

# A Oberon-2 Language Definition

## A.1 Introduction

Oberon-2 is a general-purpose language in the tradition of Oberon and Modula-2. Its most important features are block structure, modularity, separate compilation, static typing with strong type checking (also across module boundaries), and type extension with type-bound procedures.

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

Section A.12.1 defines some terms that are used to express the type checking rules of Oberon-2. Where they appear in the text, they are written in italics to indicate their special meaning (e.g., the *same* type).

## A.2 Syntax

An extended Backus-Naur Formalism (EBNF) is used to describe the syntax of Oberon-2: Alternatives are separated by |. Brackets [ and ] denote optionality of the enclosed expression, and braces { and } denote its repetition (possibly 0 times). Nonterminal symbols start with an upper-case letter (e.g., Statement). Terminal symbols either start with a lower-case letter (e.g., ident), or are written all in

upper-case letters (e.g., BEGIN), or are denoted by strings (e.g., ":=").

## A.3 Vocabulary and Representation

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

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

`ident = letter {letter | digit}.`

Examples: `x Scan Oberon2 GetSymbol firstLetter`

2. *Numbers* are (unsigned) integer or real constants. The type of an integer constant is the minimal type to which the constant value belongs (see A.6.1). If the constant is specified with the suffix H, the representation is hexadecimal; otherwise it is decimal.

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

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

Examples:

1991	INTEGER	1991
0DH	SHORTINT	13
12.3	REAL	12.3
4.567E8	REAL	456700000
0.57712566D-6	LONGREAL	0.00000057712566

3. *Character constants* are denoted by the ordinal number of the character in hexadecimal notation followed by the letter X.

character = digit {hexDigit} "X".

4. *Strings* are sequences of characters enclosed in single ('') or double ("") quotation marks. The opening quotation mark must be the same as the closing one and must not occur within the string. The number of characters in a string is called its *length*. A string of length 1 can be used wherever a character constant is allowed and vice versa.

string = ' '' {char} ' '' | " " {char} " ".

Examples: "Oberon-2" "Don't worry!" "x"

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

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

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 may be nested. They do not affect the meaning of a program.

## A.4 Declarations and Scope Rules

Every identifier occurring in a program must be introduced by a declaration unless it is a predeclared identifier. Declarations also 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.

The *scope* of an object *x* extends textually from the point of its declaration to the end of the *block* (module, procedure, or record) to which the declaration belongs and hence to which the object is *local*. It excludes the scopes of objects with the same name that are declared in nested blocks. The scope rules are:

1. No identifier may denote more than one object within a given scope (i.e., no identifier may be declared twice in a block).
2. An object may only be referenced within its scope.
3. A type *T* of the form **POINTER TO T1** (see A.6.4) can be declared before the scope of *T1*. In this case, the declaration of *T1* must follow in the same block to which *T* is local.
4. Identifiers denoting record fields (see A.6.3) or type-bound procedures (see A.10.2) are valid in record designators only.

An identifier declared in a module block may be followed by an export mark (an asterisk or a minus sign) in its declaration to indicate that it is exported. An identifier *x* exported by a module *M* may be used in other modules if they import *M* (see A.11). The identifier is then denoted as *M.x* in these modules and is called a *qualified identifier*. Identifiers marked with a minus in their declaration are *read-only* in importing modules.

**Qualident** = [ident "."] ident.  
**IdentDef** = ident ["\*" | "-"].

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

ABS	(A.10.3)	LEN	(A.10.3)
ASH	(A.10.3)	LONG	(A.10.3)
BOOLEAN	(A.6.1)	LONGINT	(A.6.1)
CAP	(A.10.3)	LONGREAL	(A.6.1)
CHAR	(A.6.1)	MAX	(A.10.3)
CHR	(A.10.3)	MIN	(A.10.3)
COPY	(A.10.3)	NEW	(A.10.3)
DEC	(A.10.3)	ODD	(A.10.3)
ENTIER	(A.10.3)	ORD	(A.10.3)
EXCL	(A.10.3)	REAL	(A.6.1)
FALSE	(A.6.1)	SET	(A.6.1)
HALT	(A.10.3)	SHORT	(A.10.3)
INC	(A.10.3)	SHORTINT	(A.6.1)
INCL	(A.10.3)	SIZE	(A.10.3)
INTEGER	(A.6.1)	TRUE	(A.6.1)

## A.5 Constant Declarations

A constant declaration associates an identifier with a constant value.

```
ConstantDeclaration = IdentDef "=" ConstExpression.
ConstExpression     = Expression.
```

A constant expression is an expression that can be evaluated by a mere textual scan without actually executing the program. Its operands are constants (A.8) or predeclared functions (A.10.3) that can be evaluated at compile time. Examples of constant declarations are:

```
N = 100
limit = 2*N - 1
fullSet = {MIN(SET) .. MAX(SET)}
```

## A.6 Type Declarations

A data type determines the set of values that variables of that type may assume, and the operators that are applicable. A type declaration associates an identifier with a type. In the case of structured types (arrays and records) it also defines the structure of variables of this type.

```
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

CenterTree = POINTER TO CenterNode
CenterNode = RECORD (Node)
            width: INTEGER;
            subnode: Tree
          END

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

### A.6.1 Basic Types

The basic types are denoted by predeclared identifiers. The associated operators are defined in A.8.2 and the predeclared function procedures in A.10.3. The values of the given basic types are the following:

BOOLEAN	truth values TRUE and FALSE
CHAR	characters of the extended ASCII set (0X .. 0FFX)
SHORTINT	integers between MIN(SHORTINT) and MAX(SHORTINT)
INTEGER	integers between MIN(INTEGER) and MAX(INTEGER)
LONGINT	integers between MIN(LONGINT) and MAX(LONGINT)
REAL	real numbers between MIN(REAL) and MAX(REAL)
LONGREAL	real numbers betw. MIN(LONGREAL) and MAX(LONGREAL)
SET	sets of integers between 0 and MAX(SET)

Types SHORTINT, INTEGER, and LONGINT are *integer types*; types REAL and LONGREAL are *real types*; together they are called *numeric types*. They form a hierarchy: each larger type *includes* (the values of) the smaller types:

LONGREAL  $\supseteq$  REAL  $\supseteq$  LONGINT  $\supseteq$  INTEGER  $\supseteq$  SHORTINT

### A.6.2 Array Types

An array is a structure consisting of a number of elements that 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 type of the form

ARRAY L0, L1, ..., Ln OF T

is understood as an abbreviation of

ARRAY L0 OF  
ARRAY L1 OF  
...  
ARRAY Ln OF T

Arrays declared without length are called *open arrays*. They are restricted to pointer base types (see A.6.4), element types of open array types, and formal parameter types (see A.10.1). Examples:

```
ARRAY 10, N OF INTEGER
ARRAY OF CHAR
```

### A.6.3 Record Types

A record type is a structure consisting of a fixed number of elements, called *fields*, with possibly different types. The record type declaration specifies the name and type of each field. The scope of the field identifiers extends from the point of their declaration to the end of the record type, but they are also visible within designators referring to fields of record variables (see A.8.1). 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 elements are called *private fields*.

```
RecordType = RECORD ["(" BaseType ")" ] FieldList { ":" FieldList } END.
BaseType   = QualIdent.
FieldList   = [ IdentList ":" Type ].
```

Record types are extensible; i.e., a record type can be declared as an extension of another record type. In the example

```
T0 = RECORD x: INTEGER END
T1 = RECORD (T0) y: REAL END
```

*T1* is a (direct) *extension* of *T0* and *T0* is the (direct) *base type* of *T1* (see A.12.1). An extended type *T1* consists of the fields of its base type and of the fields that are declared in *T1* (see A.6). Identifiers declared in the extension must be different from the identifiers declared in its base type(s). The following are examples of record type declarations:

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

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

## A.6.4 Pointer Types

Variables of a pointer type  $P$  assume as values pointers to variables of some type  $T$ .  $T$  is called the *pointer base type* of  $P$  and must be a record or array type. Pointer types adopt the extension relation of their pointer base types: if a type  $T_1$  is an extension of  $T$ , and  $P_1$  is of type `POINTER TO T1`, then  $P_1$  is also an extension of  $P$ .

`PointerType = POINTER TO Type.`

If  $p$  is a variable of type  $P = \text{POINTER TO } T$ , a call of the predeclared procedure `NEW(p)` (see A.10.3) allocates a nameless variable of type  $T$  in free storage. If  $T$  is a record type or an array type with fixed length, the allocation has to be done with `NEW(p)`; if  $T$  is an  $n$ -dimensional open array the allocation has to be done with `NEW(p, e0, ..., en-1)`, where  $T$  is allocated with lengths given by the expressions  $e_0, \dots, e_{n-1}$ . In either case a pointer to the allocated variable is assigned to  $p$ .  $p$  is of type  $P$  and the *referenced* variable  $p^\wedge$  (pronounced as *p-referenced*) is of type  $T$ .

Any pointer variable may assume the value `NIL`, which points to no variable at all. All pointer variables are initialized to `NIL`.

## A.6.5 Procedure types

Variables of a procedure type  $T$  have a procedure (or `NIL`) as their value. If a procedure  $P$  is assigned to a variable of type  $T$ , the formal parameter lists (see A.10.1) of  $P$  and  $T$  must *match* (see A.12.1).  $P$  must not be a predeclared or type-bound procedure, nor may it be local to another procedure.

`ProcedureType = PROCEDURE [FormalParameters].`

## A.7 Variable Declarations

Variable declarations introduce variables by defining an identifier and a data type for them.

`VariableDeclaration = IdentList ":" Type.`

Record and pointer variables have both a *static type* (the type with which they are declared—simply called their type) and a *dynamic type* (the type they assume at run time). For pointers and variable

parameters of record type, the dynamic type may be an extension of their static type. The static type determines which fields of a record are accessible. The dynamic type is used to call type-bound procedures (see A.10.2).

The following are examples of variable declarations (refer to examples in A.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
    name: ARRAY 32 OF CHAR;
    count: INTEGER
END
t, c: Tree
```

## A.8 Expressions

Expressions denote rules of computation whereby constants and current values of variables are combined to compute 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.

### A.8.1 Operands

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

Designator	= Qualident {".." ident   "[" ExpressionList "]"   "^"   "(" Qualident ")"}
ExpressionList	= Expression {"," Expression}

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  $a[e_0, e_1, \dots, e_n]$  stands for

$a[e_0][e_1]\dots[e_n]$ . If  $r$  designates a record, then  $r.f$  denotes the field  $f$  of  $r$  or the procedure  $f$  bound to the dynamic type of  $r$  (see A.10.2). If  $p$  designates a pointer,  $p^$  denotes the variable that is referenced by  $p$ . The designators  $p^f$  and  $p^*[e]$  may be abbreviated as  $p.f$  and  $p[e]$ ; i.e., record and array selectors imply dereferencing. If  $a$  or  $r$  are read-only, then  $a[e]$  and  $r.f$  are also read-only.

A *type guard*  $v(T)$  asserts that the dynamic type of  $v$  is  $T$  (or an extension of  $T$ ); i.e., program execution is aborted if the dynamic type of  $v$  is not  $T$  (or an extension of  $T$ ). Within the designator,  $v$  is then regarded as having the static type  $T$ . The guard is applicable if

1.  $v$  is a variable parameter of record type or  $v$  is a pointer, and if
2.  $T$  is an extension of the static type of  $v$ .

If the designated object is a constant or a variable, then the designator refers to its current value. If it is a procedure, the designator refers to that procedure unless it is followed by a (possibly empty) parameter list, in which case it implies an activation of that procedure and stands for the value resulting from its execution. The actual parameters must correspond to the formal parameters as in proper procedure calls (see A.10.1).

The following are examples of designators (refer to examples in A.7):

i	(INTEGER)
a[i]	(REAL)
w[3].name[i]	(CHAR)
t.left.right	(Tree)
t(CenterTree).subnode	(Tree)

## A.8.2 Operators

Four classes of operators with different precedences (binding strengths) are syntactically distinguished in expressions. The operator  $\sim$  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].
SimpleExpression	= ["+"   "-"] Term {AddOperator Term}.
Term	= Factor {MulOperator Factor}.
Factor	= Designator [ActualParameters]   number   character   string   NIL   Set   "(" Expression ")"   "~" Factor.

Set	= "[" [Element {"", " Element}] "]".
Element	= Expression [".." Expression].
ActualParameters	= "(" [ExpressionList] ")".
Relation	= "="   "#"   "<"   "<="   ">"   ">="   IN   IS.
AddOperator	= "+"   "-"   OR.
MulOperator	= "*"   "/"   DIV   MOD   "&".

The available operators are listed in the following tables. Some operators are applicable to operands of various types, denoting different operations. In these cases, the actual operation is identified by the type of the operands. The operands must be *expression compatible* with respect to the operator (see A.12.1).

### Logical operators

OR	logical disjunction	$p \text{ OR } q \equiv \text{"if } p \text{ then TRUE, else } q \text{ end"}$
&	logical conjunction	$p \& q \equiv \text{"if } p \text{ then } q, \text{ else FALSE end"}$
~	negation	$\sim p \equiv \text{"not } p"$

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

### Arithmetic operators

+	sum
-	difference
*	product
/	real quotient
DIV	integer quotient
MOD	modulus

The operators +, -, \*, and / apply to operands of numeric types. The type of the result is the type of that operand that includes the type of the other operand, except for division (/), where the result is the smallest real type that includes both operand types. When used as monadic operators, - 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 positive divisor  $y$ :

$$\begin{aligned}x &= (x \text{ DIV } y) * y + (x \text{ MOD } y) \\0 &\leq (x \text{ MOD } y) < y\end{aligned}$$

Examples:

x	y	x DIV y	x MOD y
5	3	1	2
-5	3	-2	1

## Set operators

+	union
-	difference ( $x - y = x * (-y)$ )
*	intersection
/	symmetric set difference ( $x / y = (x - y) + (y - x)$ )

Set operators apply to operands of type SET and yield a result of type SET. The monadic minus sign denotes the complement of  $x$ ; i.e.,  $-x$  denotes the set of integers between 0 and MAX(SET) that are not elements of  $x$ .

A set constructor defines the value of a set by listing its elements between braces. The elements must be integers in the range 0..MAX(SET). A range  $a..b$  denotes all integers in the interval  $[a, b]$ .

## Relational operators

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

Relations yield a BOOLEAN result. The relations =, #, <, <=, >, and >= apply to numeric types, CHAR, (open) character arrays, and strings. The relations = and # also apply to BOOLEAN and SET, as well as to pointer and procedure types (including the value NIL).  $x$  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 "the dynamic type of  $v$  is  $T$  (or an extension of  $T$ )" and is called a *type test*. It is applicable if

1.  $v$  is a variable parameter of record type or  $v$  is a pointer, and if
2.  $T$  is an extension of the static type of  $v$ .

The following are examples of expressions (refer to examples in A.7):

1991	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

$(0 \leq i) \& (i < 100)$	BOOLEAN
$t.key = 0$	BOOLEAN
$k \in \{..j-1\}$	BOOLEAN
$w[i].name \leq "John"$	BOOLEAN
$t \text{ IS CenterTree}$	BOOLEAN

## A.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, the return, and the exit statement. 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 | LoopStatement |
  WithStatement | EXIT | RETURN [Expression] ].
```

### A.9.1 Assignments

Assignments replace the current value of a variable with a new value specified by an expression. The expression must be *assignment compatible* with the variable (see A.12.1). The assignment operator is written as " $:=$ " and pronounced as *becomes*.

Assignment = Designator " $:=$ " Expression.

If an expression  $e$  of type  $T_e$  is assigned to a variable  $v$  of type  $T_v$ , the following happens:

1. If  $T_v$  and  $T_e$  are record types, only those fields of  $T_e$  are assigned which also belong to  $T_v$  (*projection*); the dynamic type of  $v$  must be the same as the static type of  $v$  and is not changed by the assignment.
2. If  $T_v$  and  $T_e$  are pointer types, the dynamic type of  $v$  becomes the dynamic type of  $e$ .
3. If  $T_v$  is ARRAY  $n$  OF CHAR and  $e$  is a string of length  $m < n$ ,  $v[i]$  becomes  $e_i$  for  $i = 0..m-1$  and  $v[m]$  becomes 0X.

The following are examples of assignments (refer to examples in A.7):

```
i := 0
p := i = j
x := i + 1
k := log2(i+j)
F := log2                               (* see A.10.1 *)
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y) * (x-y)
t.key := i
w[i+1].name := "John"
t := c
```

## A.9.2 Procedure Calls

A procedure call activates a procedure. It may contain a list of actual parameters which replace the corresponding formal parameters defined in the procedure declaration (see A.10). The correspondence is established by the positions of the parameters in the actual and formal parameter lists. There are two kinds of parameters: *variable* and *value parameters*.

If a formal parameter is a variable parameter, the corresponding actual parameter must be a designator denoting a variable. If it denotes an element of a structured variable, the component selectors are evaluated when the formal/actual parameter substitution takes place, i.e., before the execution of the procedure. If a formal parameter is a value parameter, the corresponding actual parameter must be an expression. The value of this expression is assigned to the formal parameter (see also A.10.1).

ProcedureCall = Designator [ActualParameters].

Examples:

```
WriteInt(i*2+1)      (* see A.10.1 *)
t.Insert("John")      (* see A.11 *)
INC(w[k].count)
```

## A.9.3 Statement Sequences

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

StatementSequence = Statement {";" Statement}.

## A.9.4 If Statements

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

If statements specify the conditional execution of guarded statement sequences. The boolean expression preceding a statement sequence 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 = ' ') OR (ch = '') THEN ReadString
ELSE SpecialCharacter
END
```

## A.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 that statement sequence is executed whose case label list contains the obtained value. The case expression must either be of an integer type that *includes* the types of all case labels, or both the case expression and the case labels must be of type CHAR. Case labels are constants, and no value may occur more than once. If the value of the expression does not match any label, the statement sequence following the symbol ELSE is selected, if there is one; otherwise the program is aborted.

```
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
    | "'", '"': ReadString
ELSE SpecialCharacter
END
```

## A.9.6 While Statements

While statements specify the repeated execution of a statement sequence while the boolean expression (its *guard*) yields TRUE. The guard is checked before every execution of the statement sequence.

`WhileStatement = WHILE Expression DO StatementSequence END.`

Examples:

```
WHILE i > 0 DO i := i DIV 2; k := k + 1 END
WHILE (t # NIL) & (t.key # i) DO t := t.left END
```

## A.9.7 Repeat Statements

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

`RepeatStatement = REPEAT StatementSequence UNTIL Expression.`

## A.9.8 For Statements

A for statement specifies the repeated execution of a statement sequence for a fixed 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 statement

`FOR v := low TO high BY step DO statements END`

is equivalent to

```
v := low; temp := high;
IF step > 0 THEN
  WHILE v <= temp DO statements; v := v + step END
ELSE
  WHILE v >= temp DO statements; v := v + step END
END
```

*low* must be *assignment compatible* with *v* (see A.12.1), *high* must be *expression compatible* (i.e., comparable) with *v*, and *step* must be a

nonzero constant expression of an integer type. If *step* is not specified, it is assumed to be 1. Examples:

```
FOR i := 0 TO 79 DO k := k + a[i] END  
FOR i := 79 TO 1 BY -1 DO a[i] := a[i-1] END
```

### A.9.9 Loop Statements

A loop statement specifies the repeated execution of a statement sequence. It is terminated upon execution of an exit statement within that sequence (see A.9.10).

LoopStatement = LOOP StatementSequence END.

Example:

```
LOOP  
  ReadInt(i);  
  IF i < 0 THEN EXIT END;  
  WriteInt(i)  
END
```

Loop statements are useful to express repetitions with several exit points or cases where the exit condition is in the middle of the repeated statement sequence.

### A.9.10 Return and Exit Statements

A return statement indicates the termination of a procedure. It is denoted by the symbol RETURN, followed by an expression if the procedure is a function procedure. The type of the expression must be *assignment compatible* (see A.12.1) with the result type specified in the procedure heading (see A.10).

Function procedures require the presence of a return statement indicating the result value. In proper procedures, a return statement is implied by the end of the procedure body. Any explicit return statement therefore appears as an additional (probably exceptional) termination point.

An exit statement is denoted by 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, associated with the loop statement that contains them.

### A.9.11 With Statements

With statements execute a statement sequence depending on the result of a type test and apply a type guard to every occurrence of the tested variable within this statement sequence.

```
WithStatement = WITH Guard DO StatementSequence
  {"!" Guard DO StatementSequence}
  [ELSE StatementSequence] END.
Guard = Qualident ":" Qualident.
```

If  $v$  is a variable parameter of record type or a pointer variable, and if it is of a static type  $T_0$ , the statement

```
WITH v: T1 DO S1 | v: T2 DO S2 ELSE S3 END
```

has the following meaning: if the dynamic type of  $v$  is  $T_1$ , then the statement sequence  $S_1$  is executed, where  $v$  is regarded as if it had the static type  $T_1$ ; else if the dynamic type of  $v$  is  $T_2$ , then  $S_2$  is executed, where  $v$  is regarded as if it had the static type  $T_2$ ; else  $S_3$  is executed.  $T_1$  and  $T_2$  must be extensions of  $T_0$ . If no type test is satisfied and if an else clause is missing, the program is aborted.

Example:

```
WITH t: CenterTree DO i := t.width; c := t.subnode END
```

## A.10 Procedure Declarations

A procedure declaration consists of a *procedure heading* and a *procedure body*. The heading specifies the procedure identifier and the *formal parameters*. For type-bound procedures it also specifies the *receiver* parameter. The body contains declarations and statements. The procedure identifier is repeated at the end of the procedure declaration.

There are two kinds of procedures: *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 of the expression. Proper procedures are activated by a procedure call. A procedure is a function procedure if its formal parameters specify a result type. The body of a function procedure must contain a return statement that defines its result.

All constants, variables, types, and procedures declared within a procedure body are *local* to the procedure. Since procedures may

be declared as local objects, too, procedure declarations may be nested. The call of a procedure within its declaration implies recursive activation.

Objects declared in the environment of the procedure are also visible in those parts of the procedure in which they are not concealed by locally declared objects with the same name.

```

ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading    = PROCEDURE [Receiver] IdentDef
                      [FormalParameters].
ProcedureBody       = DeclarationSequence
                      [BEGIN StatementSequence] END.
DeclarationSequence = {CONST {ConstantDeclaration ";"}
                      | TYPE {TypeDeclaration ";"}
                      | VAR {VariableDeclaration ";"} }
                      {ProcedureDeclaration ";"}
                      | ForwardDeclaration ";".
ForwardDeclaration  = PROCEDURE " ^ " [Receiver] IdentDef
                      [FormalParameters].

```

If a procedure declaration specifies a receiver parameter, the procedure is considered to be bound to a type (see A.10.2). A *forward declaration* serves to allow forward references to a procedure whose actual declaration appears later in the text. The formal parameter lists of the forward declaration and the actual declaration must *match* (see A.12.1).

## A.10.1 Formal Parameters

Formal parameters are identifiers declared in the formal parameter list of a procedure. They correspond to 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, *value* and *variable* parameters, indicated in the formal parameter list by the absence or presence of the keyword VAR. Value parameters are local variables to which the value of the corresponding actual parameter is assigned as an initial value. Variable parameters correspond to actual parameters that are variables, and they stand for these variables. The scope of a formal parameter extends from its declaration to the end of the procedure block in which it is declared. 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. The result type of a function procedure can be neither a record nor an array.

```
FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" Qualident].
FPSection       = [VAR] ident {";" ident} ":" Type.
```

Let  $T_f$  be the type of a formal parameter  $f$  (not an open array) and  $T_a$  the type of the corresponding actual parameter  $a$ . For variable parameters,  $T_a$  must be the *same* as  $T_f$ , or  $T_f$  must be a record type and  $T_a$  an extension of  $T_f$ . For value parameters,  $a$  must be *assignment compatible* with  $f$  (see A.12.1).

If  $T_f$  is an open array, then  $a$  must be *array compatible* with  $f$  (see A.12.1). The lengths of  $f$  are taken from  $a$ . The following are 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 <100000*)
  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 WriteString(s: ARRAY OF CHAR);
  VAR i: INTEGER;
BEGIN i := 0;
  WHILE (i < LEN(s)) & (s[i] # 0X) DO Write(s[i]); INC(i) END
END WriteString;

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
```

## A.10.2 Type-Bound Procedures

Globally declared procedures may be associated with a record type declared in the same module. The procedures are said to be *bound* to the record type. The binding is expressed by the type of the

*receiver* in the heading of a procedure declaration. The receiver may be either a variable parameter of record type  $T$  or a value parameter of type  $\text{POINTER TO } T$  (where  $T$  is a record type). The procedure is bound to the type  $T$  and is considered local to it.

```
ProcedureHeading = PROCEDURE [Receiver] IdentDef [FormalParameters].
Receiver        = "(" [VAR] ident ":" ident ")".
```

If a procedure  $P$  is bound to a type  $T_0$ , it is implicitly also bound to any type  $T_1$  that is an extension of  $T_0$ . However, a procedure  $P'$  (with the same name as  $P$ ) may be explicitly bound to  $T_1$ , in which case it overrides the binding of  $P$ .  $P'$  is considered a *redefinition* of  $P$  for  $T_1$ . The formal parameters of  $P$  and  $P'$  must *match* (see A.12.1). If  $P$  and  $T_1$  are exported (see A. 4),  $P'$  must be exported, too.

If  $v$  is a designator and  $P$  is a type-bound procedure, then  $v.P$  denotes that procedure  $P$  that is bound to the dynamic type of  $v$  (*dynamic binding*). Note that this may be a different procedure than the one bound to the static type of  $v$ .  $v$  is passed to  $P$ 's receiver according to the parameter passing rules specified in A.10.1.

If  $r$  is a receiver parameter declared with type  $T$ ,  $r.P^$  denotes the (redefined) procedure  $P$  bound to the base type of  $T$ .

In a forward declaration of a type-bound procedure, the receiver parameter must be of the *same* type as in the actual procedure declaration. The formal parameter lists of both declarations must *match* (A.12.1).

Examples:

```
PROCEDURE (t: Tree) Insert (node: Tree);
  VAR p, father: Tree;
  BEGIN
    p := t;
    REPEAT father := p;
      IF node.key = p.key THEN RETURN END;
      IF node.key < p.key THEN p := p.left ELSE p := p.right END
    UNTIL p = NIL;
    IF node.key < father.key THEN father.left := node
    ELSE father.right := node
    END;
    node.left := NIL; node.right := NIL
  END Insert;
```

```
PROCEDURE (t: CenterTree) Insert (node: Tree); (* redefinition*)
BEGIN
  WriteInt(node(CenterTree).width);
  t.Insert^ (node) (* calls the Insert procedure bound to Tree *)
END Insert;
```

### A.10.3 Predeclared Procedures

The following table lists the predeclared 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
ASH( $x, n$ )	$x, n$ : integer type	LONGINT	arithmetic shift ( $x * 2^n$ )
CAP( $x$ )	CHAR	CHAR	$x$ is letter: corresponding capital letter
CHR( $x$ )	integer type	CHAR	character with ordinal number $x$
ENTIER( $x$ )	real type	LONGINT	largest integer not greater than $x$
LEN( $v, n$ )	$v$ : array; $n$ : integer constant	LONGINT	length of $v$ in dimension $n$ (first dim. = 0)
LEN( $v$ )	$v$ : array	LONGINT	equivalent to LEN( $v, 0$ )
LONG( $x$ )	SHORTINT INTEGER REAL	INTEGER LONGINT LONGREAL	identity identity identity
MAX( $T$ )	$T$ = basic type $T$ = SET	$T$ INTEGER	maximum value of type $T$ maximum element of a set
MIN( $T$ )	$T$ = basic type $T$ = SET	$T$ INTEGER	minimum value of type $T$ 0
ODD( $x$ )	integer type	BOOLEAN	$x \text{ MOD } 2 = 1$
ORD( $x$ )	CHAR	INTEGER	ordinal number of $x$
SHORT( $x$ )	LONGINT INTEGER LONGREAL	INTEGER SHORTINT REAL	identity identity identity (truncation possible)
SIZE( $T$ )	any type	integer type	number of bytes required by $T$

#### Proper procedures

Name	Argument types	Function
COPY( $x, v$ )	$x$ : char. array, string; $v$ : char. array	$v := x$
DEC( $v$ )	integer type	$v := v - 1$
DEC( $v, n$ )	$v, n$ : integer type	$v := v - n$
EXCL( $v, x$ )	$v$ : SET; $x$ : integer type	$v := v - \{x\}$
HALT( $x$ )	integer constant	terminate program
INC( $v$ )	integer type	$v := v + 1$
INC( $v, n$ )	$v, n$ : integer type	$v := v + n$
INCL( $v, x$ )	$v$ : SET; $x$ : integer type	$v := v + \{x\}$
NEW( $v$ )	pointer to record or fixed array	allocate $v^{\wedge}$
NEW( $v, x_0, \dots, x_n$ )	$v$ : pointer to open array; $x_i$ : int. type	allocate $v^{\wedge}$ with lengths $x_0..x_n$

COPY allows the assignment between (open) character arrays with different types. If necessary, the source is shortened to the target length minus one. The target is always terminated by the character 0X. In HALT(*x*), the interpretation of *x* is left to the underlying system implementation.

## A.11 Modules

A module is a collection of declarations of constants, types, variables, and procedures, together with a sequence of statements for the purpose of assigning initial values to the variables. A module 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 names of the imported modules. If a module *A* is imported by a module *M* and *A* exports an identifier *x*, then *x* is referred to as *A.x* within *M*. If *A* is imported as *B := A*, the object *x* must be referenced as *B.x*. This allows short alias names in qualified identifiers. Identifiers that are to be exported (i.e., that are to be visible in client modules) must be marked by an export mark in their declaration (see A. 4).

The statement sequence following the symbol BEGIN is executed when the module is added to a system (loaded), which is done after the imported modules have been loaded. It follows that cyclic import of modules is illegal. Individual (parameterless and exported) procedures can be activated from the system, and these procedures serve as *commands* (see A.12.4).

```

MODULE Trees;

IMPORT Texts, Oberon;
(* exports: Tree, Node, Insert, Search, Write, Init *)
(* exports read-only: Node.name *)

TYPE
  Tree* = POINTER TO Node;
  Node* = RECORD
    name*: POINTER TO ARRAY OF CHAR;
    left, right: Tree
  END;

VAR w: Texts.Writer;

```

```

PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR);
  VAR p, father: Tree;
  BEGIN p := t;
    REPEAT father := p;
      IF name = p.name^ THEN RETURN END;
      IF name < p.name^ THEN p := p.left ELSE p := p.right END
    UNTIL p = NIL;
    NEW(p); p.left := NIL; p.right := NIL;
    NEW(p.name, LEN(name)+1); COPY(name, p.name^);
    IF name < father.name^ THEN father.left := p
    ELSE father.right := p
    END
  END Insert;

PROCEDURE (t: Tree) Search* (name: ARRAY OF CHAR): Tree;
  VAR p: Tree;
  BEGIN p := t;
    WHILE (p # NIL) & (name # p.name^) DO
      IF name < p.name^ THEN p := p.left ELSE p := p.right END
    END;
    RETURN p
  END Search;

PROCEDURE (t: Tree) Write*;
  BEGIN
    IF t.left # NIL THEN t.left.Write END;
    Texts.WriteString(w, t.name^); Texts.WriteLine(w);
    Texts.Append(Oberon.Log, w.buf);
    IF t.right # NIL THEN t.right.Write END
  END Write;

PROCEDURE Init* (VAR t: Tree);
  VAR t: Tree;
  BEGIN
    NEW(t.name, 1); t.name[0] := 0X; t.left := NIL; t.right := NIL
  END Init;

BEGIN Texts.OpenWriter(w)
END Trees.
```

## A.12 Appendices to the Language Definition

### A.12.1 Definition of Terms

**Integer types**      SHORTINT, INTEGER, LONGINT

**Real types**      REAL, LONGREAL

**Numeric types**      *integer types, real types*

#### Same types

Two variables  $a$  and  $b$  with types  $T_a$  and  $T_b$  are of the *same type* if

1.  $T_a$  and  $T_b$  are both denoted by the same type identifier, or
2.  $T_a$  is declared to equal  $T_b$  in a type declaration of the form  $T_a = T_b$ , or
3.  $a$  and  $b$  appear in the same identifier list in a variable, record field, or formal parameter declaration and are not open arrays.

#### Equal types

Two types  $T_a$  and  $T_b$  are *equal* if

1.  $T_a$  and  $T_b$  are the *same type*, or
2.  $T_a$  and  $T_b$  are open array types with *equal* element types, or
3.  $T_a$  and  $T_b$  are procedure types whose formal parameter lists *match*.

#### Type inclusion

Numeric types *include* (the values of) smaller numeric types according to the following hierarchy:

LONGREAL  $\supseteq$  REAL  $\supseteq$  LONGINT  $\supseteq$  INTEGER  $\supseteq$  SHORTINT

#### Type extension (base type)

Given a type declaration  $T_b = RECORD (T_a) \dots END$ ,  $T_b$  is a *direct extension* of  $T_a$ , and  $T_a$  is a *direct base type* of  $T_b$ . A type  $T_b$  is an *extension* of a type  $T_a$  ( $T_a$  is a *base type* of  $T_b$ ) if

1.  $T_a$  and  $T_b$  are the *same types*, or
2.  $T_b$  is a direct extension of an extension of  $T_a$

If  $P_a = POINTER TO T_a$  and  $P_b = POINTER TO T_b$ ,  $P_b$  is an *extension* of  $P_a$  ( $P_a$  is a *base type* of  $P_b$ ) if  $T_b$  is an *extension* of  $T_a$ .

### Assignment compatibility

An expression  $e$  of type  $T_e$  is *assignment compatible* with a variable  $v$  of type  $T_v$  if one of the following conditions holds:

1.  $T_e$  and  $T_v$  are the *same type*.
2.  $T_e$  and  $T_v$  are numeric types and  $T_v$  includes  $T_e$ .
3.  $T_e$  and  $T_v$  are record types and  $T_e$  is an extension of  $T_v$  and the dynamic type of  $v$  is  $T_v$ .
4.  $T_e$  and  $T_v$  are pointer types and  $T_e$  is an extension of  $T_v$ .
5.  $T_v$  is a pointer or a procedure type and  $e$  is NIL.
6.  $T_v$  is ARRAY n OF CHAR,  $e$  is a string constant with  $m$  characters, and  $m < n$ .
7.  $T_v$  is a procedure type and  $e$  is the name of a procedure whose formal parameters *match* those of  $T_v$ .

### Expression compatibility

For a given operator, the types of its operands are expression compatible if they conform to the following table (which also shows the result type of the expression). Type  $T_1$  must be an extension of type  $T_0$ :

Operator	1 <sup>st</sup> Operand	2 <sup>nd</sup> Operand	Result Type
+ - *	numeric	numeric	smallest numeric type including both opd. types
/	numeric	numeric	smallest real type including both opd. types
DIV MOD	integer	integer	smallest integer type including both opd. types
+ - * /	SET	SET	SET
OR & ~	BOOLEAN	BOOLEAN	BOOLEAN
= # < <= > >=	numeric	numeric	BOOLEAN
	CHAR	CHAR	BOOLEAN
	character array, string	character array, string	BOOLEAN
= #	BOOLEAN	BOOLEAN	BOOLEAN
	SET	SET	BOOLEAN
	NIL, pointer type $T_0$ or $T_1$	NIL, pointer type $T_0$ or $T_1$	BOOLEAN
	NIL, procedure type $T$	NIL, procedure type $T$	BOOLEAN
IN	integer	SET	BOOLEAN
IS	type $T_0$	type $T_1$	BOOLEAN

### Array compatibility

An actual parameter  $a$  of type  $T_a$  is *array compatible* with a formal parameter  $f$  of type  $T_f$  if

1.  $T_f$  and  $T_a$  are the *same type*, or

2.  $T_f$  is an open array,  $T_a$  is any array, and their element types are *array compatible*, or
3.  $T_f$  is ARRAY OF CHAR and  $a$  is a string.

### Matching formal parameter lists

Two formal parameter lists *match* if

1. they have the same number of parameters, and
2. they have either the *same* function result type or none, and
3. parameters at corresponding positions have *equal* types, and
4. parameters at corresponding positions are both either value or variable parameters.

## A.12.2 Syntax of Oberon-2

```

Module      = MODULE ident ";" [ImportList] DeclSeq
            [BEGIN StatSeq] END ident ";".
ImportList   = IMPORT [ident ":="] ident {"." [ident ":="] ident} ";".
DeclSeq     = { CONST {IdentDef "=" ConstExpr ";"}
              | TYPE {IdentDef "=" Type ";"}
              | VAR {IdentList ":" Type ";"}
              | {ProcDecl ";" | ForwardDecl;"}.
ProcDecl    = PROCEDURE [Receiver] IdentDef [FormalPars] ";" DeclSeq
            [BEGIN StatSeq] END ident.
ForwardDecl = PROCEDURE "^" [Receiver] IdentDef [FormalPars].
FormalPars   = "(" [FPSection {"." FPSection}] ")" [":" Qualident].
FPSection   = [VAR] ident {"." ident} ":" Type.
Receiver    = "(" [VAR] ident ":" ident ")".
Type        = Qualident
              | ARRAY [ConstExpr {"." ConstExpr}] OF Type
              | RECORD ["(" Qualident ")""] FieldList {"." FieldList} END
              | POINTER TO Type
              | PROCEDURE [FormalPars].
FieldList   = [IdentList ":" Type].
StatSeq     = Statement {"." Statement}.
Statement   = [ Designator ":=" Expr
              | Designator ["(" [ExprList] ")"]
              | IF Expr THEN StatSeq {ELSIF Expr THEN StatSeq}
                [ELSE StatSeq] END
              | CASE Expr OF Case {"|" Case} [ELSE StatSeq] END
              | WHILE Expr DO StatSeq END
              | REPEAT StatSeq UNTIL Expr
              | FOR ident ":"= Expr TO Expr [BY ConstExpr] DO StatSeq END
              | LOOP StatSeq END
              | WITH Guard DO StatSeq {"|" Guard DO StatSeq}
                [ELSE StatSeq] END
              | EXIT
              | RETURN [Expr] ].
Case        = [CaseLabels {"." CaseLabels} ":" StatSeq].
CaseLabels  = ConstExpr [".." ConstExpr].
Guard       = Qualident ":" Qualident.
ConstExpr   = Expr.
Expr        = SimpleExpr [Relation SimpleExpr].
SimpleExpr  = ["+" | "-"] Term {AddOp Term}.
Term        = Factor {MulOp Factor}.
Factor      = Designator ["(" [ExprList] ")"] | number | character
              | string | NIL | Set | "(" Expr ")" | " ~ " Factor.
Set         = {" [Element {"." Element}] }".
Element    = Expr [".." Expr].
Relation   = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
AddOp      = "+" | "-" | OR.
MulOp      = "*" | "/" | DIV | MOD | "&".
Designator = Qualident {"." ident | "[" ExprList "]" | " ^ " | "(" Qualident ")" }.
ExprList   = Expr {"." Expr}.
IdentList   = IdentDef {"." IdentDef}.
Qualident  = [ident "."] ident.
IdentDef   = ident [" * " | "-"].

```

### A.12.3 The Module SYSTEM

The module SYSTEM contains certain types and procedures that are necessary to implement *low-level* operations particular to a given computer and/or operating system. These include, for example, facilities for accessing devices that are controlled by the computer, and facilities to break the type compatibility rules otherwise imposed by the language definition.

It is strongly recommended that the use of the module SYSTEM be restricted to specific modules (called *low-level* modules). Such modules are inherently nonportable and unsafe, but easily recognized due to the identifier SYSTEM appearing in their import list. The following specifications hold for the implementation of Oberon-2 on the Ceres computer.

Module SYSTEM exports a type BYTE with the following characteristics: Variables of type CHAR or SHORTINT can be assigned to variables of type BYTE. If a formal variable parameter is of type ARRAY OF BYTE, then the corresponding actual parameter may be of any type.

Another type exported by module SYSTEM is the type PTR. Variables of any pointer type may be assigned to variables of type PTR. If a formal variable parameter is of type PTR, the corresponding actual parameter may be any pointer type. If the actual parameter is a pointer to a record type  $T$  the address of the type descriptor of  $T$  is passed as the actual parameter.

The procedures contained in module SYSTEM are listed in the following tables. Most of them 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( $a, n$ )	$a: \text{LONGINT}; n: \text{integer}$	BOOLEAN	bit $n$ of $\text{Mem}[a]$
CC( $n$ )	$integer$ constant	BOOLEAN	condition $n$ ( $0 \leq n \leq 15$ )
LSH( $x, n$ )	$x: \text{integer}, \text{CHAR}, \text{BYTE}; n: \text{integer}$	type of $x$	logical shift
ROT( $x, n$ )	$x: \text{integer}, \text{CHAR}, \text{BYTE}; n: \text{integer}$	type of $x$	rotation
VAL( $T, x$ )	$T, x: \text{any type}$	$T$	$x$ interpreted as of type $T$

## Proper procedures

Name	Argument types	Function
GET( $a, v$ )	$a$ : LONGINT; $v$ : any basic type, pointer, procedure type	$v := M[a]$
PUT( $a, x$ )	$a$ : LONGINT; $x$ : any basic type, pointer, procedure type	$M[a] := x$
GETREG( $n, v$ )	$n$ : integer constant; $v$ : any basic type, pointer, procedure type	$v := \text{Register}_n$
PUTREG( $n, x$ )	$n$ : integer constant; $x$ : any basic type, pointer, procedure type	$\text{Register}_n := v$
MOVE( $a_0, a_1, n$ )	$a_0, a_1$ : LONGINT; $n$ : integer	$M[a_1..a_0 + n - 1] := M[a_0..a_0 + n - 1]$
NEW( $v, n$ )	$v$ : any pointer; $n$ : integer	allocate $n$ bytes of memory; assign its address to $v$

### A.12.4 The Oberon Environment

Oberon-2 programs usually run in an environment that provides *command activation*, *garbage collection*, *dynamic loading* of modules, and certain *run-time data structures*. Although not part of the language, this environment contributes to the power of Oberon-2 and is to some degree implied by the language definition. This section describes the essential features of a typical Oberon environment and provides implementation hints. More details can be found in [WiG92], [Rei91], and [PHT91].

#### Commands

A command is any parameterless procedure  $P$  that is exported from a module  $M$ . It is denoted by  $M.P$  and can be activated under this name from the shell of the operating system. In Oberon, a user invokes commands instead of programs or modules. This gives the user a finer grain of control and allows modules with multiple entry points. When a command  $M.P$  is invoked, the module  $M$  is dynamically loaded unless it is already in memory and the procedure  $P$  is executed. When  $P$  terminates,  $M$  remains loaded. All global variables and data structures that can be reached from global pointer variables in  $M$  retain their values. When  $P$  (or another command of  $M$ ) is invoked again, it may continue to use these values.

The following module demonstrates the use of commands. It implements an abstract data structure *Counter* that encapsulates a counter variable and provides commands to increment and print its value.

```

MODULE Counter;
IMPORT Texts, Oberon;

VAR
  counter: LONGINT;
  w: Texts.Writer;

PROCEDURE Add*; (*takes a numeric argument from the command line*)
  VAR s: Texts.Scanner;
BEGIN
  Texts.OpenScanner(s, Oberon.Par.text, Oberon.Par.pos);
  Texts.Scan(s);
  IF s.class = Texts.Int THEN INC(counter, s.i) END
END Add;

PROCEDURE Write*;
BEGIN
  Texts.WriteLine(w, counter, 5); Texts.WriteLine(w);
  Texts.Append(Oberon.Log, w.buf)
END Write;

BEGIN counter := 0; Texts.OpenWriter(w)
END Counter.

```

The user may execute the following two commands:

<i>Counter.Add n</i>	adds value <i>n</i> to variable <i>counter</i>
<i>Counter.Write</i>	writes current value of <i>counter</i> to screen

Since commands are parameterless, they have to get their arguments from the operating system. In general, commands are free to take arguments from anywhere (e.g., from the text following the command, from the most recent selection, or from a marked viewer). The command *Add* uses a scanner (a data type provided by the Oberon system) to read the value that follows it on the command line.

When *Counter.Add* is invoked for the first time, the module *Counter* is loaded and its body is executed. Every call of *Counter.Add n* increments the variable *counter* by *n*. Every call of *Counter.Write* writes the current value of *counter* to the screen.

Since a module remains loaded after the execution of its commands, there must be an explicit way to unload it (e.g., when

the user wants to substitute a recompiled version for the loaded version.) The Oberon system provides a command to do that.

### Dynamic Loading of Modules

A loaded module may invoke a command of a still unloaded module by calling the loader and passing the name of the desired command as a parameter. The specified module is then dynamically loaded and the designated command is executed. Dynamic loading allows the user to start a program as a small set of basic modules and to extend it by adding further modules at run time as the need becomes evident.

A module  $M_0$  may cause the dynamic loading of a module  $M_1$  without importing it.  $M_1$  may of course import and use  $M_0$ , but  $M_0$  need not know about the existence of  $M_1$ .  $M_1$  can be a module that is designed and implemented long after  $M_0$ .

### Garbage Collection

In Oberon-2, the predeclared procedure NEW is used to allocate data blocks in free memory. There is, however, no way to explicitly dispose of an allocated block. Rather, the Oberon environment uses a *garbage collector* to find the blocks that are not referenced by a pointer any more and to make them available for allocation again.

A garbage collector frees a programmer from the nontrivial task of deallocating data structures correctly and thus helps to avoid errors. However, it requires information about dynamic data at run time.

### Browser

The interface of a module (the declaration of the exported objects) is extracted from the module by a *browser*, which is a separate tool of the Oberon environment. For example, the browser produces the following interface of the module *Trees* from A.11.

```
DEFINITION Trees;
TYPE
  Tree = POINTER TO Node;
  Node = RECORD
    name: POINTER TO ARRAY OF CHAR;
    PROCEDURE (t: Tree) Insert (name: ARRAY OF CHAR);
    PROCEDURE (t: Tree) Search (name: ARRAY OF CHAR): Tree;
    PROCEDURE (t: Tree) Write;
  END;
```

```

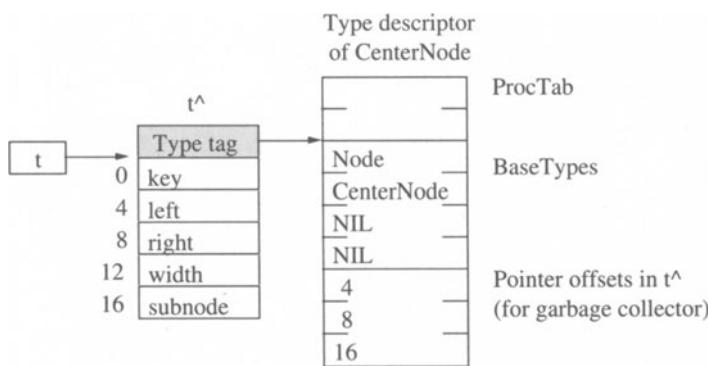
PROCEDURE Init (VAR t: Tree);
END Trees.
```

For a record type, the browser also collects all procedures bound to this type and shows their declaration in the record type declaration.

### Run-Time Data Structures

Certain information about records has to be available at run time: The dynamic type of records is needed for type tests and type guards. A table with the addresses of the procedures bound to a record is needed for calling them using dynamic binding. Finally, the garbage collector needs information about the locations of pointers in dynamically allocated records. All that information is stored in *type descriptors*, of which there is one for every record type at run time. The following paragraphs show a possible implementation of type descriptors.

The dynamic type of a record corresponds to the address of its type descriptor. For dynamically allocated records, this address is stored in a *type tag*, which precedes the actual record data and is invisible to the programmer. If *t* is a variable of type *CenterTree* (see the example in A.6), Figure A.12.1 shows one possible implementation of the run-time data structures.



**Fig. A.12.1** A variable *t* of type *CenterTree*, the record *t<sup>^</sup>* of type *CenterNode*, and its type descriptor

Since both the table of procedure addresses and the table of pointer offsets must have a fixed offset from the type descriptor address, and since both may grow when the type is extended and further procedures and pointers are added, the tables are located at the

opposite ends of the type descriptor and grow in different directions.

A type-bound procedure  $t.P$  is called as  $t.tag.ProcTab[Indexp]$ . The procedure table index of every type-bound procedure is known at compile time. A type test  $v$  IS  $T$  is translated into  $v.tag.BaseTypes[ExtensionLevel_T] = TypeDescrAdr_T$ . Both the extension level of a record type and the address of its type descriptor are known at compile time. For example, the extension level of *Node* is 0 (it has no base type), and the extension level of *CenterNode* is 1.

# B The Module OS

OS is a cover module for various constants, types, variables and procedures of the Oberon System that are used in examples throughout this book. It serves to keep the interface between the examples and the Oberon System small and permits us to avoid a description of the complete Oberon module library. (Interested readers are referred to [Rei91].)

DEFINITION OS; *Interface of OS*  
IMPORT Display, Files, Fonts; (\*Oberon modules that are not explained here\*)

CONST  
right = 0; middle = 1; left = 2; (\*mouse button codes\*)  
ticks = 300; (\*OS.Time returns the time in units of 1/ticks seconds\*)

TYPE  
File = Files.File;  
Font = Fonts.Font;  
Message = RECORD END; (\*base type for all message records\*)  
Object = POINTER TO ObjectDesc;  
Pattern = Display.Pattern;

Rider = RECORD (Files.Rider) (\*read/write position in a file\*)  
PROCEDURE (VAR r: Rider) Set (f: Files.File; pos: LONGINT);  
PROCEDURE (VAR r: Rider) Read (VAR x: CHAR);  
PROCEDURE (VAR r: Rider) ReadString (VAR s: ARRAY OF CHAR);  
PROCEDURE (VAR r: Rider) ReadInt (VAR x: INTEGER);  
PROCEDURE (VAR r: Rider) ReadLint (VAR x: LONGINT);  
PROCEDURE (VAR r: Rider) ReadObj (VAR x: Object);  
PROCEDURE (VAR r: Rider) ReadChars  
    (VAR x: ARRAY OF CHAR; n: LONGINT);  
PROCEDURE (VAR r: Rider) Write (x: CHAR);  
PROCEDURE (VAR r: Rider) WriteString (s: ARRAY OF CHAR);  
PROCEDURE (VAR r: Rider) WriteInt (x: INTEGER);  
PROCEDURE (VAR r: Rider) WriteLint (x: LONGINT);  
PROCEDURE (VAR r: Rider) WriteObj (x: Object);  
PROCEDURE (VAR r: Rider) WriteChars  
    (VAR x: ARRAY OF CHAR; n: LONGINT);  
END;  
ObjectDesc = RECORD *Object*

```

PROCEDURE (x: Object) Load (VAR r: Rider);
PROCEDURE (x: Object) Store (VAR r: Rider);
END;

VAR
  Caret: Pattern;
  screenH-, screenW-: INTEGER; (*screen height and width in pixels*)

```

*Screen operations*

```

PROCEDURE CopyBlock (sx, sy, w, h, dx, dy: INTEGER);
PROCEDURE FillBlock (x, y, w, h: INTEGER);
PROCEDURE EraseBlock (x, y, w, h: INTEGER);
PROCEDURE InvertBlock (x, y, w, h: INTEGER);
PROCEDURE DrawPattern (pat: Pattern; x, y: INTEGER);
PROCEDURE DrawCursor (x, y: INTEGER);
PROCEDURE FadeCursor;

```

*Font operations*

```

PROCEDURE DefaultFont (): Font;
PROCEDURE FontWithName (name: ARRAY OF CHAR): Font;
PROCEDURE GetCharMetric (f: Font; ch: CHAR;
  VAR dx, x, y, w, h: INTEGER; VAR pat: Pattern);

```

*Mouse and keyboard operations*

```

PROCEDURE AvailChars (): INTEGER;
PROCEDURE ReadKey (VAR ch: CHAR);
PROCEDURE GetMouse (VAR buttons: SET; VAR x, y: INTEGER);

```

*File operations*

```

PROCEDURE NewFile (name: ARRAY OF CHAR): File;
PROCEDURE OldFile (name: ARRAY OF CHAR): File;
PROCEDURE Register (f: File);
PROCEDURE InitRider (VAR r: Rider);

```

*Miscellaneous*

```

PROCEDURE NameToObj (name: ARRAY OF CHAR; VAR obj: Object);
PROCEDURE Move (VAR fromBuf: ARRAY OF CHAR; from: LONGINT;
  VAR toBuf: ARRAY OF CHAR; to, n: LONGINT);
PROCEDURE Time (): LONGINT;
PROCEDURE Call (command: ARRAY OF CHAR);

```

END OS.

*Screen operations*

*CopyBlock* (*sx, sy, w, h, dx, dh*)

copies the rectangular screen area (*sx, sy, w, h*) to (*dx, dy, w, h*).

*FillBlock* (*x, y, w, h*)

fills the rectangular screen area (*x, y, w, h*).

*EraseBlock* (*x, y, w, h*)

deletes the rectangular screen area (*x, y, w, h*).

*InvertBlock* (*x, y, w, h*)

inverts the rectangular screen area (*x, y, w, h*).

*DrawPattern* (*pat, x, y*)

copies the rectangular bit pattern *pat* to the screen position with left bottom corner (*x, y*).

*DrawCursor* ( $x, y$ )

moves the mouse pointer to position  $(x, y)$ .

*FadeCursor*

hides the mouse pointer.

$f := \text{DefaultFont}()$

*Font operations*

returns the standard font.

$f := \text{FontWithName}(n)$

returns the font with the name  $n$ .

*GetCharMetric* ( $fnt, ch, dx, x, y, w, h, pat$ )

returns the character metrics  $(x, y, w, h, dx)$  and the bit pattern  $pat$  of the character  $ch$  in font  $fnt$ . The meaning of the metrics is shown in Fig. 11.22.

$n := \text{AvailChars}()$

*Mouse and keyboard operations*

returns the number of characters in the keyboard buffer.

*ReadKey* ( $ch$ )

reads and removes the next character  $ch$  from the keyboard buffer. If the buffer is empty, the method stalls until a character is typed in.

*GetMouse* ( $b, x, y$ )

returns the mouse coordinates  $(x, y)$  relative to the lower left corner of the screen as well as the set  $b$  of pressed mouse buttons ( $0 = \text{right}$ ,  $1 = \text{middle}$ ,  $2 = \text{left}$ ).

$f := \text{NewFile}(n)$

*File operations*

creates a new (temporary) file  $f$  with name  $n$  and opens it.

$f := \text{OldFile}(n)$

opens an existing file  $f$  with name  $n$ . If no such file exists,  $f = \text{NIL}$ .

*Register* ( $f$ )

transforms the temporary file  $f$  created with *NewFile* into a permanent file.

*InitRider* ( $r$ )

initializes the rider  $r$  (see Section 8.3).

$r.\text{Set}(f, pos)$

*Methods of class Rider*

sets rider  $r$  to position  $pos$  in file  $f$ .

$r.\text{Read}(ch)$

reads character  $ch$  from rider  $r$ .

$r.\text{ReadInt}(x)$

reads integer  $x$  from rider  $r$ .

*r.ReadLInt (x)*  
     reads long integer *x* from rider *r*.  
*r.ReadString (s)*  
     reads string *s* (stored in compressed form) from rider *r*.  
*r.ReadChars (buf, len)*  
     reads *len* characters from rider *r* into buffer *buf*.  
*r.ReadObj (obj)*  
     creates and reads an arbitrary object written with *WriteObj* and  
     returns it in *obj* (see Section 8.3).  
*r.Write (ch)*  
     writes character *ch* to rider *r*.  
*r.WriteInt (x)*  
     writes integer *x* to rider *r*.  
*r.WriteLine (x)*  
     writes long integer *x* to rider *r*.  
*r.WriteString (s)*  
     writes string *s* in compressed form to rider *r*.  
*r.WriteChars (buf, len)*  
     writes *len* characters from buffer *buf* to rider *r*.  
*r.WriteObj (obj)*  
     writes an arbitrary object *obj* to rider *r* (see Section 8.3).

*Other operations**NameToObj (name, obj)*

The parameter *name* is a string of the form "M.T". *NameToObj* creates a record of type *T* exported by module *M* and returns a pointer to it in *obj*. If the module or type name is incorrect or the created object is not assignment compatible with *obj*, NIL is returned.

*Move (buf0, pos0, buf1, pos1, len)*

copies *len* bytes from *buf0[pos0]* to *buf1[pos1]*.

*t := Time ()*

returns the elapsed time since system start in  $1/ticks$  seconds (*ticks* being a constant declared in OS).

*Call (cmd)*

activates the command *cmd* and loads the module containing the command if it is not already loaded.

# C The Module *IO*

Module *IO* handles simple input/output of numbers, characters and strings. Input is handled via a scanner that is able to read various symbols in a text. Output is via procedures.

To read a text, a scanner *s* is set to the desired text position via *s.Set*. Successive symbols can be read via *s.Read*.

For output, a text *t* must be assigned to the variable *out* and its write position must be set via *t.SetPos*. Output routines write to the text *out* starting at position *out.pos*.

DEFINITION IO;  
IMPORT Texts0;

*Interface of IO*

CONST none = 0; integer = 1; name = 2; string = 3; char = 4;

TYPE

Scanner = RECORD  
text: Texts0.Text; (\*text to which scanner is set\*)  
class: INTEGER; (\*class of recognized symbol\*)  
int: LONGINT; (\*filled if class=integer\*)  
str: ARRAY 32 OF CHAR; (\*filled if class=string or name\*)  
ch: CHAR; (\*filled if class=char\*)  
PROCEDURE (VAR s: Scanner) Set (: Texts0.Text; pos: LONGINT);  
PROCEDURE (VAR s: Scanner) SetToParameters;  
PROCEDURE (VAR s: Scanner) Read;  
PROCEDURE (VAR s: Scanner) Eot (): BOOLEAN;  
PROCEDURE (VAR s: Scanner) Pos (): LONGINT;  
END ;

*Scanner*

VAR out: Texts0.Text; (\*output procedures write to this text\*)

PROCEDURE Ch (ch: CHAR);  
PROCEDURE Str (s: ARRAY OF CHAR);  
PROCEDURE Int (x: LONGINT; w: INTEGER);  
PROCEDURE Real (x: REAL; w: INTEGER);  
PROCEDURE NL;

*Output routines*

END IO.

<i>Scanner messages</i>	<i>s.Set(t, pos)</i> sets the scanner <i>s</i> to position <i>pos</i> in text <i>t</i> .
	<i>s.SetToParameters</i> sets the scanner <i>s</i> to the text after the last command clicked. This permits reading of command parameters.
	<i>s.Read</i> reads the next symbol from the current scanner position and returns its value in <i>s.int</i> , <i>s.str</i> or <i>s.ch</i> . <i>s.class</i> specifies the kind of symbol read. Blanks are skipped. Examples:

<i>Input</i>	<i>Kind of Symbol</i>	<i>Value in</i>
<eot>	<i>s.class</i> = none	—
123	<i>s.class</i> = integer	<i>s.int</i>
-123	<i>s.class</i> = integer	<i>s.int</i>
xxx	<i>s.class</i> = name	<i>s.str</i>
xxx.yyy	<i>s.class</i> = name	<i>s.str</i>
"xxx"	<i>s.class</i> = string	<i>s.str</i> (no quotes)
other	<i>s.class</i> = char	<i>s.ch</i>

<i>bool := s.Eof()</i>	returns TRUE if no more symbols can be read from the scanner <i>s</i> , otherwise FALSE.
<i>pos := s.Pos ()</i>	returns the current text position of the scanner <i>s</i> .

<i>Output routines</i>	<i>Str(s)</i> outputs the string <i>s</i> .
	<i>Ch(ch)</i> outputs the character <i>ch</i> .
	<i>Int(i, w)</i> outputs the signed integer <i>i</i> right-justified in a character field of width <i>w</i> .
	<i>Real(r, w)</i> outputs the real number <i>r</i> in a character field of width <i>w</i> (e.g., <i>Real(123.45, 7) = 0.12E02</i> ).
	<i>NL</i> causes a line feed.

# D How to Get Oberon

The Oberon System, including the Oberon-2 compiler and various tools such as a text editor, a graphics editor and a browser, is available at no cost. It can either be obtained via *ftp* from ETH Zurich or ordered from Springer-Verlag on diskette.

*Oberon*

The Oberon System is currently available for Sun SPARC-Station, DECstation, IBM RS/6000, Apple Macintosh II and IBM-PC (MS-DOS). Oberon-2 compilers are currently available for Sun SPARCStation, DECstation and IBM RS/6000.

*Platforms*

The parameters for the *ftp* program are:

*ftp*

Ftp host name:	neptune.inf.ethz.ch
Internet address:	129.132.101.33
Login name:	ftp
Password:	<your e-mail address>
Ftp directory:	Oberon

Oberon can also be purchased from Springer-Verlag on diskette. Please specify the computer version.

*Diskette*

In addition to this book on Oberon-2 and its application in object-oriented programming, the following books serve as a documentation of Oberon:

*Documentation*

- *N. Wirth and M. Reiser Programming in Oberon. Steps beyond Pascal and Modula-2.* Addison-Wesley 1992  
Tutorial for the Oberon programming language and concise language reference.
- *M. Reiser: The Oberon System. User Guide and Programmer's Manual.* Addison-Wesley, 1991  
User manual for the programming environment and reference for the standard module library.

- *N. Wirth and J. Gutknecht: Project Oberon. Addison-Wesley, 1992*  
Program listings with explanations for the whole Oberon System, including the compiler for the NS32000.

*Oberon0*

The source code for Oberon0 described in Chapter 11 can also be obtained at no cost so that the reader can play with it and extend it. The source code is available via the same ftp address as specified above. It is in the subdirectory Oberon0. If the Oberon System is purchased on diskette, the source code of Oberon0 is included.

# Bibliography

- [Abb83] Abbott R.: Program Design by Informal English Descriptions. *Communications of the ACM*, 26 (11), 1983
- [BDMN79] Birtwistle G.M., Dahl O.-J., Myhrhaug B., Nygaard K.: *Simula Begin*, Studentlitteratur, Lund, Sweden, 1979
- [BeC89] Beck K., Cunningham W.: A Laboratory for Teaching Object-Oriented Thinking. *Proceedings OOPSLA'89. SIGPLAN Notices*, 24 (10), 1989
- [Boo91] Booch G.: *Object-Oriented Design with Applications*. Benjamin Cummings, 1991
- [Bud91] Budd T.: *Object-Oriented Programming*. Addison-Wesley, 1991
- [Cha92] Chambers C.: The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages. Ph.D. thesis, Stanford, 1992
- [CoY90] Coad P., Yourdon E.: *Object-Oriented Analysis*. Prentice Hall, 1990
- [Deu89] Deutsch P.: Design Reuse and Frameworks in the Smalltalk-80 System. In Biggerstaff T.J., Perlis A.J. (ed.): *Software Reusability*, Volume 2, ACM Press, 1989
- [DoD83] Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A), United States Departement of Defense, Washington D.C., 1983

- [GWM88] Gamma E., Weinand A., Marty R.: ET++ – An Object-Oriented Application Framework in C++. Proceedings OOPSLA'88, SIGPLAN Notices, 23 (11), 1988
- [GoR83] Goldberg A., Robson D.: Smalltalk-80, The Language and its Implementation. Addison-Wesley, 1983
- [Hof90] Hoffman D.: On Criteria for Module Interfaces. IEEE Trans. on Software Engineering, 16 (5), 1990
- [JoF88] Johnson R.E., Foote B.: Designing Reusable Classes. Journal of Object-Oriented Programming, June/July 1988
- [KrP88] Krasner G., Pope S.: A Cookbook for Using the MVC User Interface Paradigm in Smalltalk. Journal of Object-Oriented Programming Aug./Sep. 1988
- [Mey86] Meyer B.: Genericity versus Inheritance. Proceedings OOPSLA'86, SIGPLAN Notices, 21 (11), 1986
- [Mey87] Meyer B.: Object-Oriented Software Construction. Prentice Hall, 1987
- [Par72] Parnas D.L.: On the Criteria to be Used in Decomposing Systems into Modules, Communications of the ACM, 15 (12), 1972
- [PHT91] Pfister C., Heeb B., Templ J.: Oberon Technical Notes. Computer Science Report 156, ETH Zürich, March 1991
- [RBP91] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: Object-Oriented Modeling and Design. Prentice Hall, 1991
- [Rei91] Reiser M.: The Oberon System; Users Guide and Programmers Manual. Addison-Wesley, 1991
- [ReW92] Reiser M., Wirth N.: Programming in Oberon. Steps Beyond Pascal and Modula-2. Addison-Wesley, 1992
- [Sch86] Schmucker K.J.: Object-Oriented Programming for the Macintosh. Hayden, 1986
- [Sed88] Sedgewick R.: Algorithms. Addison-Wesley, 1988
- [ShM88] Shlaer S., Mellor S.: Object-Oriented Systems Analysis: Modeling the World in Data. Yourdon Press, 1988

- [Str86] Stroustrup B.: The C++ Programming Language, Addison-Wesley, 1986 (second edition 1991)
- [Str89] Stroustrup B.: Multiple Inheritance for C++. Proceedings EUUG Spring Conference, Helsinki, May 1989
- [Swe85] Sweet R.E.: The Mesa Programming Environment. SIGPLAN Notices, 20 (7), 1985
- [Szy92] Szyperski C.A.: Write-ing Applications. Proceedings of Tools Europe 92, Dortmund, 1992
- [Web89] Webster B.F.: The NeXT Book. Addison-Wesley, 1989
- [WiG92] Wirth N., Gutknecht J.: Project Oberon. The Design of an Operating System and Compiler. Addison-Wesley, 1992
- [Wir71] Wirth N.: Program Development by Stepwise Refinement. Communications of the ACM, 14, (4), 1971
- [WiW89] Wirfs-Brock A., Wilkerson B.: Variables Limit Reusability. Journal of Object-Oriented Programming, May/June 1989
- [WWW90] Wirfs-Brock R., Wilkerson R., Wiener L.: Designing Object-Oriented Software. Addison-Wesley, 1990

# Index

- & 235
- 0X (terminal character) 247
- Abbott's method 131
- ABS 246
- abstract
  - classes **69, 135**
  - classes as design 71
  - data structure **36, 37**
  - data types 8, 10, 15, 39, 40, **79**
  - figures 88
  - methods 71, 222
  - output medium 91
- abstract data structures **36, 37**
  - advantages 38
- abstract data types 8, 10, 15, 39, 40, **79**
- abstraction 126, 219
  - mechanisms **13**
  - medium 29
- access 217, 220, 222, 223
- actual parameters 23, 243, 244, 251
- acyclic graph 113
- Ada 81
- adaptable components **93**
- Add 50
- ADR 254
- alias 247
- allocation 232
- application
  - domains 148
  - frameworks **153, 155**
  - independence 148
- application-specific tasks 148
- applications for OOP **79**
- arithmetic expressions 21
- arithmetic operators **235**
- array compatibility 244, **251**
- array types 19, **230, 232, 234**
- AsciiTexts 171, 173
- ASH 246
- assignment **237**
- assignment compatibility 57, 58, 237, 241, 244, **251**
- asterisk 26, 228
- atomicity 133, 134
- attr 179
- AttrDesc 179
- Attribute 177, 179
- attribute
  - list 177, 183, 184
  - node 184
  - segments 183, 184
- AvailChars 260, 261
- Backus-Naur Formalism 225
- BallItem 115
- BallProcess 115
- base class 53
- base types 9, 18, 53, 56, 57, **230, 231, 250**
- beg 179
- BEGIN 247
- behavior 220
  - at run time 91
  - common 69, 167
  - replaceable 29, 79, **91**
- benefits of OOP **219**
- BIT 254

block 228  
BOOLEAN 230, 235, 236  
boolean expressions 21, 239, 240  
bottom-up 126  
broadcast 76, 166  
browser 27, 222, 257  
BYTE 254  
  
C++ 223  
Call 198, 260, 262  
CAP 246  
CapTerminal 68  
caret 191  
case  
    analysis 6, 87  
    distinctions 89  
    labels 239  
    statement **239**  
case sensitivity 225  
case study 157  
casting text 185  
CC 254  
Ceres computer 254  
Ch 264  
ChangeFont 177, 182, 200, 201  
CHAR 230  
character constants 227  
character metrics 189  
CHR 246  
Circle 88  
class (see classes)  
    interface 132  
    hierarchy 145, 222  
    libraries 113, 221  
    multiple interfaces 85  
class relationships  
    has-a **136**, 143  
    is-a 137, 143  
    uses-a 137  
classes 10, 13, 15, 43, 219, 221  
    alternatives 140  
    and modules 47  
    identifying 127  
    physical/logical entities 128  
    relationships **136**  
    when to use 139  
  
Clear 26, 37, 40, 45, 49, 171, 174, 182  
clients 36, 38, 137  
clipping 197, 209  
Close 162, 165, 166  
cmdFrame 191  
cmdPos 191  
collaboration graphs 138  
commands **29**, 248, **255**, 256  
    activation 255  
    arguments 30, 256  
    purpose 31  
    use of 256  
comments 227  
common behavior 69, 167  
common interface 71  
commonalities 148  
compatibility 9, 57, 67, 108, 217, 232  
    assignment 57, 58, 237, 241, 244,  
        **251**  
    expression 235, 241, **251**  
    of base type and extension **56**  
compilation 28  
compilation units 29, 47  
compile time 259  
complex class libraries 113  
complexity 79, 139, 219  
components 219, 220, 224  
composite data types 19  
comprehension 219, 221  
compression of type names 107, 108  
computer game 114  
concentrating behavior 129  
concepts 221  
concrete classes 95  
concrete data structures **33**, 36  
    drawbacks 36  
concrete figures 88  
concurrency 168  
conflicting criteria 134  
consistency 133  
constant declaration **229**  
constant expression 229  
constants 229  
cont 163  
Contains 50  
contents frame **159**

- contract 136
- control 168
- control flow 150
- control variable 240
- Copy 163, 166, 178, 180, 188, 197, 203, 204, 206, 209, 213, 215, 247
- CopyBlock 260
- CopyTo 49
- costs of OOP **221**
- coupling 217
- CRC cards **131**
- creating objects 100
- CryptFile 73
- curShape 204
- cyclic import 247
- 
- data 11
- data abstraction **8, 33**, 140, 223
  - cost of 80
- data packages 76
- data structures
  - abstract 36
  - concrete 33
  - extensible 168
  - heap 33
  - heterogeneous **86**, 168
  - priority queue 33
- data type 18
  - abstract 39
- DEC 247
- decimal 226
- declaration **18, 25, 228**
- decomposing programs into
  - procedures 5
- decomposition 5, 6, 130
- DefaultFont 260, 261
- Defocus 161, 163, 188, 194
- Delete 171, 174, 181
- DeleteSelected 203, 204
- deleting 172
- delimiters 227
- dereferencing 234
- design **71, 98, 126, 158**
  - Abbott's method 131
  - abstract classes 136
- as language design 135
- class library 221
- class relationships 136
- considerations **128, 129**
- CRC cards **131**
- data abstraction 140
- data fields 129
- deriving classes **131**
- errors **141**
- finding classes 128
- for reusability 136
- functional **125**
- identifying classes 128
- identifying methods 128
- learning from others 130
- mistakes 129
- modeling 128
- object-oriented **125**
- patterns 98, 130
- subclasses 135
- verbal specification **131**
- when classes 139
- without classes 140
- design errors **141**
  - class hierarchy 145
  - confusing relationships 143
  - identical variants 144
  - superclass and subclass 144
  - trivial classes 142
  - wrong class 145
- designators 233, 234
- DESStream 110
- development time 220
- dialog programs 153, 154
- diamond structure 112
- Dictionaries 120
- dictionary 25, 26, 30, 120
- digit 226
- direct access 48
- directed acyclic graphs 113
- DiskFile 70, 72
- displaying text 185
- distinguishing object variants 68
- distributed responsibilities 11
- distributed state 11
- distribution of functionality 222

DIV 235, 251  
 documentation 222  
 Draw 161, 163, 164, 178, 179, 187,  
     194, 203, 204, 205, 208, 209, 212,  
     215, 216  
 Draw0 201, 210  
 DrawCursor 260, 261  
 drawing primitives 209  
 DrawLine 197  
 DrawPattern 260  
 dsc 179  
 dx 190  
 dynamic  
     allocation of memory 20  
     binding 8, 67, 221, 223, 245, 258  
     extensibility 153  
     loading 90, 102, 212, 223, 255, 257  
     type 59, 61, 69, 111, 233, 236, 237,  
         238, 242, 245, 251, 258  
  
 early binding 67  
 EBNF 225  
 Edit0 199, 200  
 editing text 185  
 editors 151  
 efficiency 48, 222  
 Eiffel 81, 83  
 Elem 120, 179  
 ElemDesc 179  
 Element 153, 176, 178, 179, 215  
 element type 230  
 elementary statements 22, 237  
 elements 152, 176, 177, 178, 199, 213,  
     217  
 ElemPos 177, 182  
 Ellipse 102  
 Ellipses 103  
 ELSE 239  
 embedding 64  
 embedding elements 213  
 empty statement 237  
 encapsulation 7, 38, 47, 139  
 EncryptionStream 110  
 end 179  
 Enter 26  
  
 ENTIER 246  
 Eof 264  
 equal types 250  
 EraseBlock 260  
 errors in code 220  
 event loop 154, 168  
 event-driven applications 154  
 Excl 49, 247  
 existing code 7  
 exit  
     condition 241  
     points 241  
     statement 241, 242  
 export 25, 231  
 export mark 26, 228, 247  
 expression compatibility 235, 241, 251  
 expressions 20, 233, 234, 237, 251  
     arithmetic 21  
     boolean 21  
     relational 21  
     set 22  
 extended type 8  
 extensibility 53, 63, 65, 89, 90, 96,  
     109, 147, 152, 184, 220, 224  
     dynamic 153  
     in multiple dimensions 109  
     run-time 153  
 extensible data structures 168  
 extensible viewer system 168  
 extension 9, 53, 56, 57, 97, 135, 140,  
     148, 156, 176, 184, 199, 231, 232,  
     233, 236, 242, 250, 251  
     approaches 64  
     at run time 101  
     by inheritance 65  
     embedding 64  
     level 259  
     modify duplicate 64  
     modify original 64  
     of concrete classes 95  
  
 factoring out common behavior 69  
 FadeCursor 260, 261  
 fields 231  
 Figure 50, 54, 55, 87, 88, 101, 105

FigureFrames 101  
figures 101, 202, 206  
    abstract 88  
    concrete 88  
File 133  
file operations 260, 261  
FillBlock 260  
finding classes 127  
finding objects 126  
flexibility 222  
fnt 179  
focus 163  
focus frame 162, 168  
font operations 260, 261  
fonts **176**, 184  
FontWithName 260, 261  
for statement **240**  
formal parameters 23, 232, 242, 243,  
    244, 251  
forward declaration 243, 245  
Frame 92, 96, 97, 101, 117, 118, 122,  
    159, 160, 162, 167, 170, 186, 190,  
    208, 214, 216  
frames 91, 118, 159, 161  
    classes 96  
    coordinates 160  
    layout 189  
    methods 163  
    metrics 188  
frameworks 85, **147**, 148  
    example 149, 150, 151, 220  
function designator 244  
function procedures 24, 241, 243, 244,  
    246, 254  
functional decomposition 125  
functional design **125**  
future modifications 130

gap 172  
garbage collection 20, 255, 257, 258  
generality 223  
generic 81  
    binary tree 81  
    components **81**, 220  
    procedures 246

stack 83  
generosity 81, 85, 106, 199  
GET 255  
GetBox 203, 204, 205, 213  
GetCharMetric 260, 261  
GetCharPos 198  
GetMetric 197  
GetMouse 260, 261  
GetPointPos 198  
GETREG 255  
GetSelection 197, 199  
global procedures 48  
global variables 48  
Graphic 202, 204, 208  
graphic elements 214, 216  
    in text **213**  
graphical notation 138  
graphical representation 55  
GraphicElems0 215, 217  
GraphicFrames0 **206**, 208  
graphics editor 86, 88, 108, **201**, 210  
graphics frame **206**, 214, 216, 217  
growing and shrinking texts 172  
guard 239, 240  
guarded statement sequences 239

h 163, 179  
HALT 247  
Handle 161, 163, 165, 188, 195, 198,  
    199, 209, 210  
HandleKey 161, 163, 188, 195  
HandleMouse 161, 163, 164, 178, 179,  
    188, 194, 198, 208, 210, 215, 216  
handling 168  
has-a relationship 137  
heap 33  
heterogeneous  
    data structures 86, 90, 220  
    list 89  
hexadecimal 226  
hierarchy  
    levels 222  
    of classes 145, 222  
    of procedures 125  
Hollywood principle 150

- hook method 123
- hybrid languages 16, 139, 223
- hypertext 152
- identifiers 226, 228
- identifying
  - classes 127, 128
  - data fields 129
  - methods 128
- idle 168
- if statement 239
- import 27, 247
- IN 236
- INC 247
- Incl 49, 247
- independent compilation 28
- information hiding 7, 39, 47, 130, 220
- inheritance 8, 17, **53**, 122, 221
  - multiple 66, 112
  - relationship 55
- Init 49
- InitGraphic 204
- initialization of objects **99**
- initialization procedure 99
- InitRider 260, 261
- input/output 263
  - procedures 106
- Insert 40, 45, 171, 174, 180, 203, 204, 216
- inserting text 172
- insertion point 172
- Int 264
- integer types 226, 230, 250
- interactive programs 116, 170
- interchangeable components 29
- interface 25, 27, 67, 132, 135, 161, 171, 186, 214, 219
  - checking 27
  - criteria 133
  - description 27
  - design 133, 135
  - of abstract classes 71
  - of Oberon0 168
- InterfaceItem 135
- interpretation of messages 61
- Intersect 50
- InvertBlock 207, 208, 209, 260
- IO 263
- IS 61, 236, 259
- Is-a relationship 55, 108, 137
- iteration 22
- iterator class 120
- iterators **120**, 121
- keyboard input 168, 185
- keyboard operations 260, 261
- language definition **225**
- language extension 41
- late binding 67
- layer around Text 95
- learning effort 221
- len 173, 246
- length 230
- lexical rules 226
- Line 88, 190
- line descriptor 189, 190
- line metrics 189
- linker 223
- linking loader 31
- Load 172, 174, 180, 182, 203, 206, 213, 215, 260
- local 228, 243
- locality of code and data 51, 219
- localization 89
- localized code 76
- location 187
- logical entities 128
- logical operators **235**
- LONG 246
- LONGINT 230
- LONGREAL 230
- Lookup 26
- Loop 169
- loop statement **241**
- low-level operations 254
- LSH 254
- MacApp 155

machine-oriented programs 13  
maintainability 219  
match 232, 243, 245, 251  
matching formal parameter lists **252**  
MAX 246  
MeasureLine 197  
memory allocation 20  
menu 163  
menu frame 159  
menu selection 149  
message 10, 44, **67**  
    flow 117  
    forwarding 111  
    handler 74, 118  
    handling 111  
    understanding 76  
message records 58, **74**, 121  
    advantages 76  
    drawbacks 77  
    interpretation 77  
    using 75  
messages 10, **67**  
    to ASCII texts 171  
    to elements 178  
    to figures 202  
    to frames 161  
    to graphics 203  
    to text frames 187  
    to texts 177  
    to viewers 161  
methods 10, 17, 43, 44, 136, 145  
    notation 46  
Meyer 64  
MIN 246  
mnemonic operation names 14  
MOD 235, 251  
Model/View/Controller 116, 150  
modifications 126, 130, 133  
Modify 161, 163, 164, 188, 194, 208  
Modula-2 17, 225  
modularity 224  
module 38, **247**  
    body 28  
    interface 25  
modules 14, **25**  
    and classes 47  
purpose 29  
monadic operators 235  
mouse  
    buttons 206  
    clicks 168, 186  
    movements 122  
    operations 260, 261  
Move 161, 163, 164, 203, 204, 212,  
    255, 260, 262  
MoveSelected 203, 205  
multiple inheritance 66, **112**  
    avoiding 113  
    drawbacks 112  
    run-time costs 113  
multiple interfaces 85  
multiprogramming 168  
MVC concept 116, 117, 159, 169, 170,  
    185, 199, 201  
MVC framework 150  
  
name clashes 112  
NameToObj 210, 260, 262  
naming 51  
naming conventions 134  
nested blocks 228  
nested comments 227  
Neutralize 161, 163, 164, 188, 194,  
    203, 204, 205  
NEW 72, 100, 165, 197, 209, 232,  
    247, 255  
NewFile 260, 261  
NewMenu 197  
next 163  
NIL 232, 236, 261  
NL 264  
Node 81  
nonterminal symbols 225  
notify messages 176  
NotifyChangeMsg 204, 210  
NotifyDelMsg 179, 198  
NotifyInsMsg 179, 198  
NotifyReplMsg 179, 198  
numbers 226  
numeric types 230, 250, 251

- Oberon 17, 90, 96, 104, 157, 217, 223, 225, 255
  - books 265
  - commands **29**, 159, 198
  - documentation 265
  - environment **255**
  - interface extraction 27
  - module 168
  - operating system 17, 29
  - run-time environment 17
- Oberon System 2, 117, 118, 151, 154, 157, 168, 212, **255**, 265
- Oberon-2 16, **17**, **225**
  - basic types 18
  - compiler 265
  - expressions 20
  - features **18**, 225
  - procedures **23**
  - syntax **225**, **253**
- Oberon0 157, 169
  - components 158
  - source code 266
- object (see objects)
- object-oriented design **125**, **126**
  - Abbott's method **131**
  - abstract classes 136
  - advantages 126
  - class library 221
  - class relationships 136
  - considerations **128**, **129**
  - CRC cards **131**
  - data abstraction 140
  - data fields 129
  - deriving classes **131**
  - drawbacks 127
  - errors **141**
  - finding classes 128
  - for reusability 136
  - identifying classes 128
  - identifying methods 128
  - learning from others 130
  - mistakes 129
  - modeling 128
  - patterns 130
  - subclasses 135
  - verbal specification **131**
- when classes 139
- without classes 140
- object-oriented programming 1, 10, 63, 147, 157
  - assessment 224
  - benefits of 219
  - contrast with conventional **10**
  - costs and benefits 219
  - costs of **221**
  - future **224**
  - languages 7, 17
  - properties 16
  - strengths 1
- object-oriented programming languages 7, 17
  - history **15**
- object-oriented terminology 10
- object-oriented thinking **6**
- Object-Pascal 16, 223
- objects 10, 105, 126, 259
  - as variable parameters 58
  - finding 126
  - in texts 151
- ODD 246
- offsets 258
- OldFile 260, 261
- Open 200, 210, 211
- open array 20, 244
- open array parameter 24
- open/closed principle 64
- operands 233
- operating system 255, 256
- operations 139
- operators 227, 233, **234**
- OR 235
- ORD 246
- org 190, 191
- orgX 208
- orgY 208
- OS 157, 259
- overriding methods 55, 122, 123, 166, 222
- parallel 168
- parameter 238

- actual 24
- formal 24
- open array 24
- value 57
- variable 24, 58
- parameter
  - list 234
  - matching lists 252
  - passing 245
  - passing rules 244
- parameterless 256
- Parnas 7
- Pascal 17, 33
- persistent objects 103
- pictures in texts 213
- platforms 265
- pointer 46, 223, 234, 236
  - assignment 57, 59
  - base type 232
  - type 46, 232, 238, 251, 254
  - variable 19
- polymorphism 9, 220
- Port 91
- pos 173, 190, 264
- position 187, 190
- precedence 234
- predeclared
  - functions 229, 230
  - identifiers 228
  - procedures 246
- primitives 209
- Print 26
- PrinterPort 92
- priority queue 40, 47
  - abstract data structure 37
  - concrete 33
- PriorityQueue 37
- PriorityQueues 40, 43, 44
- private fields 231
- problem-oriented specifications 13
- procedural interface 7
- procedure 23, 48, 234, 241
  - body 242
  - call 238, 243
  - components of 243
  - constants 46
  - declaration 242
  - heading 242
  - identifier 242
  - invocation 238
  - libraries 148
  - table index 259
  - type 232, 251
- procedure variables 9, 20, 45, 46, 121
  - drawbacks 46
- procedure-oriented thinking 5
- productivity 147
- program hierarchy 125
- programming by difference 156
- programming by extension 11
- projection 57, 237
- proper procedures 241, 243, 247, 255
- PTR 254
- public fields 231
- PUT 255
- PUTREG 255
- qualified identifiers 228, 247
- Queue 40, 45
- Read 171, 174, 181, 259, 261, 264
- read-only export 28, 37, 228, 234
- readability 219
- ReadChars 259, 262
- ReadInt 259, 261
- ReadKey 260, 261
- ReadLInt 259, 262
- ReadNextElem 177, 181
- ReadObj 259, 262
- ReadString 259, 262
- real types 226, 230, 250, 264
- real-world system 128
- receiver 44, 111, 242, 243, 245
- record 19, 234, 258
  - as variable parameter 60
  - assignment 58, 60
  - fields 228
  - type 231, 232, 236, 237, 245, 251, 259
- Rectangle 54, 56, 74, 88, 212
- rectangles 211

Rectangles 0 212  
 recursion 243  
 redefinition 245  
 RedrawFrom 197  
 redundancy 133, 134  
 referencing 228, 232  
 Register 260, 261  
 relational expressions 21  
 relational operators **236**  
 relations 236  
 relationships between classes **136**, 137  
 Remove 38, 41, 45  
 RemoveCaret 188, 193  
 RemoveSelection 188, 194  
 repeat statement **240**  
 replaceable behavior 29, 79, **91**  
 requirements definition 127  
 reserved words **227**  
 resizing 158  
 result type 241, 252  
 return statement **241**, 243  
 reusability 11, 127, 133, 136, 140, 220  
 reusable design 98  
 reused components 220  
 rider 259, 261  
 robustness 133  
 ROT 254  
 run-time  
     behavior 91  
     data structures 255  
     extensibility 101, 153  
     extensions 90  
     inefficiency. 222  
     type check 61  
     type checking 223  
     type information 104  
  
 same type **250**  
 scanner 256, 263  
 scheduler 92  
 scope 24, 228, 244  
 scope rules **228**  
 screen operations 260  
 screen position 190  
 ScreenPort 92  
 scroll bar 186  
 selBeg 191  
 selected 203  
 selection 22, 239  
 SelectionMsg 199  
 selEnd 191  
 semantic gap 13  
 semifinished products **96**, 97, 184, 220  
 separate compilation 28  
 sequence 240  
 server 136  
 services 130  
 Set 49, 213, 230, 236, 259, 261, 264  
 set  
     constructor 236  
     diagram 95  
     expressions 22  
     operators **236**  
 SetBox 204, 212  
 SetCaret 188, 194  
 SetFocus 161, 163  
 SetPos 171, 174, 181  
 Sets 49  
 SetSelection 188, 194, 203, 204, 205  
 SetToParameters 264  
 Shape 202, 203  
 shape methods 204  
 shapes 204  
 Shapes0 **202**, 203  
 shell 255  
 SHORT 246  
 short circuit evaluation 21  
 SHORTINT 230  
 simplicity 133  
 Simula 15  
 SIZE 246  
 Smalltalk 15, 47, 91, 113, 139, 221,  
         222  
 smart linkers 223  
 socket 184  
 source code 157  
 specialization 55  
 split position 183  
 standard  
     behavior 70  
     components 221

- operations 14
- types 14
- state 154, 168
- statement 22, **237**
  - case 239
  - elementary 22
  - exit 242
  - for 240
  - if 239
  - loop 241
  - repeat 240
  - sequence 237, **239**, 240, 241, 242, 247
  - structured 22
  - while 240
  - with 242
- static
  - binding 67
  - type **59**, 89, 233, 236, 237, 242
  - type checking 15, 61
- stepwise refinement 125
  - advantages 126
  - drawbacks 126
- storage inefficiency 223
- Store 172, 175, 180, 182, 200, 203, 206, 210, 211, 213, 215, 260
- Str 264
- Stream 70, 71, 109
- String 26, 107, 227, 238
- string constant 251
- StringNode 83
- structured statements 22, 237
- structured types 229
- structuring medium 29, 79, 184
- Style 94
- StyledText 94
- subclass 53, 144
- subsystems 147, 148, 153
- Subtract 50
- superclass 53, 144
- symbol file 27
- symbols 225
- symmetry 106
- syntax of Oberon-2 **225**, **253**
- SYSTEM 254
- system-specific parts 130
- techniques for OOP **99**
- Terminal 68
- terminal symbols 225
- Text 93, 108, 173, 179, 191
- text
  - buffer 172
  - editor **169**, 217
  - insertion 172
  - methods 173
  - with floating objects 151
- TextBox 55
- TextFigure 108
- TextFrame 97, 170
- TextFrames 0 **185**, 186, 190
- Texts 0 179
- Time 260, 262
- title bar 158, 166
- top-down 125
- Track 210
- tracking mouse movement 122
- transitivity 55
- Tree 81, 82, 85
- twin class 114, 115
- type
  - BYTE 254
  - checking 28
  - declaration **229**
  - descriptor 104, 223, 258, 259
  - dynamic 233
  - extension 17, **53**, 220, **250**
  - guard 61, 184, 234, 242
  - hierarchy 55
  - inclusion **250**
  - names 108
  - PTR 254
  - SET 236
  - static 233
  - tag 258
  - test 61, 236, 242, 259
- type-bound procedures 17, 43, 233, 242, **245**, 259
- uniformity 221
- unloading modules 31, 256
- up-calls 103

- Update 216
- update frame 216
- user input 168
- user-defined data types 14
- user-defined procedures 14
- user friendliness 154
- uses relationship 125, 137
  
- VAL 254
- value parameter 24, 57, 238, 243, 244, 245, 252
- variable
  - declarations **232**
  - names 14
  - parameter 24, 58, 60, 236, 238, 243, 244, 245, 252
- variant records 87
- variants 75, 86, 109, 130, 139, 168, 220
- distinguishing 87
- verbal specification **131**
- Viewer 160, 161, 163, 166, 167, 170, 216
  - list 162
  - methods 163
  - new 167
  - position 167
  - system **158**
  
- ViewerAt 165
- viewers 118, 158
- Viewers0 160, 162
- virtual languages 135, 221
- visibility 231
  
- w 163, 179
- while statements **240**
- windows 158
- Wirth 2, 17
- with statement 75, 77, 242
- wrapping 108, 114
- Write 172, 174, 181, 259, 262
- WriteChars 259, 262
- WriteElem 177, 182
- WriteInt 259, 262
- WriteLInt 259, 262
- WriteObj 259, 262
- WriteString 259, 262
  
- x 162, 190
- Xerox PARC 15
  
- y 162, 190