variable, and it stands for that variable. A value parameter corresponds to an actual parameter that is an expression, and it stands for its value, which cannot be changed by assignment. However, if a value parameter is of a basic type, it represents a local variable to which the value of the actual expression is initially assigned.

The kind of a parameter is indicated in the formal parameter list: Variable parameters are denoted by the symbol VAR and value parameters by the absence of 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 = [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.

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)	x: numeric type	type of x	absolute value
ODD(x)	x: INTEGER	BOOLEAN	x MOD 2 = 1
LEN(v)	v: array	INTEGER	the length of v
LSL(x, n)	x, n: INTEGER	INTEGER	logical shift left, $x * 2^n$
ASR(x, n)	x, n: INTEGER	INTEGER	signed shift right, $x \text{ DIV } 2^n$
ROR(x, n)	x, n: INTEGER	INTEGER	x rotated right by n bits

Type conversion functions:

Name	Argument type	Result type	Function
FLOOR(x)	REAL	INTEGER	round down
FLT(x)	INTEGER	REAL	identity
ORD(x)	CHAR, BOOLEAN, SET	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
INCL(v, x)	v: SET; x: INTEGER	v := v + {x}
EXCL(v, x)	v: SET; x: INTEGER	v := v - {x}

NEW(v)	pointer type	allocate v^
ASSERT(b)	BOOLEAN	abort, if ~b
PACK(x, n)	REAL; INTEGER	pack x and n into x
UNPK(x, n)	REAL; INTEGER	unpack x into x and n

The function FLOOR(x) yields the largest integer not greater than x.

FLOOR(1.5) = 1 FLOOR(-1.5) = -2

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

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. Variables are always exported in *read-only* mode.

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: INTEGER);
        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.
```

11.1 The Module SYSTEM

The optional 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 perhaps facilities to break the data type compatibility rules otherwise imposed by the language definition.

There are two reasons for provoding facilites in Module SYSTEM; (1) Their value is implementation-dependent, that is, it is not derivable from the language's definition, and (2) they may corrupt a system (e.g. PUT). It is strongly recommended to restrict their use to specific low-level modules, as such modules are inherently non-portable and not "type-safe". However, they are easily recognized due to the identifier SYSTEM appearing in the module's import lists. The subsequent definitions are generally applicable. However, individual implementations may include in their module SYSTEM additional definitions that are particular to the specific, underlying computer. In the following, *v* stands for a variable, *x, 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]

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
COPY(src, dst, n)	all INTEGER	copy *n* consecutive words from *src* to *dst*

The following are additional procedures accepted by the compiler for the RISC processor:

Function procedures:

Name	Argument types	Result type	Function
VAL(T, n)	scalar	T	identity
ADC(m, n)	INTEGER	INTEGER	add with carry C
SBC(m, n)	INTEGER	INTEGER	subtract with carry C
UML(m, n)	INTEGER	INTEGER	unsigned multiplication
COND(n)	INTEGER	BOOLEAN	IF Cond(8) THEN ...

Proper procedures:

Name	Argument types	Function
LED(n)	INTEGER	display n on LEDs

Appendix

The Syntax of Oberon

```
letter  =  "A" | "B" | … | "Z" | "a" | "b" | … | "z".
digit  =  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
hexDigit  =  digit | "A" | "B" | "C" | "D" | "E" | "F".

ident  =  letter {letter | digit}.
qualident  =  [ident "."] ident.
identdef = ident ["*"].

integer  =  digit {digit} | digit {hexDigit} "H".
real  =  digit {digit} "." {digit} [ScaleFactor].
ScaleFactor  =  ("E" | "D") ["+" | "-"] digit {digit}.
number  =  integer | real.
string  =  """ {character} """ | digit {hexDigit} "X".

ConstDeclaration  =  identdef "=" ConstExpression.
ConstExpression  =  expression.

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

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

statement  =  [assignment | ProcedureCall | IfStatement | CaseStatement |
   WhileStatement | RepeatStatement | ForStatement].
assignment  =  designator ":=" expression.
ProcedureCall  =  designator [ActualParameters].
StatementSequence  =  statement {";" statement}.
IfStatement  =  IF expression THEN StatementSequence
   {ELSIF expression THEN StatementSequence}
```

```
   [ELSE StatementSequence] END.
CaseStatement  =  CASE expression OF case {"|" case} END.
case  =  [CaseLabelList ":"  StatementSequence].
CaseLabelList  = LabelRange {"," LabelRange}.
LabelRange  =  label [".." label].
label  =  integer | string | qualident.
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.

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

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