Component Pascal Language Report

Authors:	Oberon microsystems, Inc. March 2001, last update October 2006
Authors of Oberon-2 report:	Hanspeter Mössenböck, Niklaus Wirth Institut für Computersysteme, ETH Zürich October 1993
Author of Oberon report:	Niklaus Wirth Institut für Computersysteme, ETH Zürich 1987

Oberon microsystems, Inc.
Technoparkstrasse 1
CH-8005 Zürich
Switzerland

Tel +41 1 445 17 51
Fax +41 1 445 17 52

info@oberon.ch
www.oberon.ch

0. Contents

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

1. Introduction

Component Pascal is Oberon microsystems' refinement of the Oberon-2 language. Oberon microsystems thanks H. Mössenböck and N. Wirth for the friendly permission to use their Oberon-2 report as basis for this document.

Component Pascal is a general-purpose language in the tradition of Pascal, Modula-2, and Oberon. Its most important features are block structure, modularity, separate compilation, static typing with strong type checking (also across module boundaries), type extension with methods, dynamic loading of modules, and garbage collection.

Type extension makes Component Pascal an object-oriented language. An object is a variable of an abstract data type consisting of private data (its state) and procedures that operate on this data. Abstract data types are declared as extensible records. Component Pascal covers most terms of object-oriented languages by the established vocabulary of imperative languages in order to minimize the number of notions for similar concepts.

Complete type safety and the requirement of a dynamic object model make Component Pascal a component-oriented language.

This report is not intended as a programmer's tutorial. It is intentionally kept concise. Its function is to serve as a reference for programmers. 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 to commit the definition when a general commitment appears as unwise.

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

It is recommended to minimize the use of procedure types and super calls, since they are considered obsolete. They are retained for the time being, in order to simplify the use of existing Oberon-2 code. Support for these features may be reduced in later product releases.

2. Syntax

An extended Backus-Naur formalism (EBNF) is used to describe the syntax of Component Pascal: Alternatives are separated by |. Brackets [and] denote optionality of the enclosed expression, and braces { and } denote its repetition (possibly 0 times). Ordinary parentheses (and) are used to group symbols if necessary. Non-terminal 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., ":=").

3. Vocabulary and Representation

The representation of (terminal) symbols in terms of characters is defined using ISO 8859-1, i.e., the Latin-1 extension of the ASCII character set. Unicode (16 bit) characters are allowed in string constants only. 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 in comments, and blanks in strings). They are ignored unless they are essential to separate two consecutive symbols. Capital and lower-case letters are considered as distinct.

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

1. *Identifiers* are sequences of letters, digits, and underscores. The first character must not be a digit.

ident = (letter | "_") {letter | "_" | digit}.
letter = "A" .. "Z" | "a" .. "z" | "À".."Ö" | "Ø".."ö" | "ø".."ÿ".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

Examples: x Scan Oberon2 GetSymbol firstLetter

2. *Numbers* are (unsigned) integer or real constants. The type of an integer constant is INTEGER if the constant value belongs to INTEGER, or LONGINT otherwise (see 6.1). If the constant is specified with the suffix 'H' or 'L', the representation is hexadecimal, otherwise the representation is decimal. The suffix 'H' is used to specify 32-bit constants in the range -2147483648 .. 2147483647. At most 8 significant hex digits are allowed. The suffix 'L' is used to specify 64-bit constants.

A real number always contains a decimal point. Optionally it may also contain a decimal scale factor. The letter E means "times ten to the power of". A real number is always of type REAL.

number = integer | real.
integer = digit {digit} | digit {hexDigit} ("H" | "L").
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = "E" ["+" | "-"] digit {digit}.
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".

Examples:

1234567	INTEGER 1234567
0DH	INTEGER 13
12.3	REAL 12.3
4.567E8	REAL 456700000
0FFFF0000H	INTEGER -65536
0FFFF0000L	LONGINT 4294901760

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 (") quote marks. The opening quote must be the same as the closing quote 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. Characters in string constants are allowed to be Unicode (16 bit) characters.

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

Examples: "Component Pascal" "Don't worry!" "x" “αβ“

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.

+	:=	ABSTRACT	EXTENSIBLE	POINTER
-	^	ARRAY	FOR	PROCEDURE

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

*	=	BEGIN	IF	RECORD
/	#	BY	IMPORT	REPEAT
~	<	CASE	IN	RETURN
&	>	CLOSE	IS	THEN
.	<=	CONST	LIMITED	TO
,	>=	DIV	LOOP	TYPE
;	..	DO	MOD	UNTIL
|	:	ELSE	MODULE	VAR
$		ELSIF	NIL	WHILE
(	)	EMPTY	OF	WITH
[	]	END	OR	
{	}	EXIT	OUT	

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.

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 equally named objects which 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 declaration of a type *T* containing references to another type *T1* may occur at a point where *T1* is still unknown. The declaration of *T1* must follow in the same block to which *T* is local;
4. Identifiers denoting record fields (see 6.3) or methods (see 10.2) are valid in record designators only.

An identifier declared in a module block may be followed by an export mark (" * " or " - ") 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 Ch.11). The identifier is then denoted as *M.x* in these modules and is called a *qualified identifier*. Variables and record fields marked with " - " in their declaration are *read-only* (variables and fields) or *implement-only* (methods) in importing modules.

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

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

ABS	(10.3)	INTEGER	(6.1)
ANYPTR	(6.1)	FALSE	(6.1)
ANYREC	(6.1)	LEN	(10.3)

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

ASH	(10.3)	LONG	(10.3)
ASSERT	(10.3)	LONGINT	(6.1)
BITS	(10.3)	MAX	(10.3)
BOOLEAN	(6.1)	MIN	(10.3)
BYTE	(6.1)	NEW	(10.3)
CAP	(10.3)	ODD	(10.3)
CHAR	(6.1)	ORD	(10.3)
CHR	(10.3)	REAL	(6.1)
DEC	(10.3)	SET	(6.1)
ENTIER	(10.3)	SHORT	(10.3)
EXCL	(10.3)	SHORTCHAR	(6.1)
HALT	(10.3)	SHORTINT	(6.1)
INC	(10.3)	SHORTREAL	(6.1)
INCL	(10.3)	SIZE	(10.3)
INF	(6.1)	TRUE	(6.1)

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 (Ch.8) or predeclared functions (Ch.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)}
```

6. Type Declarations

A data type determines the set of values which variables of that type may assume, and the operators that are applicable. A type declaration 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. A structured type cannot contain itself.

TypeDeclaration = IdentDef "=" Type.
Type = Qualident | ArrayType | RecordType | PointerType | ProcedureType.

Examples:

```
Table = ARRAY N OF REAL
Tree = POINTER TO Node
Node =  EXTENSIBLE RECORD
   key: INTEGER;
   left, right: Tree
```


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

```
END
CenterTree = POINTER TO CenterNode
CenterNode = RECORD (Node)
   width: INTEGER;
   subnode: Tree
END
Object = POINTER TO ABSTRACT RECORD END;
Function = PROCEDURE (x: INTEGER): INTEGER
```

6.1 Basic Types

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

1.	BOOLEAN	the truth values TRUE and FALSE
2.	SHORTCHAR	the characters of the Latin-1 character set (0X .. 0FFX)
3.	CHAR	the characters of the Unicode character set (0X .. 0FFFFX)
4.	BYTE	the integers between MIN(BYTE) and MAX(BYTE)
5.	SHORTINT	the integers between MIN(SHORTINT) and MAX(SHORTINT)
6.	INTEGER	the integers between MIN(INTEGER) and MAX(INTEGER)
7.	LONGINT	the integers between MIN(LONGINT) and MAX(LONGINT)
8.	SHORTREAL	the real numbers between MIN(SHORTREAL) and MAX(SHORTREAL), the value INF
9.	REAL	the real numbers between MIN(REAL) and MAX(REAL), the value INF
10.	SET	the sets of integers between 0 and MAX(SET)

Types 4 to 7 are *integer types*, types 8 and 9 are *real types*, and together they are called *numeric types*. They form a hierarchy; the larger type *includes* (the values of) the smaller type:

REAL >= SHORTREAL >= LONGINT >= INTEGER >= SHORTINT >= BYTE

Types 2 and 3 are character types with the type hierarchy:

CHAR >= SHORTCHAR

6.2 Array Types

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

```
ArrayType     = ARRAY [Length {"," Length}] OF Type.
Length        = ConstExpression.
```

A type of the form

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

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

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 6.4), element types of open array types, and formal parameter types (see 10.1). Examples:

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

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 elements of record variables (see 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   = RecAttributes RECORD ["("BaseType")"] FieldList {";" FieldList} END.
RecAttributes = [ABSTRACT | EXTENSIBLE | LIMITED].
BaseType     = Qualident.
FieldList    = [IdentList ":" Type].
IdentList    = IdentDef {"," IdentDef}.
```

The usage of a record type is restricted by the presence or absence of one of the following attributes: ABSTRACT, EXTENSIBLE, and LIMITED.

A record type marked as ABSTRACT cannot be instantiated. No variables or fields of such a type can ever exist. Abstract types are only used as base types for other record types (see below).
Variables of a LIMITED record type can only be allocated inside the module where the record type is defined. The restriction applies to static allocation by a variable declaration (Ch. 7) as well as to dynamic allocation by the standard procedure NEW (Ch. 10.3).

Record types marked as ABSTRACT or EXTENSIBLE are extensible, i.e., a record type can be declared as an extension of such a record type. In the example

```
T0 = EXTENSIBLE 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 App. A). An extended type *T1* consists of the fields of its base type and of the fields which are declared in *T1*. All identifiers declared in the extended record must be different from the identifiers declared in its base type record(s). The base type of an abstract record must be abstract.

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

Alternatively, a pointer type can be specified as the base type. The record base type of the pointer is used as the base type of the declared record in this case.
A record which is an extension of a hidden (i.e., non-exported) record type may not be exported.
Each record is implicitly an extension of the predeclared type ANYREC. ANYREC does not contain any fields and can only be used in pointer and variable parameter declarations.

Summary of attributes:

attribute	*extension*	*allocate*
none	no	yes
EXTENSIBLE	yes	yes
ABSTRACT	yes	no
LIMITED	in defining module only	

Examples of record type declarations:

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

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

6.4 Pointer Types

Variables of a pointer type *P* assume as values pointers to variables of some type *T*. *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 *T1* is an extension of *T*, and *P1* is of type POINTER TO *T1*, then *P1* is also an extension of *P*.

```
   PointerType = POINTER TO Type.
```

If *p* is a variable of type *P* = POINTER TO *T*, a call of the predeclared procedure *NEW(p)* (see 10.3) allocates a 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 type the allocation has to be done with *NEW(p, e_0, ..., e_{n-1})* where *T* is allocated with lengths given by the expressions e_0, ..., e_{n-1}. In either case a pointer to the allocated variable is assigned to *p*. *p* is of type *P*. The *referenced* variable *p^* (pronounced as *p-referenced*) is of type *T*. Any pointer variable may assume the value NIL, which points to no variable at all.
All fields or elements of a newly allocated record or array are cleared, which implies that all embedded pointers and procedure variables are initialized to NIL.
The predeclared type ANYPTR is defined as POINTER TO ANYREC. Any pointer to a record type is therefore an extension of ANYPTR. The procedure NEW cannot be used for variables of type ANYPTR.

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

6.5 Procedure Types

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

ProcedureType = PROCEDURE [FormalParameters].

6.6 String Types

Values of a string type are sequences of characters terminated by a null character (0X). The *length* of a string is the number of characters it contains excluding the null character.
Strings are either constants or stored in an array of character type. There are no predeclared identifiers for string types because there is no need to use them in a declaration.
Constant strings which consist solely of characters in the range 0X..0FFX and strings stored in an array of SHORTCHAR are of type Shortstring, all others are of type String.

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 of their value 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 methods (see 10.2).

Examples of variable declarations (refer to examples in Ch. 6):

```
i, j, k: INTEGER
x, y: REAL
p, q: BOOLEAN
s: SET
F: Function
a: ARRAY 100 OF REAL
w: ARRAY 16 OF
   RECORD
      name: ARRAY 32 OF CHAR;
      count: INTEGER
   END
t, c: Tree
```


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

8. Expressions

Expressions are constructs denoting 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.

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 Ch. 4 and 11) and may be followed by *selectors* if the designated object is an element of a structure.

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

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 $a[e_0, e_1, ..., e_n]$ stands for $a[e_0][e_1]...[e_n]$. If *r* designates a record, then *r.f* denotes the field *f* of *r* or the method *f* of the dynamic type of *r* (Ch. 10.2). If *a* or *r* are read-only, then also *a[e]* and *r.f* are read-only.

If *p* designates a pointer, *p^* denotes the variable which is referenced by *p*. The designators *p^.f*, *p^[e],* and *p^$* may be abbreviated as *p.f*, *p[e],* and *p$*, i.e., record, array, and string selectors imply dereferencing. Dereferencing is also implied if a pointer is assigned to a variable of a record or array type (Ch. 9.1), if a pointer is used as actual parameter for a formal parameter of a record or array type (Ch. 10.1), or if a pointer is used as argument of the standard procedure LEN (Ch. 10.3).

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 an IN or VAR parameter of record type or *v* is a pointer to a record type, 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 10.1).

If *a* designates an array of character type, then *a$* denotes the null terminated string contained in *a*. It leads to a run-time error if *a* does not contain a 0X character. The $ selector is applied implicitly if a is used as an operand of the concatenation operator (Ch. 8.2.4), a relational operator (Ch. 8.2.5), or one of the predeclared procedures LONG and SHORT (Ch. 10.3).

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

```
i                   (INTEGER)
a[i]                (REAL)
```


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

w[3].name[i]	(CHAR)
t.left.right	(Tree)
t(CenterTree).subnode	(Tree)
w[i].name$	(String)

8.2 Operators

Four classes of operators with different precedences (binding strengths) are syntactically distinguished in expressions. The operator ~ has the highest precedence, followed by multiplication operators, addition operators, and relations. Operators of the same precedence associate from left to right. For example, x-y-z stands for (x-y)-z.

Expression	= SimpleExpression [Relation SimpleExpression].
SimpleExpression	= ["+" | "-"] Term {AddOperator Term}.
Term	= Factor {MulOperator Factor}.
Factor	= Designator | number | character | string | NIL | Set | "(" Expression ")" | "~" Factor.
Set	= "{" [Element {"," Element}] "}".
Element	= Expression [".." Expression].
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 App. A).

8.2.1 Logical operators

OR	logical disjunction	p OR q	"if p then TRUE, else q"
&	logical conjunction	p & q	"if p then q, else FALSE"
~	negation	~ p	"not p"

These operators apply to BOOLEAN operands and yield a BOOLEAN result. The second operand of a disjunction is only evaluated if the result of the first is FALSE. The second oprand of a conjunction is only evaluated if the result of the first is TRUE.

8.2.2 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 REAL if the operation is a division (/) or one of the operand types is a REAL. Otherwise the result type is

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

SHORTREAL if one of the operand types is SHORTREAL, LONGINT if one of the operand types is LONGINT, or INTEGER in any other case. If the result of a real operation is too large to be represented as a real number, it is changed to the predeclared value INF with the same sign as the original result. Note that this also applies to 1.0/0.0, but not to 0.0/0.0 which has no defined result at all and leads to a run-time error. 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:

x = (x DIV y) * y + (x MOD y)
0 <= (x MOD y) < y or 0 >= (x MOD y) > y

Note: x DIV y = ENTIER(x / y)

Examples:

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

Note:
(-5) DIV 3 = -2
but
-5 DIV 3 = -(5 DIV 3) = -1

8.2.3 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) which are not elements of *x*. Set operators are not associative *((a+b)-c ≠ a+(b-c))*.
A set constructor defines the value of a set by listing its elements between curly brackets. The elements must be integers in the range 0..MAX(SET). A range *a..b* denotes all integers *i* with *i* >= *a* and *i* <= *b*.

8.2.4 String operators

+	string concatenation

The concatenation operator applies to operands of string types. The resulting string consists of the characters of the first operand followed by the characters of the second operand. If both operands are of type Shortstring the result is of type Shortstring, otherwise the result is of type String.

8.2.5 Relations

=	equal

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

#	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 the numeric types, character types, and string types. 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 an integer in the range 0..MAX(SET), 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 an IN or VAR parameter of record type or *v* is a pointer to a record type, and if
2. *T* is an extension of the static type of *v*

Examples of expressions (refer to examples in Ch.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<=i) & (i<100)	BOOLEAN
t.key = 0	BOOLEAN
k IN {i..j-1}	BOOLEAN
w[i].name$ <= "John"	BOOLEAN
t IS CenterTree	BOOLEAN

9. Statements

Statements denote actions. There are elementary and structured statements. Elementary statements are not composed of any parts that are themselves statements. They are the assignment, the procedure call, 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] ].
```


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

9.1 Assignments

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

Assignment = Designator ":=" Expression.

If an expression *e* of type *Te* is assigned to a variable *v* of type *Tv*, the following happens:

1. if *Tv* and *Te* are record types, all fields of that type are assigned.
2. if *Tv* and *Te* are pointer types, the dynamic type of *v* becomes the dynamic type of *e*;
3. if *Tv* is an array of *character type* and *e* is a string of length *m* < *LEN(v)*, *v[i]* becomes *ei* for *i* = 0..*m*-1 and *v[m]* becomes 0X. It leads to a run-time error if *m* >= *LEN(v)*.

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

```
i := 0
p := i = j
x := i + 1
k := Log2(i+j)
F := Log2     (* see 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
```

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 Ch. 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. This expression is evaluated before the procedure activation, and the resulting value is assigned to the formal parameter (see also 10.1).

ProcedureCall = Designator [ActualParameters].

Examples:

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


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

9.3 Statement Sequences

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

```
StatementSequence = Statement {";" Statement}.
```

9.4 If Statements

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

If statements specify the conditional execution of guarded 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
```

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 be of an *integer* or *character type* that includes the values of all case labels. Case labels are constants, and no value must occur more than once. If the value of the expression does not occur as a label of any case, the statement sequence following the symbol ELSE is selected, if there is one, otherwise 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
```


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

```
|   "0" .. "9": ReadNumber
|   """, "": ReadString
ELSE SpecialCharacter
END
```

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
```

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.
```

9.8 For Statements

A for statement specifies the repeated execution of a statement sequence 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 := beg TO end BY step DO statements END
```

is equivalent to

```
temp := end; v := beg;
IF step > 0 THEN
   WHILE v <= temp DO statements; INC(v, step) END
ELSE
   WHILE v >= temp DO statements; INC(v, step) END
END
```

temp has the *same* type as *v*. *step* must be a nonzero constant expression. If *step* is not specified, it is assumed to be 1.

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

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

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 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.

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 App. A) with the result type specified in the procedure heading (see Ch.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 which contains them.

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 *T0*, the statement

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

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

has the following meaning: if the dynamic type of *v* is *T1*, then the statement sequence *S1* is executed where *v* is regarded as if it had the static type *T1*; else if the dynamic type of *v* is *T2*, then *S2* is executed where *v* is regarded as if it had the static type *T2*; else *S3* is executed. *T1* and *T2* must be extensions of *T0*. 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

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 methods it also specifies the receiver parameter and the attributes (see 10.2). 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 which 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.

Local variables whose types are pointer types or procedure types are initialized to NIL before the body of the procedure is executed.

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

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

If a procedure declaration specifies a *receiver* parameter, the procedure is considered to be a method of the type of the receiver (see 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 App. A) and the names of corresponding parameters must be equal.

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

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*, the latter indicated in the formal parameter list by the presence of one of the keywords VAR, IN, or OUT. 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. Variable parameters can be used for input only (keyword IN), output only (keyword OUT), or input and output (keyword VAR). IN can only be used for array and record parameters. Inside the procedure, input parameters are read-only. Like local variables, output parameters of pointer types and procedure types are initialized to NIL. Other output parameters must be considered as undefined prior to the first assignment in the procedure. 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 procedure can be neither a record nor an array.

```
FormalParameters        = "(" [FPSection {";" FPSection}] ")" [":" Type].
FPSection               = [VAR | IN | OUT] ident {"," ident} ":" Type.
```

Let *f* be the formal parameter and *a* the corresponding actual parameter. If *f* is an open array, then *a* must be *array compatible* to *f* and the lengths of *f* are taken from *a*. Otherwise *a* must be *parameter compatible* to *f* (see App. A)

Examples of procedure declarations:

```
PROCEDURE ReadInt (OUT 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 (IN 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
```


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

```
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

PROCEDURE Modify (VAR n: Node);
BEGIN
   INC(n.key)
END Modify
```

10.2 Methods

Globally declared procedures may be associated with a record type declared in the same module. The procedures are said to be *methods* 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 VAR or IN parameter of record type *T* or a value parameter of type POINTER TO *T* (where *T* is a record type). The method is bound to the type *T* and is considered local to it.

```
ProcedureHeading    = PROCEDURE [Receiver] IdentDef
                      [FormalParameters] MethAttributes.
Receiver            = "(" [VAR | IN] ident ":" ident ")".
MethAttributes      = ["," NEW] ["," (ABSTRACT | EMPTY | EXTENSIBLE)].
```

If a method M is bound to a type T0, it is implicitly also bound to any type T1 which is an extension of T0. However, if a method M' (with the same name as M) is declared to be bound to T1, this overrides the binding of M to T1. M' is considered a redefinition of M for T1. The formal parameters of M and M' must match, except if M is a function returning a pointer type. In the latter case, the function result type of M' must be an extension of the function result type of M (covariance) (see App. A). If M and T1 are exported (see Chapter 4) M' must be exported too.
If M is not exported, M' must not be exported either. If M and M' are exported, they must use the same export marks.

The following attributes are used to restrict and document the desired usage of a method:
NEW, ABSTRACT, EMPTY, and EXTENSIBLE.

NEW must be used on all newly introduced methods and must not be used on redefining methods. The attribute helps to detect inconsistencies between a record and its extension when one of the two is changed without updating the other.

Abstract and empty method declarations consist of a procedure header only. Abstract methods are never called. A record containing abstract methods must be abstract. A method redefined by an abstract method must be abstract. An abstract method of an exported record must be exported. Calling an empty method has no effect. Empty methods may not return function results and may not have OUT parameters. A record containing new empty methods must be extensible or abstract. A method redefined by an empty method must be empty or abstract. Abstract or empty methods are usually redefined (implemented) in a record extension. They may not be called via super calls. A

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

concrete (nonabstract) record extending an abstract record must implement all abstract methods bound to the base record.

Concrete methods (which contain a procedure body) are either extensible or final (no attribute). A final method cannot be redefined in a record extension. A record containing extensible methods must be extensible or abstract.

If *v* is a designator and *M* is a method, then *v.M* denotes that method *M* which is bound to the dynamic type of *v*. Note, that this may be a different method than the one bound to the static type of *v*. *v* is passed to *M*'s receiver according to the parameter passing rules specified in Chapter 10.1.

If *r* is a receiver parameter declared with type *T*, *r.M^* denotes the method *M* bound to the base type of *T* (super call). In a forward declaration of a method the receiver parameter must be of the *same* type as in the actual method declaration. The formal parameter lists of both declarations must *match* (App. A) and the names of corresponding parameters must be equal.

Methods marked with " - " are "implement-only" exported. Such a method can be redefined in any importing module but can only be called within the module containing the method declaration. (Currently, the compiler also allows super calls to implement-only methods outside of their defining module. This is a temporary feature to make migration easier.)

Examples:

```
PROCEDURE (t: Tree) Insert (node: Tree), NEW, EXTENSIBLE;
   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 method of Tree *)
END Insert

PROCEDURE (obj: Object) Draw (w: Window), NEW, ABSTRACT

PROCEDURE (obj: Object) Notify (e: Event), NEW, EMPTY
```


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

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 *y* for expressions, and *T* for a type. The first matching line gives the correct result type.

Function procedures

Name	Argument type	Result type	Function
ABS(x)	<= INTEGER	INTEGER	absolute value
	real type, LONGINT	type of x	
ASH(x, y)	x: <= INTEGER	INTEGER	arithmetic shift (x * 2^y)
	x: LONGINT	LONGINT	
	y: integer type		
BITS(x)	INTEGER	SET	{i | ODD(x DIV 2^i)}
CAP(x)	character type	type of x	x is a Latin-1 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, x)	v: array; x: integer constant	INTEGER	length of v in dimension x (first dimension = 0)
LEN(v)	array type	INTEGER	equivalent to LEN(v, 0)
	String	INTEGER	length of string (not counting 0X)
LONG(x)	BYTE	SHORTINT	identity
	SHORTINT	INTEGER	
	INTEGER	LONGINT	
	SHORTREAL	REAL	
	SHORTCHAR	CHAR	
	Shortstring	String	
MAX(T)	T = basic type	T	maximum value of type T
	T = SET	INTEGER	maximum element of a set
MAX(x, y)	<= INTEGER	INTEGER	the larger value of x and y
	integer type	LONGINT	
	<= SHORTREAL	SHORTREAL	
	numeric type	REAL	
	SHORTCHAR	SHORTCHAR	
	character type	CHAR	
MIN(T)	T = basic type	T	minimum value of type T
	T = SET	INTEGER	0
MIN(x, y)	<= INTEGER	INTEGER	the smaller of x and y
	integer type	LONGINT	
	<= SHORTREAL	SHORTREAL	
	numeric type	REAL	
	SHORTCHAR	SHORTCHAR	
	character type	CHAR	
ODD(x)	integer type	BOOLEAN	x MOD 2 = 1

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

ORD(x)	CHAR	INTEGER	ordinal number of x
	SHORTCHAR	SHORTINT	ordinal number of x
	SET	INTEGER	(SUM i: i IN x: 2^i)
SHORT(x)	LONGINT	INTEGER	identity
	INTEGER	SHORTINT	identity
	SHORTINT	BYTE	identity
	REAL	SHORTREAL	identity (truncation possible)
	CHAR	SHORTCHAR	projection
	String	Shortstring	projection
SIZE(T)	any type	INTEGER	number of bytes required by T

SIZE cannot be used in constant expressions because its value depends on the actual compiler implementation.

Proper procedures

Name	Argument types	Function
ASSERT(x)	x: Boolean expression	terminate program execution if not x
ASSERT(x, n)	x: Boolean expression; n: integer constant	terminate program execution if not 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, 0 <= x <= MAX(SET)	v := v - {x}
HALT(n)	integer constant	terminate program execution
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, 0 <= x <= MAX(SET)	v := v + {x}
NEW(v)	pointer to record or fixed array	allocate v ^
NEW(v, x_0, ..., x_n)	v: pointer to open array; xi: integer type	allocate v ^ with lengths x_0.. x_n

In ASSERT(x, n) and HALT(n), the interpretation of *n* is left to the underlying system implementation.

10.4 Finalization

A predeclared method named FINALIZE is associated with each record type as if it were declared to be bound to the type ANYREC:

```
PROCEDURE (a: ANYPTR) FINALIZE-, NEW, EMPTY;
```

The FINALIZE procedure can be implemented for any pointer type. The method is called at some unspecified time after an object of that type (or a base type of it) has become unreachable via other pointers (not globally anchored anymore) and before the object is deallocated.

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

It is not recommended to re-anchor an object in its finalizer and the finalizer is not called again when the object repeatedly becomes unreachable. Multiple unreachable objects are finalized in an unspecified order.

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]
                  [CLOSE 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. A module must not import itself. 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 Chapter 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 exported procedures can be activated from the system, and these procedures serve as *commands*.

Variables declared in a module are cleared prior to the execution of the module body. This implies that all pointer or procedure typed variables are initialized to NIL.

The statement sequence following the symbol CLOSE is executed when the module is removed from the system.

Example:

```
MODULE Trees;  (* exports: Tree, Node, Insert, Search, Write, Init *)

   IMPORT StdLog;

   TYPE
      Tree* = POINTER TO Node;
      Node* = RECORD   (* exports read-only: Node.name *)
         name-: POINTER TO ARRAY OF CHAR;
         left, right: Tree
      END;

   PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR), NEW;
      VAR p, father: Tree;
   BEGIN
      p := t;
      REPEAT father := p;
         IF name = p.name^ THEN RETURN END;
```


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

```
      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$) + 1); p.name^ := name$;
    IF name < father.name^ THEN father.left := p ELSE father.right := p END
  END Insert;

  PROCEDURE (t: Tree) Search* (name: ARRAY OF CHAR): Tree, NEW;
    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*, NEW;
  BEGIN
    IF t.left # NIL THEN t.left.Write END;
    StdLog.String(t.name); StdLog.Ln;
    IF t.right # NIL THEN t.right.Write END
  END Write;

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

BEGIN
  StdLog.String("Trees loaded"); StdLog.Ln
CLOSE
  StdLog.String("Trees removed"); StdLog.Ln
END Trees.
```


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

Appendix A: Definition of Terms

Character types	SHORTCHAR, CHAR
Integer types	BYTE, SHORTINT, INTEGER, LONGINT
Real types	SHORTREAL, REAL
Numeric types	integer types, real types
String types	Shortstring, String
Basic types	BOOLEAN, SET, character types, numeric types

Same types
Two variables *a* and *b* with types *Ta* and *Tb* are of the *same* type if
1. *Ta* and *Tb* are both denoted by the same type identifier, or
2. *Ta* is declared in a type declaration of the form *Ta = Tb*, or
3. *a* and *b* appear in the same identifier list in a variable, record field, or formal parameter declaration.

Equal types
Two types *Ta* and *Tb* are *equal* if
1. *Ta* and *Tb* are the *same* type, or
2. *Ta* and *Tb* are open array types with *equal* element types, or
3. *Ta* and *Tb* are procedure types whose formal parameter lists *match*, or
4. *Ta* and *Tb* are pointer types with *equal* base types.

Matching formal parameter lists
Two formal parameter lists *match* if
1. they have the same number of parameters, and
2. they have either *equal* function result types or none, and
3. parameters at corresponding positions have *equal* types, and
4. parameters at corresponding positions are both either value, IN, OUT, or VAR parameters.

Type inclusion
Numeric and character types *include* (the values of) smaller types of the same class according to the following hierarchies:

REAL >= SHORTREAL >= LONGINT >= INTEGER >= SHORTINT >= BYTE
CHAR >= SHORTCHAR

Type extension (base type)
Given a type declaration *Tb = RECORD (Ta) ... END*, *Tb* is a *direct extension* of *Ta*, and *Ta* is a *direct base type* of *Tb*. A type *Tb* is an *extension* of a type *Ta* (*Ta* is a *base type* of *Tb*) if
1. *Ta* and *Tb* are the *same* types, or
2. *Tb* is a *direct extension* of an *extension* of Ta, or
3. *Ta* is of type ANYREC.

If *Pa = POINTER TO Ta* and *Pb = POINTER TO Tb*, *Pb* is an *extension* of *Pa* (*Pa* is a *base type* of *Pb*) if *Tb* is an *extension* of *Ta*.

Assignment compatible
An expression *e* of type *Te* is *assignment compatible* with a variable *v* of type *Tv* if one of the following conditions hold:

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

1. *Te* and *Tv* are *equal* and neither abstract, extensible, or limited record nor open array types;
2. *Te* and *Tv* are numeric or character types and *Tv includes Te*;
3. *Te* and *Tv* are pointer types and *Te* is an *extension* of *Tv*;
4. *Tv* is a pointer or a procedure type and *e* is NIL;
5. *Tv* is a numeric type and *e* is a constant expression whose value is contained in *Tv*;
6. *Tv* is an array of CHAR, *Te* is String or Shortstring, and *LEN(e)* < *LEN(v)*;
7. *Tv* is an array of SHORTCHAR, *Te* is Shortstring, and *LEN(e)* < *LEN(v)*;
8. *Tv* is a procedure type and *e* is the name of a procedure whose formal parameters *match* those of *Tv*.

Array compatible

An actual parameter *a* of type *Ta* is *array compatible* with a formal parameter *f* of type *Tf* if

1. *Tf* and *Ta* are *equal* types, or
2. *Tf* is an open array, *Ta* is any array, and their element types are *array compatible*, or
3. *Tf* is an open array of CHAR and *Ta* is String, or
4. *Tf* is an open array of SHORTCHAR and *Ta* is Shortstring.

Parameter compatible

An actual parameter *a* of type *Ta* is *parameter compatible* with a formal parameter *f* of type *Tf* if

1. *Tf* and *Ta* are *equal* types, or
2. *f* is a value parameter and *Ta* is *assignment compatible* with *Tf*, or
3. *f* is an IN or VAR parameter and *Tf* and *Ta* are record types and *Ta* is an *extension* of *Tf.*

Expression compatible

For a given operator, the types of its operands are *expression compatible* if they conform to the following table. The first matching line gives the correct result type. Type T1 must be an extension of type T0:

operator	*first operand*	*second operand*	*result type*
+ - * DIV MOD	<= INTEGER	<= INTEGER	INTEGER
	integer type	integer type	LONGINT
/	integer type	integer type	REAL
+ - * /	<= SHORTREAL	<= SHORTREAL	SHORTREAL
	numeric type	numeric type	REAL
	SET	SET	SET
+	Shortstring	Shortstring	Shortstring
	string type	string type	String
OR & ~	BOOLEAN	BOOLEAN	BOOLEAN
= # < <= > >=	numeric type	numeric type	BOOLEAN
	character type	character type	BOOLEAN
	string type	string type	BOOLEAN
= #	BOOLEAN	BOOLEAN	BOOLEAN
	SET	SET	BOOLEAN
	NIL, pointer type T0 or T1	NIL, pointer type T0 or T1	BOOLEAN
	procedure type T, NIL	procedure type T, NIL	BOOLEAN
IN	integer type, 0..MAX(SET)	SET	BOOLEAN
IS	T0	type T1	BOOLEAN

Constant expressions are calculated at compile time with maximum precision (LONGINT for integer types, REAL for real types) and the result is handled like a numeric literal of the same value.

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

If a real constant x with |x| <= MAX(SHORTREAL) or x = INF is combined with a nonconstant operand of type SHORTREAL, the constant is considered a SHORTREAL and the result type is SHORTREAL.

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

Appendix B: Syntax of Component Pascal

```
Module          = MODULE ident ";" [ImportList] DeclSeq
                  [BEGIN StatementSeq]
                  [CLOSE StatementSeq] END ident ".".
ImportList      = IMPORT [ident ":="] ident {"," [ident ":="] ident} ";".
DeclSeq         = { CONST {ConstDecl ";" } | TYPE {TypeDecl ";"} |
                  VAR {VarDecl ";"}} {ProcDecl ";" | ForwardDecl ";"}.
ConstDecl       = IdentDef "=" ConstExpr.
TypeDecl        = IdentDef "=" Type.
VarDecl         = IdentList ":" Type.
ProcDecl        = PROCEDURE [Receiver] IdentDef [FormalPars] MethAttributes
                  [";" DeclSeq [BEGIN StatementSeq] END ident].
MethAttributes  = ["," NEW] ["," (ABSTRACT | EMPTY | EXTENSIBLE)].
ForwardDecl     = PROCEDURE " ^ " [Receiver] IdentDef [FormalPars] MethAttributes.
FormalPars      = "(" [FPSection {";" FPSection}] ")" [":" Type].
FPSection       = [VAR | IN | OUT] ident {"," ident} ":" Type.
Receiver        = "(" [VAR | IN] ident ":" ident ")".
Type            = Qualident
                  |  ARRAY [ConstExpr {"," ConstExpr}] OF Type
                  | [ABSTRACT | EXTENSIBLE | LIMITED]
                   RECORD ["("Qualident")"] FieldList {";" FieldList} END
                  | POINTER TO Type
                  | PROCEDURE [FormalPars].
FieldList       = [IdentList ":" Type].
StatementSeq    = Statement {";" Statement}.
Statement       = [ Designator ":=" Expr
                  | Designator ["(" [ExprList] ")"]
                  | IF Expr THEN StatementSeq
                   {ELSIF Expr THEN StatementSeq}
                   [ELSE StatementSeq] END
                  | CASE Expr OF Case {"|" Case}
                   [ELSE StatementSeq] END
                  | WHILE Expr DO StatementSeq END
                  | REPEAT StatementSeq UNTIL Expr
                  | FOR ident ":=" Expr TO Expr [BY ConstExpr]
                   DO StatementSeq END
                  | LOOP StatementSeq END
                  | WITH [ Guard DO StatementSeq ]
                   {"|" [ Guard DO StatementSeq ] }
                   [ELSE StatementSeq] END
                  | EXIT
                  | RETURN [Expr]
                  ].
Case            = [CaseLabels {"," CaseLabels} ":" StatementSeq].
CaseLabels      = ConstExpr [".." ConstExpr].
Guard           = Qualident ":" Qualident.
ConstExpr       = Expr.
```


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

```
Expr        =  SimpleExpr [Relation SimpleExpr].
SimpleExpr  =  ["+" | "-"] Term {AddOp Term}.
Term        =  Factor {MulOp Factor}.
Factor      =  Designator | 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] ")"} [ "$" ].
ExprList    =  Expr {"," Expr}.
IdentList   =  IdentDef {"," IdentDef}.
Qualident   =  [ident "."] ident.
IdentDef    =  ident [" * " | "-"].
```


Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch

Appendix C: Domains of Basic Types

Type	Domain
BOOLEAN	FALSE, TRUE
SHORTCHAR	0X .. 0FFX
CHAR	0X .. 0FFFFX
BYTE	-128 .. 127
SHORTINT	-32768 .. 32767
INTEGER	-2147483648 .. 2147483647
LONGINT	-9223372036854775808 .. 9223372036854775807
SHORTREAL	-3.4E38 .. 3.4E38, INF (32-bit IEEE format)
REAL	-1.8E308 .. 1.8E308, INF (64-bit IEEE format)
SET	set of 0 .. 31

Appendix D: Mandatory Requirements for Environment

The Component Pascal definition implicitly relies on four fundamental assumptions.

1) There exists some run-time type information that allows to check the dynamic type of an object. This is necessary to implement type tests and type guards.

2) There is no DISPOSE procedure. Memory cannot be deallocated manually, since this would introduce the safety problem of memory leaks and of dangling pointers, i.e., premature deallocation. Except for those embedded systems where no dynamic memory is used, or where it can be allocated once and never needs to be released, an automatic garbage collector is required.

3) Modules and at least their exported procedures (commands) and exported types must be retrievable dynamically. If necessary, this may cause modules to be loaded. The programming interface used to load modules or to access the mentioned meta information is not defined by the language, but the language compiler needs to preserve this information when generating code.
Except for fully linked applications where no modules will ever be added at run-time, a linking loader for modules is required. Embedded systems are important examples of applications that can be fully linked.

4) The environment must provide the ability to display Unicode characters, except for "headless" embedded applications that need no display.

An implementation that doesn't fulfil these compiler and environment requirements is not compliant with Component Pascal.

Tel +41 1 445 17 51
Fax +41 1 445 17 52
info@oberon.ch
www.oberon.ch