ETH Oberon (2019) Language Report

Felix Friedrich, Florian Negele

September 24, 2019

Abstract

This report describes the syntax and semantics of the programming language Active Oberon as it is supported by the Fox Oberon compiler by 2019. It is based on previous Oberon reports by Felix Friedrich, Jürg Gutknecht, Hanspeter Mössenböck, Florian Negele, Patrick Reali, Niklaus Wirth.

Work in Progress !

Contents

1 Syntax and Notation in this Report

We display the syntax of Active Oberon in the Extended Backus Naur Form (EBNF). We present productions (syntactic equations) as equations with a single equal sign =. On the left hand side of a production stands the defined nonterminal symbol, the right hand side contains the substitution rule and is terminated by a period. Terminal symbols are embraced by single or double quotes (for example `':='`, `"'"` and `'BEGIN'`). An alternative in a production is denoted by a vertical bar `|`. Brackets `[` and `]` denote optionality of the enclosed expression, while braces `{` and `}` denote its repetition (possibly 0 times). Additionally, parentheses `(` and `)` are used to enclose expressions and thereby control additional precedence.

The Syntax of the Oberon Language desribed herein is concluded in Section A in the appendix on page 54.

2 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 rule applies: Blanks and line breaks must not occur within symbols (except in comments and strings). They are ignored unless they are essential to separate two consecutive symbols. Capital and lower-case letters are considered as distinct.

2.1 Identifiers

Identifiers are sequences of characters, digits and special characters. The first character must be a letter:

```
Identifier = Letter {Letter | Digit | '_' }.

Letter = 'A' | 'B' | .. |'Z' | 'a' | 'b' | .. | 'z' .

Digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.
```

```
KernelLog
Abc013
Trace_me
```

Fig. 2.1: Examples of valid identifiers

2.2 Number Literals

Numbers are (unsigned) integer or float constants. The type of an integer constant is the minimal type to which the constant value belongs. The compiler represents constants with the highest available size such that in constant folding the value determines the type (and not the type of the folded arguments).

An integer number can start with a prefix that specifies its (hexadecimal or binary) representation. If a number without prefix ends with suffix `H`, the representation is hexadecimal otherwise the representation is decimal.

A real number always contains a decimal point. Optionally it may also contain a decimal scale factor. The letters `E` and `D` mean 'times ten to the power of'.

A real number is of type `FLOAT32` (and as such assignment compatible to any floating point variable) but it is represented as `FLOAT64` by the compiler. This implies that constant folding is applied with highest implemented accuracy and conversion to `FLOAT64` happens retreiving the highest possible accuracy.

```
Number     =  Integer | Real.

Integer    =  Digit {["'"]Digit} | Digit {["'"]HexDigit} 'H'
              | '0x' {["'"]HexDigit} | '0b' {["'"]BinaryDigit}.

Real       =  Digit {["'"]Digit} '.' {Digit} [ScaleFactor].

ScaleFactor = ('E' | 'D') ['+' | '-'] digit {digit}.

HexDigit   =  Digit | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
              | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' .
```

```
CONST
a = 42 ;
b = 0ABH ;
c = 13H ;
d = 0xAFFE ;
e = 0b100 ;
f = 0b1000'0010'1000 ;
g = 3. ;
h = 3.82 ;
i = 3.82E+20 ;
```

Fig. 2.2: Examples of Number Literals in constant declarations

2.2.1 Difference to original Oberon

Hexadecimal numbers of the form `0x123abc` and binary numbers of the form `0b1001` have been newly introduced.

For all numbers, the single quote sign `'` can be used as separator in numbers. Between digits, there can be not more than one `'`. A fixed distance between the separators is not enforced. This separator is ignored by the compiler.

The use of scaling character `D` in floats is deprecated. In original Oberon, it was possible to specify `REAL` literals and `LONGREAL` literals. This is considered unnecessary as constant folding is now always done with highest available precision. If necessary, a literal can be converted to `FLOAT32` with an explicit conversion.

2.3 Character Literals

Character constants are denoted by the ordinal number of the character in hexadecimal notation followed by the letter `X` or by the ASCII symbol of the character embraced by single quotes.

```
Character = Digit {HexDigit} 'X' | "'" char "'".
```

```
CONST
a = 'A' ;
b = 13X ;
```

Fig. 2.3: Examples of Character Literals in constant declarations

2.4 String Literals

Strings are sequences of characters enclosed in double (`"`) or single (`'`) quote marks. The opening quote must be the same as the closing quote and must not occur within the string. A string always contains always an invisible termination character `0X` at its end. An empty string `""` therefore consists of the character `0X`. The number of characters (including the termination character) in a string is called its length. Strings of length 2 can be used wherever a character constant is allowed and vice versa.

If double and single quotes need to be used within the string or when a multi-line string shall be entered, an escaped string format is available. A string that starts and ends with `\"` can contain line breaks and control characters such as `\n` (new line) or `\t` (tab).

```
String = '"' {Character} '"' | "'" {Character} "'" | '\"' {Character} '\"'.
```

```
CONST
a = "Hello ETH" ;
b = 'This string contains "double quotes"' ;
```

```
c = \"This is an escaped string \n with a new line character\" ;
d = \"Escaped strings
may contain new line
characters\" ;
```

Fig. 2.4: Examples of string literals in constant declarations

2.4.1 Difference to original Oberon

The escaped strings have been newly introduced. They provide a convenient way to write control characters into streams and to allow multi-line strings.

2.5 Set Literals

A set can be written in literal form as follows

```
Set     =  "{" [Element {"," Element}] "}".

Element = RangeExpression.
```

The elements of a set literal need to be constant expressions.

```
CONST
a = {1,2,16};
b = {0..10, 20};
c = {MIN(SET), MAX(SET)};
```

Fig. 2.5: Examples of set literals in constant declarations

2.6 Array Literals

Arrays can be written in literal form as follows

```
ArrayLiteral = '[' Expression {',' Expression} ']'.
```

The expressions in in an array literal need to be constant expressions. In particular they can also be array literals.

Elements of an array literal A need to be such that there is an (array base) type T such that the type t of each expression in A is assignment compatible to T. We write $t \leq T$. The type of an array literal is a static Math Array with length of A and smallest possible array base type T (i.e. T must be such that there is no T' with above compatibility and $T' \leq T$ and not $T \leq T'$).

```
CONST
A = [1,2,3]; (* ARRAY [3] OF SIGNED8 *)
B = [A, [2,5,7], [10,100,MAX(SIGNED32)]]; (* ARRAY [3,2] OF SIGNED32) *)
C = [1.0, 3, 8]; (* ARRAY [3] OF FLOAT32 *)
D = [REAL(2.0), 4, 10]; (* ARRAY [3] OF REAL *)
```

Fig. 2.6: Examples of array literals in constant declarations

2.7 Keywords, Operators and Delimiters

Operators and delimiters are the special characters, strings or reserved words listed below. The reserved words cannot be used as identifiers. The following figure lists all reserved keywords and operator symbols that are directly recognized by the scanner.

```
AWAIT BEGIN BY CONST CASE CELL CELLNET CODE DO DIV END ENUM ELSE ELSIF EXIT
EXTERN FALSE FOR FINALLY IF IGNORE IMAG IN IS IMPORT LOOP MODULE MOD NIL OF
OR OUT OPERATOR PORCEDURE PORT REPEAT RETURN SELF NEW RESULT THEN TRUE TO
TYPE UNTIL VAR WHILE WITH
ARRAY OBJECT POINTER RECORD ADDRESS SIZE ALIAS
( ) [ ] { } |
" ' , . .. : ;
& ~ ^ ?
# .# = .= < .< <= .<= > .> >= .>=
+ +* - * .* ** / ./ \ `
```

Additionally there are the following *reserved* words used for built-in procedures and types. These names are also not available as identifiers for symbols in modules.

```
ABS ADDRESS ADDRESSOF ALL ANY ASH ASSERT BOOLEAN CAP CAS CHAR CHR COMPLEX
COMPLEX32 COMPLEX64 COPY DEC DECMUL DIM ENTIER ENTIERH EXCL FIRST FLOAT32
FLOAT64 FLOOR HALT IM INC INCL INCMUL INCR INTEGER INTEGERSET LAST LEN LONG
LONGINTEGER LSH MAX MIN OBJECT ODD RANGE RE REAL RESHAPE ROL ROR ROT SET
SET8 SET16 SET32 SET64 SHL SHORT SHR SIGNED8 SIGNED16 SIGNED32 SIGNED64 SIZE
SIZEOF STEP SUM UNSIGNED8 UNSIGNED16 UNSIGNED32 UNSIGNED32 UNSIGNED64
```

Remark 1 *It should be mentioned that it is possible to change the EBNF presented in this report such that (some of) the reserved words above become keywords (appear in the EBNF) without changing the semantics of the language presented. From the viewpoint of a compiler implementer, this means that some reserved words move from the checking phase to the parsing phase of a multi-stage compiler.*

There are some more built-in procedures and types that play a special role in the Active Oberon programming language. They are bound to a special module called `SYSTEM` and do not interfere with the use of identifiers. For completeness, however, we also list them below.

```
SYSTEM.BYTE SYSTEM.GET SYSTEM.PUT SYSTEM.PUT8 SYSTEM.PUT16 SYSTEM.PUT32
SYSTEM.PUT64 SYSTEM.GET8 SYSTEM.GET16 SYSTEM.GET32 SYSTEM.GET64
SYSTEM.VAL SYSTEM.MOVE SYSTEM.REF SYSTEM.NEW SYSTEM.TYPECODE SYSTEM.HALT
SYSTEM.SIZE SYSTEM.ADR SYSTEM.MSK SYSTEM.BIT SYSTEM.Time SYSTEM.Date
SYSTEM.GetStackPointer SYSTEM.SetStackPointer SYSTEM.GetFramePointer
SYSTEM.SetFramePointer SYSTEM.GetActivity SYSTEM.SetActivity
```

Remark 2 *Built-in procedures are different from conventional procedures in that they do not necessarily conform to a particular procedure interface (i.e. a particular formal parameter list). There is no overloading concept in Oberon (besides that for Operators) implying that some of the built-in procedures cannot be implemented as conventional procedures in some separate module.*

2.8 Comments

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

```
(* This is a comment *)
MODULE Test;
CONST a (* constant symbol a *) = 3 * (* times *) 5 (* five *);
(* nested comments
  (* are possible
    "anything here is ignored, also strings"
  *)
*)
END Test.
```

Fig. 2.7: Examples of comments

Remark 3 *There is a special notation within comments for documentation purposes. These notations do not affect the meaning of the program either but are useful for automatic generation of source code documentation.*

2.9 Conditional Compilation

A program may contain arbitrary blocks of code that are conditionally compiled. Such blocks are introcuded by a `#` symbol at the beginning of a line followed by either `if`, `elseif`, or `else` according to the following syntax:

```
Block = '#' 'if' Expression 'then' Block
      { '#' 'elsif' Expression 'then' Block }
      [ '#' 'else' Block]
        '#' 'end'
      | any symbol until next new line character
```

The boolean expression may consist of identifiers and logical operators. Any identifier in such expressions is called a definition and evaluates to either `TRUE` or `FALSE` depending on whether the definition was provided to the current invocation of the compiler. The code within a conditional block is only part of the compiled program if the expression evaluates to `TRUE` and is completely ignored otherwise. Conditional blocks may be nested but must be concluded using `#end`.

3 Declaration and Scope Rules

Every identifier occurring in a program must be introduced with a declaration, unless it is a pre-declared identifier. Declarations also specify certain permanent properties of an item such as whether it is a constant, type, variable or procedure. The identifier is then used to refer to the associated item. In the following we refer to a declared identifier as a *symbol*.

Scopes are enclosing contexts where symbols can be declared and referenced. In Active Oberon, scopes can be nested. The scope of an item x is the smallest (w.r.t. nesting) block (module, procedure, record or object) in which it is declared. The item is *local* to this scope. Scope rules are

1. No identifier may denote more than one item within a given scope.
2. An item may be directly referenced within its scope only.
3. The order of declaration within a scope does not affect the meaning of a program.

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`. the identifier is then denoted as `M.x` and is called a *qualified identifier*. Identifier marked with `-` in their declaration are *read-only* in importing modules.

```
QualifiedIdentifier = Identifier ['.' Identifier].
IdentifierDefinition = Identifier [ '*' | '-' ].
```

3.1 Difference to original Oberon

The scope rules differ from the rules of the original Oberon language, a rationale is given here:

In the original Oberon-2 language report the scopes started at the declaration of an item and ended at the end of the block in which they were declared. By this construction a forwarding declaration was formally impossible which was resolved with the explicit allowance of forward pointers. With the advent of Objects in the language, heavy use was made of this implicit forward referencing together with special rules for accessing (global) variables from within objects being declared before the declaration of variables had taken place.

With a multi-stage compiler is well possible to resolve references in all directions (if and only if there are no circular dependencies that cannot be resolved). The scope rules have therefore be altered to this extend.

With the old definition, the following example code was valid

```
TYPE A = INTEGER;
PROCEDURE P;
VAR
   b:A;
   A: INTEGER;
BEGIN (* ... *)
END P;
```

while the following code was invalid:

```
TYPE A = INTEGER;
PROCEDURE P;
VAR
   A: INTEGER;
   b:A;
BEGIN (* ... *)
END P;
```

The following example was also formally invalid (but still accepted by all compilers we know of)

```
TYPE A = INTEGER;
PROCEDURE P;
VAR A:A;
BEGIN (* ... *)
END P;
```

With the new definition all three examples are invalid (and not accepted by the compiler).

4 Declaration Sequences

A declaration sequence is a sequence of constant, type, variable, procedure or operator declarations. In contrast to previous implementations of Oberon, an order of the different types of declarations is not prescribed.

```
DeclarationSequence = {
         'const' [ConstDeclaration] {';' [ConstDeclaration]}
         |'type' [TypeDeclaration] {';' [TypeDeclaration]}
         |'var'  [VariableDeclaration] {';' [VariableDeclaration]}
         | ProcedureDeclaration
         | OperatorDeclaration
         | ';'
         }
```

The different forms of declaration are described in the sequel.

```
CONST (* constant declarations *)
UARTBufLen* = 3000;
TYPE (* type declarations *)
UARTBuffer = ARRAY UARTBufLen OF SYSTEM.BYTE;
UartDesc* = RECORD (Device.DeviceDesc)
id: INTEGER;
in, out, oin, oout: SIZE;
open: BOOLEAN;
inbuffer, outbuffer: UARTBuffer
END;
Uart* = POINTER TO UartDesc;
VAR (* variable declarations *)
uarts: ARRAY Platform.NUMCOMPORTS OF Uart;

(* procedure declarations *)
PROCEDURE Close( dev: Device.Device );
BEGIN
IF dev( Uart ).open = TRUE THEN
Platform.ClearBits(Platform.UART_CR, {Platform.UARTEN});
Kernel.EnableIRQ( Platform.UartInstallIrq, FALSE );
dev( Uart ).open := FALSE;
END;
END Close;

PROCEDURE Available( dev: Device.Device ): SIZE;
```

```
BEGIN
RETURN (dev( Uart ).in − dev( Uart ).out) MOD UARTBufLen
END Available;
```

Fig. 4.1: Example of a declaration sequence

5 Modules

A module is the compilation unit of Oberon and, at the same time, a module consitutes a (singleton) object providing (global) data and code. In addition to classical Oberon module, a module can also be a template module that is parameterizable.

```
Module = 'MODULE' [TemplateParameters] Identifier ['IN' Identifier] ';'
         {ImportList} DeclarationSequence [Body]
        'END' Identifier '.'.

TemplateParameters = '(' TemplateParameter {',' TemplateParameter} ')'.

TemplateParameter = ('CONST' | 'TYPE') Identifier.

ImportList = 'IMPORT' Import { ',' Import } ';'.

Import    = Identifier [':=' Identifier] ['(' ExpressionList ')' ]
            ['IN' Identifier].
```

```
MODULE SPI; (* Raspberry Pi 2 SPI Interface —— Bitbanging *)
IMPORT Platform, Kernel;

CONST HalfClock = 100; (* microseconds —— very conservative*)

PROCEDURE SetGPIOs;
BEGIN
Platform.ClearAndSetBits(Platform.GPFSEL0, {21..29},{21,24});
Platform.ClearAndSetBits(Platform.GPFSEL1, {0..5},{0,3});
END SetGPIOs;

PROCEDURE Write* (CONST a: ARRAY OF CHAR);
VAR i: SIZE;
BEGIN
Kernel.MicroWait(HalfClock);
```

```
Platform.WriteBits(Platform.GPCLR0, SELECT); (* signal select *)
Kernel.MicroWait(HalfClock);
FOR i := 0 TO LEN(a)−1 DO
WriteByte(a[i]); (* write data, toggling the clock *)
END;
Kernel.MicroWait(HalfClock);
Platform.WriteBits(Platform.GPSET0, SELECT); (* signal deselect *)
END Write;
...

BEGIN
SetGPIOs;
END SPI;
```

Fig. 5.1: Example of a module (excerpt)

5.1 Difference to original Oberon

5.1.1 Contexts

The source code of the current A2 system consists of over a thousand modules of which one third belongs to the legacy Oberon sub-system. In order to distinguish their membership, some names of the modules belonging to the newer A2 system were prefixed by "Aos" (its previous name). Unfortunately this namingconvention has several drawbacks:

- The membership of modules with unprefixed names is not recognisable atfirst sight and confuses new users.
- existing prefixes do not reflect and even reverse the intented priority of the modules within the system.
- New modules have to be prefixed as most names are already taken by modules that belong to Oberon.As the AOS system was currently renamed to A2, modules have again to berenamed. We therefore introduced a more generic concept that avoids all of these shortcomings.

A *Context* acts as a single-level namespace for modules. It allows modules with the same name to co-exist within different contexts. Each module belongs toexactly one context. The pseudo-module SYSTEM is available in all contexts but does not belong to any of them. There are currently two contexts available for the user: Oberon and A2.

Language Extensions As modules should be able to import modules from different contexts at the same time, classifications based on a compiler-switch or different source-code filenames are not sufficient. Therefore the programmer should be able to specify thecontext of a module within its code.

The optional identifier after keyword IN specifies the name of the context a module belongs to.The context defaults to A2 if it is omitted.

We additionally have added a syntax-extension for the import section of a module: the optional context specification tells the compiler in which context to look for modules to import. This allows to use A2 modules from within Oberon and vice versa. The context defaults to the context of the module if it is omitted by the programmer.

Runtime Extensions For the execution of commands, the runtime-environment implicitly specifies the correct context. Only the modules within the same context shall be consideredwhen a command is searched for and executed. This also avoids the annoying problem of loading the complete Oberon system when some text displayed in A2 is middle-clicked accidentally.

Naming conventions The filenames of module files and their corresponding object-files are prefixed by the name of their context followed by a dot. As most of the files will belongto the default A2 context, this prefix shall be omitted for a better overview. Prefixing module files helps the programmer to be able to distinguish the membership by looking at a filename instead of having to browse its contents. The prefix for object-files is needed by the compiler and the runtime-system inorder to dynamically load the correct modules.

Simplicity The introduction of the context concept required only a few and very simple modifications of the language, compiler and the runtime-system and is fully backwards-compatible to the previous solution. It even offers a more generic solution the actual problem asked for. It could therefore even be used to assemble other big software packages like GUI applications and libraries in the longterm.

5.2 Templates

6 Constant Declarations

A constant declaration associates an identifier with a constant value. Syntactically a constant declaration consists of an identifier definition and an expression.

```
ConstantDeclaration = [IdentifierDefinition '=' ConstantExpression].

ConstanExpression = Expression.
```

Semantically the constant expression must be an expression that can be evaluated by a mere textual scan plus constant folding, without actually executing the program. Its operands are constants or predeclared functions that can be evaluated at compile time.

```
CONST
N = 320; (* constant name a associated to value 320 *)
b* = 300; (* exportet constant name b associated to value 300 *)
c* = "A string"; (* constant name c associated to a string *)
limit = 2*a-1;
fullset = {MIN(SET) .. MAX(SET)}
```

7 Variable Declarations

Variable declarations introduce variables by defining an identifier and a data type for them. Variables can be initialized with a value. If they are not initialized, the initialization with a null value is guaranteed for pointers and otherwise it depends on the implementation of the compiler.

```
VariableDeclaration = VariableNameList ':' Type.
VariableNameList = VariableName {"," VariableName}.
VariableName = IdentifierDefinition [Flags]
               [':=' Expression | 'EXTERN' Expression].
Flags = '{' [ Flag {',' Flag} ] '}'.
Flag = Identifier ['(' Expression ')' | '=' Expression].
```

Variables can be marked as `EXTERN` in which case their identifier is just an alias for a fixed memory location. This address may be specified by a constant expression or a string literal referring to an entity which is defined elsewhere. Since extern variables just refer to some other data they cannot be initialized.

```
VAR
a : REAL;
b := 10, c : INTEGER;
c* {UNTRACED} : POINTER TO ARRAY OF CHAR;
d EXTERN "BaseTypes.Pointer" : ADDRESS;
```

8 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 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.

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

```
ProcedureDeclaration = 'PROCEDURE' ['^'|'&'|'~'|'-'|Flags ['-']]
                     ['(' ParameterDeclaration ')']
                      IdentifierDefinition [FormalParameters]
                      ['EXTERN' Expression ';' | ';'
                      DeclarationSequence [Body]
                      'END' Identifier].

FormalParameters = '(' [ParameterDeclaration {';' ParameterDeclaration}] ')'
                      [':' [Flags] Type].

ParameterDeclaration = ['VAR'|'CONST'] Identifier [Flags] ['=' Expression]
           {',' Identifier [Flags] ['=' Expression]} ':' Type.

Body = 'BEGIN' [Flags] StatementSequence ['FINALLY' StatementSequence]
           | 'CODE' Code.
```

Procedures can be marked as `EXTERN` in which case their identifier is just an alias for a fixed memory location. This address may be specified by a constant expression or a string literal referring to an entity which is defined elsewhere. Since extern procedures just refer to some other code they do not have a body.

In the following some examples of a procedure declaration are shown. The last (right most) formal parameters of a procedure can be associated with a default value. The procdure can then be called with less actual parameters and the remaining formal parameters take on the default values. If a procedure declaration specifies a receiver parameter (as in the third example below), the procedure is considered to be bound to a type (here: type Student).

```
PROCEDURE Send∗(CONST data: ARRAY OF CHAR; ofs,len: SIZE;VAR res: INTEGER );
BEGIN (∗ ... ∗)
END Send;

PROCEDURE & Init∗(scanner: Scanner.Scanner; diagnostics: Diagnostics);
BEGIN (∗ ... ∗)
END Init;

PROCEDURE Float∗(x: FLOAT64; n := 4, f := 3, d := 0: INTEGER);
BEGIN
Commands.GetContext().out.FloatFix(x,n,f,d);
END Float;


PROCEDURE (CONST s: Student) GetGrade(Subject: INTEGER): REAL;
BEGIN (∗ ... ∗)
END GetGrade;
```

8.1 Operator Declaration

```
OperatorDeclaration = 'OPERATOR' [Flags] ['−'] String ['∗'|'−'] FormalParameters ';'
                        DeclarationSequence
                        [Body]
                      'END' String.
```

9 Types

The Active Oberon Language features the following classes of types:

(i) **Basic Types**
(ii) **Array Types** and **Math Array Types**
(iii) **Record Types**
(iv) **Object Types**
(v) **Pointer Types**
(vi) **Procedure Types**
(vii) **Enumeration Types**
(viii) **Port Types**, **Cell Types** and **Cellnet Types**

9.1 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, mathematical arras, records and objects) it also defines the structure of this type.

```
TypeDeclaration = IdentifierDefinition '=' Type ';'.
Type = ArrayType | MathArrayType | RecordType | PointerType | ObjectType
       | ProcedureType | EnumerationType | QualifiedIdentifier
       | CellType | CellnetType | PortType.
```

```
TYPE
  Count = UNSIGNE64 ;
  Pair = RECORD
    first, second: Count;
  END;
```

9.2 Categories of Types

9.2.1 Basic Types

Basic types are predefined by the Oberon language and can be addressed by the corresponding predefined identifiers. There are types that grow or shrink with the target hardware (in contrast to the original language, cf. **??**) and types that have a fixed sized representation.

The predeclared basic types of Active Oberon are shown with names and valid values in Table 1.

Some of the types are represented with a **fixed size** that does not depend on the target hardware. With the exception of `CHAR` and `BOOLEAN`, the fixed bit width is expressed as a bit-width suffix at the type (e.g. `SET16` provides 16 bits).

There is another type with fixed width, declared in (pseudo-)module SYSTEM, the *a Byte type* `SYSTEM.BYTE`. Moreover, strictly speaking there is *a String type* that is implicitly associated with string literals and not available as explicit type in declarations.

The other types grow and shrink with the target hardware.

Figure 9.1 shows the (implicit) compatibility of the integer types. An arrow from A to B means: a variable of type A can be assigned to a variable of type B.

The range of the larger type includes the ranges of the smaller types. The smaller type is said to be *compatible* with the larger one in the sense that it can without danger of loss

Type name	Size	Valid values
`BOOLEAN`	1 byte	`TRUE` or `FALSE`
`CHAR`	1 byte	characters of the extended ASCII set (`0X` ... `0FFX`)
`REAL`		default floating point type, corresponds to double in C
`INTEGER`	machine word	signed integers in machine word size, corresponds to `int` in C
`ADDRESS`	address width	unsigned integers in address range
`SIZE`	address width	signed integers in address range
`SET`	address width	set with address width
`SIGNED8`	1 byte	integers between -2^7 and $2^7 - 1$
`SIGNED16`	2 bytes	integers between -2^{15} and $2^{15} - 1$
`SIGNED32`	4 bytes	integers between -2^{31} and $2^{31} - 1$
`SIGNED64`	8 bytes	integers between -2^{63} and $2^{63} - 1$
`UNSIGNED8`	1 byte	integers between 0 and $2^8 - 1$
`UNSIGNED16`	2 bytes	integers between 0 and $2^{16} - 1$
`UNSIGNED32`	4 bytes	integers between 0 and $2^{32} - 1$
`UNSIGNED64`	8 bytes	integers between 0 and $2^{64} - 1$
`FLOAT32`	4 bytes	floating point value between -3.4028^{38} and $+3.4028^{38}$
`FLOAT64`	8 bytes	floating point value between -1.7976^{308} and $+1.7976^{308}$
`SET8`	1 byte	any set combination of the integer values between 0 and 7
`SET16`	2 bytes	any set combination of the integer values between 0 and 15
`SET32`	4 bytes	any set combination of the integer values between 0 and 31
`SET64`	8 bytes	any set combination of the integer values between 0 and 63

Table 1: Predeclared Basic Types

of leading digits be converted. In assignments and in expansions the conversion of internal representations is automatic.

Unsigned integers are compatible with signed or unsigned integer of same or smaller size. This implies that the assignment from a signed to an unsigned integer of same size is ok. The other direction does not work:

$$\begin{tikzcd}
\texttt{SIGNED8} \arrow[r] \arrow[d] & \texttt{SIGNED16} \arrow[r] \arrow[d] & \texttt{SIGNED32} \arrow[r] \arrow[d] & \texttt{SIGNED64} \arrow[d] \\
\texttt{UNSIGNED8} \arrow[r] & \texttt{UNSIGNED16} \arrow[r] & \texttt{UNSIGNED32} \arrow[r] & \texttt{UNSIGNED64}
\end{tikzcd}$$

Fig. 9.1: Integer Compatibilities

Although the `SIZE` type is signed and `ADDRESS` is unsigned, `SIZE` and `ADDRESS` types are assignment compatible in both directions.

Moreover, integer types are compatible to floating point types, i.e. any integer type can be assigned to `FLOAT32` or `FLOAT64` and `FLOAT32` is compatible to `FLOAT64`.

Where there is no implicit compatibility between types, they can be converted with an explicit conversion. The type name itself can be used for a type conversion.

```
VAR
s8: SIGNED8; s16: SIGNED16; s64: SIGNED64;
u8: UNSIGNED8; u16: UNSIGNED16; u64: UNSIGNED64;
adr: ADDRESS; size: SIZE;
BEGIN
s16 := s8; (* ok *)
u16 := s8; (* ok *)
s16 := u8; (* ok *)
u16 := s16; (* ok *)
adr := size; (* ok *)
size := adr; (* ok *)

s16 := u16; (* error *)
s16 := SIGNED16(u16); (* ok *)
```

9.2.2 Difference to original Oberon

The original Oberon fundamental types comprised four integer types **`SHORTINT`**, **`INTEGER`**, **`LONGINT`** and **`HUGEINT`**.

In the early days of Oberon, it was believed by many developers that the type sizes would grow with the hardware. Effectively, however, the types were fixed to sizes of 8, 16, 32 and 64 bit because a substantial amount of libraries had made assumptions on the implemented type sizes and changing the sizes would have have broken them.

Unfortunately, types that express hardware-dependent properties, such as the address width, were not included, which made it hard to port Oberon to, for example, 64-bit architectures. Already addresses in the higher 2G of 32-bit systems made problems because they were represented with `LONGINT`, a signed 32-bit integer type.

We decided to make a radical step and to abandon the old types names completely and to introduce types with type-names that clearly document that they are either bound to a certain bit-width or to features of the hardware.

We introduced unsigned integer types because they can come handy and because they behave different for fundamental operations such as shifts or comparisons.

When to use `SIZE` The type `SIZE` is the signed analogon of type `ADDRESS`. While type `ADDRESS` is primarily designed for low-level programming, type `SIZE` is of high relevance in all kinds of programs.

`SIZE` must be used when any kind of memory size or interval is (implicitly) addressed. This implies the use for the lenght of an array, iterating or counting array elements but also iterating or counting elements in other dynamic data structures.

When to use `INTEGER` The type `INTEGER` represents the word-size of the underlying architecture. As such, no assumptions on the bit-width of this type should be made. There are platforms with quite some difference between the address width and the optimal width for integer computation. For AMD64, for example, the machine word size is defined as 32-bit.

A programmer usually needs to pay attention to some aspects of the internal representation of a type, even if it is only a certain intuition about the types that he or she is using.

A programmer can hope that the word size of a machine matches the typical application domain ("is useful") for generic integers that do not constitute addresses or address differences. This is the case for "int" in C and it should and probably will be for `INTEGER` in this dialect of Oberon.

However, we think that the use of `INTEGER` is quite restricted. It is certainly useful in education, for rapid prototyping or for any case where the programmer can expect that the result of a computation will be reasonably small for the typical application domain on a given machine. In all other cases, a programmer needs to use a type with size guarantees (e.g. `SIGNED64`), the type `SIZE`or use a declared type to be flexible.

Use Type Declarations We generally believe that in the same way as it is good practice to use meaningful variable names, the use of declared types with meaningful type names (such as, "Velocity" or "Amount" or "Bitwidth") provides a good way to document the intended purpose of a type.

9.3 Arrays

Arrays can be declared in the following form:

```
ArrayType = 'ARRAY' [Expression {',' Expression}] 'OF' Type.
```

There are thus two kinds of arrays possible:

(a) *Static Arrays* being declared as `ARRAY` x `OF` type, where x must be a constant expression,

(b) *Open Arrays* being declared as `ARRAY OF` type.

The expression `ARRAY` x,y `OF` type is an abbreviatory notation for `ARRAY` x `OF ARRAY` y `OF` type.

Semantic rules:

- Static arrays of open arrays are not permitted.
- Arrays of mathematical arrays are not permitted.
- A length expression `x` in `ARRAY x OF` type must be a constant, positive integer or zero

```
TYPE
  Vector = ARRAY 4 OF REAL;
  Matrix = ARRAY 4,4 OF REAL;
VAR
  buffer: ARRAY 16 OF SIZE;

PROCEDURE Print(CONST x: ARRAY OF CHAR)
```

9.4 Math Arrays

Special mathematical types have been added to the Oberon language recently. They can be declared as in

```
MathArrayType = 'ARRAY' '[' MathArraySize {',' MathArraySize} ']' 'OF' Type.
MathArraySize = Expression | '*' | '?'.
```

There are three forms of mathematical arrays possible

(a) *Static Mathematical Arrays* being declared as `ARRAY` [x] `OF` type, where x must be a constant,

(b) *Open Mathematical Arrays* being declared as `ARRAY` [$*$] `OF` type,

(c) *Tensors* being declared as `ARRAY` [?] `OF` type, Again, the expression `ARRAY [x,y] OF` type is an abbreviatory notation for `ARRAY [x] OF ARRAY [y] OF` type.

Semantic rules:

- Mathematical arrays of (conventional) arrays are not permitted.
- Arrays of Tensors and Tensors of Arrays are not permitted.
- Static Mathematical Arrays of Open Mathematical Arrays or Static Mathematical Arrays Tensors are not permitted.
- A length expression `x` in `ARRAY [x] OF` type must be a constant, positive integer or zero

```
VAR
  x: ARRAY [*] OF REAL;
  vec: ARRAY [4] OF REAL;
  tensor: ARRAY [?] OF FLOAT32;
```

9.5 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. 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' ['(' QualifiedIdentifier ')']
           [VariableDeclaration {';' VariableDeclaration}]
           {ProcedureDeclaration [';']| OperatorDeclaration [';']}
           'END'.
```

```
TYPE
   Date = RECORD
      day*, month*, year*: INTEGER
   END

VAR
   x: RECORD
      name, firstname: ARRAY 32 OF CHAR;
      age: INTEGER;
```

```
        salary: REAL
    END
```

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. 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).

Semantic rules

- The base type `T0` of a record `T1` must be a record if the record is defined in the form `T1 = RECORD (T0) ... END`.
- If a type `T1` is defined as pointer to a record in the form `T1 = POINTER TO RECORD (T0) ... END`, then T0 may be a record or a pointer to a record.

Records can also contain procedures that operate on the record fields by ways of a hidden SELF variable parameter.

```
TYPE
    Date = RECORD
        day, month, year: INTEGER;

        PROCEDURE Reset();
        BEGIN
          day := 1; month := 1; year := 2020;
        END Reset;
    END
```

9.6 Pointer Types

Formally, pointers can be defined in the form

```
PointerType = 'POINTER' [Flags] 'TO' Type.
```

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 inherit 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.

There are thus actually only two kinds of pointers possible in the Active Oberon language:

(a) *Pointer to array* being declared as `POINTER TO` array type

(b) *Pointer to record* being declared as `POINTER TO` record type

If p is a variable of type P = POINTER TO T, a call of the predeclared procedure NEW(p) 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, e0, ..., en-1) where T is allocated with lengths given by the expressions e0, ..., en-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 pointer variables inherit the extension relation of the basetype `ANY` and are initialized to NIL.

For systems programming, the Oberon language discussed herein contains unsafe pointers. An unsafe pointer is assignment compatible to type `ADDRESS` and pointer arithmetics are allowed.

```
CONST
  GPIO = 03F200000H;
VAR
  gpio*: POINTER {UNSAFE} TO RECORD
    GPFSEL: ARRAY 6 OF SET32;
    reserved: ADDRESS;
    GPFSET: ARRAY 2 OF SET32;
    GPFCLR: ARRAY 2 OF SET32;
  END;

BEGIN
  gpio := GPIO;
```

9.7 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 of `P` and `T` must match. `P` must not be local to another procedure. If `P` is a type bound procedure, then `T` must be flagged as delegate.

```
ProcedureType = 'PROCEDURE' [Flags] [FormalParameters].
```

```
TYPE
 Sender* = PROCEDURE {DELEGATE} ( CONST buf: ARRAY OF CHAR; ofs, len: SIZE);
VAR
 Available*: PROCEDURE ( dev: Device ): LONGINT;
```

9.8 Object Types

Objects are basically *reference* records that can be equipped with procedures. Procedures in an object are methods: they reside in the object scope and have access to the object's variables. An object can be explicitly referred to in its method using the SELF identifier.

A method prefixed by an ampersand character & is an *object initializer*. This method is automatically called when an instance of the object is created and processed before the object becomes publicly available. An object may have at most one initializer. If absent, the initializer of the base type is inherited. Initializers can be called like methods.

```
ObjectType = 'OBJECT'
          | 'OBJECT' [Flags] ['(' QualifiedIdentifier ')']
              DeclarationSequence
              [Body]
            'END' [Identifier].
```

9.9 Enumeration Types

An `ENUM` type declares a set of scoped constant values called enumerators. The use of enumeration types provides for type safety by ensuring that invalid values cannot be used for any variable or parameter of an enumeration type involving operations on variables of that type.

The type of an enumerator is the containing enumeration which supports assignment and all ordering relations. An enumeration can also be extended in which case variables of this type and all of its enumerators become compatible to extending enumerations. In order to access an enumerator, its name has to be qualified by the name of an enumeration type definition.

Each enumerator has an ordinal value which can be explicitly specified using an arbitrary constant integer expression. If omitted, the ordinal value of an enumerator corresponds to the value of its immediate predecessor incremented by one. The implicit ordinal value of the first enumerator is either zero or the biggest ordinal value of all extended enumerations incremented by one. The actual value of an enumerator or enumeration variable can be obtained by using the `ORD` operation which yields the smallest integer type capable of representing all ordinal values of the corresponding enumeration.

Individual identifiers of an ENUM type list can be exported by marking them with $*$.

```
EnumerationType = 'ENUM' ['('QualifiedIdentifier')']
                   IdentifierDefinition ['=' Expression]
                   {',' IdentifierDefinition ['=' Expression]}
                 'END'.
```

An `ENUM` type may be defined as an extension of an existing ENUM type declaration by including identifier of the base type in the type definition of the extending type. All enumerated values of the base type become valid values of the new type. But note that the base type is only downwards compatible with any extended types derived from it, extensions are not upwards compatible with their base type. This restriction exists because any value of the base type is always a legal value of any extension type derived from it, however not every value of an extension type is also a valid value of the base type.

Examples Suppose that a variable of enumeration type is exported from a module

```
MODULE Graphics;
TYPE
  Monochrome* = ENUM
    black*, white* (* ORD(black) has the value 0*)
  END;
VAR
  pixel* : Monochrome;         (* pixel is exported *)
```

After importing the variable it can be used

```
MODULE Application;
IMPORT Graphics;
VAR pixel: Graphics.Monochrome;
BEGIN
  pixel := Monochrome.white; (* qualified *)
```

And the enumeration type can be extended, based on the original type

```
TYPE
 Monochrome = Graphics.Monochrome;
 ColourRGB = ENUM (Monochrome)
  red, blue, green
 END;
 ColourCYM = ENUM (Monochrome)
  cyan, yellow, magenta
 END;
```

```
VAR a:Monochrome; b,d: ColourRGB, c: ColourCYM;
  BEGIN
    (* the following are valid, compatible for assignment *)
    a:= ColourRGB.white;
    b:= ColourRGB.blue;
    c:= ColourRGB.yellow;
    b:= a (*valid — value of b is now white *);
    d:= b (*valid — value of d is now blue *);
    (* the following are invalid due to type mismatch *)
    a:= b (*invalid*)
    b:= c (*invalid*)
```

9.9.1 Comparison to original Oberon

The original Oberon did not feature enumeration types at all and they were added to the language. Two main objections have previously been levelled against enumeration types: Potential ambiguity of naming when importing an enumeration type from another module and lack of type extensibility. To provide a simple solution to the potential ambiguity problem all enumeration identifiers are qualified with the identifier of the type. The maximum number of identifiers in an enumeration and the value that can be assigned to them is implementation dependent. In a language without enumeration types, or with “quasi” enumeration types programmers must manually check that values are not out of range, but for large programs this becomes practically impossible, even for small programs it is difficult. For example the source of the Oberon System is littered with groups of CONST declarations which provide a typeless and error prone substitute for enumeration types.

9.10 Active Cells: Cell Types, Cellnet Types and Port Types

```
CellType = ('CELL' | 'CELLNET') [Flags] [PortList] [';'] {ImportList}
             DeclarationSequence
           [Body] 'END' [Identifier].

PortList = [PortDeclaration {';' PortDeclaration}].

PortDeclaration = Identifier [Flags] {',' Identifier [Flags]}':' PortType.

PortType = 'PORT' ('IN'|'OUT') ['(' Expression ')'].
```

10 Expressions

10.1 Ingredients of an Expression

Expressions are of the following form:

```
Expression = RangeExpression [RelationOp RangeExpression].
RelationOp = '=' | '#' | '<' | '<=' | '>' | '>=' | 'IN' | 'IS'
           | '.=' | '.#' | '.<' | '.<=' | '.>' '.>='
           | '??' | '!!' | '<<?' | '>>?'.
```

The operators in the second and third line of RelationOp are defined specifically for the Math Arrays and Active Cells subset of the language, respectively.

Range Expressions, Simple Expressions and Terms are defined as follows

```
RangeExpression = SimpleExpression
         | [SimpleExpression] '..' [SimpleExpression]['by' SimpleExpression]
         | '*' .

SimpleExpression = ['+'|'-'] Term {AddOp Term}.
AddOp            = '+' | '-' | 'or'.

Term  = Factor {MulOp Factor}.
MulOp = '*' | '/' | 'div' | 'mod' | '&'
      | '.*' | './' | '\' | '**' | '+*' .
```

The operators defined in the second line of MulOp are defined specifically for Math Arrays.

A Factor is defined as

```
Factor = Number [Guard]
         | Character | String | Set | ArrayLiteral
         | Designator
         | '(' Expression ')' [Guard]
         | '~' Factor | Factor '`'
         | 'NIL' | 'IMAG' | 'TRUE' | 'FALSE' |
         | 'SIZE' 'OF' Designator | 'ADDRESS' 'OF' Designator
         | 'ALIAS' OF Expression

Guard = "(" ExpressionList ")".

ExpressionList = Expression { ',' Expression }.
```

The suffix operator "`" is defined specifically for Math Arrays.

todo : Check the Guard. In the compiler it is DesignatorOperations but I am not sure if all of it is applicable / wanted?

10.1.1 Difference to original Oberon

The notion of Range Expressions has been elevated from the Set type to Expressions in order to represent slices in Math Oberon.

Array literals have been added to the language.

A type guard on numbers has been introduced. A guarded number is converted to the given type if and only if its value is not changed. If this fails, a trap is raised.

Complex numbers have beend added to the language. Therefore the literal 'IMAG' has been introduced.

Because `SIZE` and `ADDRESS` are types, the meaning of **`SIZE(x)`** and **`ADDRESS(y)`** (formally **`SYSTEM.SIZE(x)`** and **`SYSTEM.ADDRESS(x)`** has changed. Therefore, the forms **`ADDRESS OF`** and **`SIZE OF`** have been introduced.

Similarly **`ALIAS OF`** has been introduced for Math Arrays.

Various operators, including the suffix transpose operator "'" have been introduced for Math Arrays.

10.2 Designators

```
Designator    = DesignatorName [DesignatorOperations] [Flags].

DesignatorName = 'SELF' | 'RESULT' | 'ADDRESS' | 'SIZE' | Identifier
               | 'NEW' QualifiedIdentifier '(' ExpressionList ')'.

DesignatorOperations = { "(" [ExpressionList] ")"
                       | "." Identifier
                       | '[' ExpressionList ']'
                       | '^'
                      } .

IndexList = '?' [',' ExpressionList]
          |  ExpressionList [',' '?' [',' ExpressionList] ]
```

```
a[10]
P(3,5).GetArray[5].CallMe()
```

```
myVariable(Type)
```

Fig. 10.1: Example of Designators

10.2.1 Difference to original Oberon

The **RESULT** designator can be used to access the (implicit) return parameter of a procedure return parameter. This feature was implemented in order to reduce memory pressure in avoiding reallocations of already allocated return parameters. The statement **RETURN RESULT;** only returns from a procedure without writing the result. The latter can also be used when the return value of a procedure has already been written in inline assembly code.

```
PROCEDURE ReturnLargeArray(): ARRAY[*] OF REAL;
BEGIN
IF LEN(RESULT) < LargeSize THEN
NEW(RESULT, LargeSize)
END;
...
RETURN RESULT;
END ReturnLargeArray;
```

In contrast to the original **NEW** statement, the designator form **NEW Type(parameters)** has been introduced in order to be able to allocate a type and assign it to a base type at the same type.

```
VAR
x: Expression;
BEGIN
x := NEW BinaryExpression(left, right);
```

Most of the restrictions to designators were removed. For example, designators can be arbitrarily chained and references returned from procedure calls can be passed to const parameters.

```
P(3)[10].p(Q());
```

Fig. 10.2: Example of a syntactially valid designator

11 Statements

```
Statement = [
  Designator
    [':=' Expression
    | '!' Expression | '?' Expression | '<<' Expresssion | '>>' Expression
    ]
  | 'IF' Expression 'THEN' StatementSequence
    {'ELSIF' Expression 'THEN' StatementSequence}
    ['ELSE' StatementSequence]
    'END'
  | 'WITH' Identifier ':' QualifiedIdentifier 'DO' StatementSequence
    {'|' QualifiedIdentifier 'DO' StatementSequence}
    [ELSE StatementSequence]
    'END'
  | 'CASE' Expression 'OF' ['|'] Case
    {'|' Case}
    ['ELSE' StatementSequence]
    'END'
  | 'WHILE' Expression 'DO'
      StatementSequence
    'END'
  | 'REPEAT'
      StatementSequence
    'UNTIL' Expression
  | 'FOR' Identifier ':=' Expression 'TO' Expression ['BY' Expression] 'DO'
      StatementSequence
    'END'
  | 'LOOP' StatementSequence 'END'
  | 'EXIT'
  | 'RETURN' [Expression]
  | 'AWAIT' Expression
  | StatementBlock
  | 'CODE' {any} 'END'
  | 'IGNORE' Designator
].

Case = RangeExpression {',' RangeExpression} ':' StatementSequence.

StatementBlock = 'BEGIN' [Flags] StatementSequence 'END'.

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

11.1 Statement Block and Statement Sequences

Statements may be grouped into a block delimited by BEGIN and END. The block may include modifiers in braces `{}` which modify the properties of actions within the block. See individual statement types for details of their modifiers. A sequence of more than one statement denotes the sequence of actions specified by the component statements, they are delimited by semicolons.

```
StatementBlock = 'BEGIN' [Flags] StatementSequence 'END'.
StatementSequence = Statement {';' Statement}.
```

```
BEGIN{EXCLUSIVE} (* an exclusive statement block *)
  state := States.normal;
  AWAIT(state = States.alert)
END;
```

11.2 Assignment Statement

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

```
 Designator ':=' Expression
```

The type of the designator must be assignment compatible with the type of the expression.

```
  i := 0;
  x := 3.4;
  y := i*i;
```

Fig. 11.1: Examples of Assignments

11.3 Procedure Call Statement

A procedure call serves to activate a procedure. The procedure call may contain a list of actual parameters which are substituted in place of their corresponding formal parameters defined in the procedure declaration. The correspondence is established by the relative positions of the parameters in the lists of actual and formal parameters respectively.

```
Designator ["(" ExpressionList ")"];
```

There are three kinds of parameters: value, constant and variable parameters. In the case of variable parameters, the actual parameter must be a designator representing an address. If it designates an element of a structured variable, the selector is evaluated when the formal/actual parameter substitution takes place, i.e. before the execution of the procedure.

If the parameter is a value parameter, the corresponding actual parameter must be an expression. This expression is evaluated prior to the procedure activation, and the resulting value is assigned to the formal parameter which now constitutes a local variable on the callee side.

If the parameter is a constant parameter, the corresponding actual parameter can be copied but it does not have to. The compiler can optimize. This is particularly handy for strings and large records.

```
PROCEDURE Test(a: SIGNED32; VAR r: REAL;
               CONST c: Student; CONST s: ARRAY OF CHAR);
BEGIN
  IF c.semester = 2 THEN (* c might be passed by reference here *)
    a := a * 2; (* no effect to caller *)
    r := 20; (* effect to caller *)
    TRACE(s);
  END;
END Test;

VAR r: REAL;
Test(22, r, student, "Rabbit");
```

11.3.1 Difference to original Oberon

We have introduced CONST parameters in order to support the read-only access to large arrays in the MathArray framework. It turned out to be very useful for records and strings also.

11.4 Communication Statement

The various communication statements are used in the Active Cells subset of the language. They are used to send and receive data via a port in a blocking or non-blocking way.

11.5 If-Elsif-Else-End Statement

An IF statement specifies the conditional execution of statements depending on the evaluation of a a Boolean expression called its guard. The guards are evaluated in sequence of occurrence, until one evaluates to TRUE, thereafter its associated statement sequence is executed. If no guard is satisfied, the statement sequence following the symbol ELSE is executed, if there is one.

```
'IF' Expression 'THEN' StatementSequence
   {'ELSIF' Expression 'THEN' StatementSequence}
   ['ELSE' StatementSequence]
   'END'
```

```
IF ch <= "9" THEN RETURN ORD( ch ) - ORD( "0" )
ELSIF ch <= "F" THEN RETURN ORD( ch ) - ORD( "A" ) + 10
ELSIF ch <= "f" THEN RETURN ORD( ch ) - ORD( "a" ) + 10
ELSE Error( Basic.NumberIllegalCharacter ); RETURN 0
END
```

11.6 Case Statement

A CASE statement specifies the selection and execution of a statement sequence according to the value of an expression. First the case expression is evaluated, then the statement sequence is executed whose case label list contains the obtained value. The case expression and all labels must be of the same type, which must be an enumeration type, integer type, any `SET` type or `CHAR`. Case labels are constants, and no value must occur more than once in a single `CASE` statement. 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. If, in this situation, there is no `ELSE` part, a trap is raised.

```
'CASE' Expression 'OF' ['|'] Case
   {'|' Case}
   ['ELSE' StatementSequence]
   'END'
```

```
CASE ch OF
  EOT: s := EndOfText
  | '#': s := Unequal; GetNextCharacter
  | '&': s := And; GetNextCharacter
  | '[': s := LeftBracket; GetNextCharacter
  | ']': s := RightBracket; GetNextCharacter
  (* ... *)
ELSE
  s := Identifier; GetIdentifier( token );
END;
```

11.7 With Statement

When working with a variables p of dynamic type (e.g. pointers to records or reference parameters of record type), often it is required to guard p to a type that extends the static type of p. The WITH statement assumes a role similar to the type guard, extending the guard over an entire statement sequence. It may be regarded as a regional type guard.

Moreover, the `WITH` statement allows to check for a type and conditionally execute a statement sequence depending of the dynamic type. A `WITH` statement can, in this sense, be regarded like a `CASE` statement for types.

If the dynamic type of the guarded variable is not of dynamic type of any of the alternatives given, the `ELSE` branch is taken. If no `ELSE` branch is present in such cases, a trap will be raised.

```
'WITH' Identifier ':' QualifiedIdentifier 'DO' StatementSequence
    {'|' QualifiedIdentifier 'DO' StatementSequence}
    [ELSE StatementSequence]
    'END'
```

If a type T_1 is referred to by a qualified identifier in the `WITH` statement and a type T_2 is referred to by a later qualified identifier later in the same `WITH`, T_2 may not extend T_1 (otherwise the branch with T_2 would never be reachable).

```
WITH x:
| SyntaxTree.ResultDesignator DO result := ResolveResultDesignator(x)
| SyntaxTree.SelfDesignator DO result := ResolveSelfDesignator(x)
| SyntaxTree.BinaryExpression DO result := ResolveBinaryExpression(x)
| SyntaxTree.UnaryExpression DO result := ResolveUnaryExpression(x)
END;
```

Fig. 11.2: Example of a WITH statement

```
WITH x:
| SyntaxTree.Expression DO (* general case *)
| SyntaxTree.UnaryExpression DO (* forbidden ! *)
END;
```

Fig. 11.3: Example of a rejected WITH statement when UnaryExpression inherits from Expression

11.7.1 Difference to original Oberon

In Oberon-2 the `WITH` statement with alterantives was present buth then it was removed again. We reintroduced it in a slighty modified form. We removed the necessity to repeat the variable name for each case occuring.

11.8 While Statement

A `WHILE` statement specifies repetition zero or more times of some statements. If the Boolean expression (guard) yields `TRUE`, then the statement sequence is executed. The expression evaluation and the statement execution are repeated as long as the Boolean expression yields `TRUE`.

```
'WHILE' Expression 'DO'
   StatementSequence
'END'
```

```
WHILE len > 0 DO
  data[length] := buf[ofs];
  INC(ofs); INC(length); DEC(len)
END;
```

11.9 Repeat-Until Statement

A `REPEAT` statement specifies the repeated execution of a statement sequence until the Boolean expression (guard) yields TRUE. The statement sequence is thus executed one or more times.

```
'REPEAT'
   StatementSequence
```

```
'UNTIL' Expression
```

```
REPEAT
  expression := Expression();
  expressionList.AddExpression( expression )
UNTIL ~Optional( Scanner.Comma );
```

11.10 For Statement

The `FOR` statement provides a means of repeating a sequence of statements for a number of times whilst automatically incrementing or decrementing a variable by a fixed constant value. The loop continues whilst the value of the `FOR` variable (counter) is within the range specified by the two expressions.

It is mainly used in arithmetic algorithms where the counter may typically be used as an array index.

```
'FOR' Identifier ':=' Expression 'TO' Expression ['BY' Expression] 'DO'
   StatementSequence
'END'
```

The for statement is equivalent to a while statement with a additional temporary variable for the end value. The start and end value of the for-loop are only evaluated once. The increment of a for-loop must be constant expression.

```
FOR i := start TO end BY increment DO
  Statements
END;
```

is equivalent to

```
temp := end;
is := start;
WHILE i != end DO
  Statements;
  i := i + increment;
END;
```

```
FOR i := 0 TO EndOfText DO ASSERT(symbols[i] # "") END;
```

11.11 Loop and Exit Statement

A `LOOP` statement specifies the repeated execution of a statement sequence, the loop is terminated by the execution of any `EXIT` statement within that sequence.

An EXIT statement consists of the symbol EXIT. It specifies termination of the enclosing loop statement and continuation with the statement following the END of that loop statement. Each Exit statement is contextually, although not syntactically bound to the loop statement which contains it.

```
'LOOP' StatementSequence 'END'
```

The use of `WHILE` and `REPEAT` statements is recommended for the most cases. A loop statement can be useful to express repetition where there are several termination conditions at different points in the code.

```
LOOP
  IF ("0" <= ch) & (ch <= "9") OR (d = 0) & ("A" <= ch) & (ch <= "F") THEN
    dig[n] := ch; INC( n ) END;
  ELSIF ch = "." THEN
   m := n;
  ELSE EXIT
  END
END;
```

11.12 Return Statement

A `RETURN` statement is used to return from a procedure. If the procedure is declared to return a value of type T, the return statement must return an expression of assignment compatible type.

```
PROCEDURE Ten( e: SIGNED32 ): FLOAT64;
VAR x, p: FLOAT64;
BEGIN
  x := 1; p := 10;
  WHILE e > 0 DO
```

```
    IF ODD( e ) THEN x := x * p END;
    e := e DIV 2;
    IF e > 0 THEN p := p * p END (* prevent overflow *)
  END;
  RETURN x
END Ten;
```

11.13 Await Statement

The `AWAIT` statement is a statement to synchronize runnning processes (threads).

```
'AWAIT' Expression
```

11.14 Code Block

Code Blocks can be used in order to write inline assembly code in Oberon.

```
'CODE' {any} 'END'
```

```
OPERATOR -"-"*(x {REGISTER}: Vector): Vector;
VAR res{REGISTER}: Vector;
BEGIN
  CODE
    XORPS res, res
    SUBPS res, x
  END;
  RETURN res;
END "-";
```

11.15 Ignore Statement

The `IGNORE` statement can be used in order to ignore the result of a procedure call.

```
'IGNORE' Designator
```

The IGNORE statement was introduced for interfacing with C libraries, where often the result of a library call is ignored. It was not present in the original Oberon.

```
IGNORE User32.SetWindowText(root.hWnd, windowTitle);
IGNORE User32.BringWindowToTop( root.hWnd );
IGNORE User32.SetForegroundWindow( root.hWnd );
```

It turned out to be also convenient for supporting passing on results in chained stream-expressions, as they are, for example, available in C++.

```
IGNORE Out.GetWriter() << "This is text " << "that is concatenated";
```

12 Built-in Functions and Symbols

There are some built-in procedures and functions in Active Oberon. We give a short overview in the following table. Note that Integer stands for SIGNEDx or UNSIGNEDx, SIZE or INTEGER.Float stands for any of FLOATx or REAL. Number stands for Integers or Float. Set stands for any of SETx or SET. Complex stands for any of COMPLEXx or COMPLEX.

12.1 Global

12.1.1 Conversions

Function	Argument Types	Result Type	Description
ABS(x)	x: Number	Number	return absolute value of x
CAP(x)	x: CHAR	CHAR	return capital letter of x
CHR(x)	x: Integer	CHAR	return character with ascii-number x
ENTIER(x)	x: Float	SIGNED32	return largest integer not greater than x
ENTIERH(x)	x: Float	SIGNED64	return largest integer not greater than x. DEPRECATED
FIRST(r)	r: RANGE	SIZE	return first element of range
IM(x)	x: Complex	Float	return imaginary part of c
LAST(r)	r: RANGE	SIZE	return last element of range

Function	Argument Types	Result Type	Description
LONG(x)	x: Number	Number	number conversion up DEPRECATED
ODD(x)	x: Integer	BOOLEAN	return if least significant bit of x is set
ORD(x)	x: CHAR	SIGNED16	return ascii-number of x
RE(c)	c: Complex	Float	return real part of c
SHORT(x)	x: Number	Number	number conversion down DEPRECATED
STEP(r)	r: RANGE	SIZE	return step size of range

The deprecated number conversion routines SHORT and LONG operate with respect to the relations

$$FLOAT64 \supset FLOAT \text{ and } SIGNED46 \supset SIGNED32 \supset SIGNED16 \supset SIGNED8.$$

All type names of numeric types can also be used for conversion. The deprecated `ENTIERH`(x) can be replaced by `SIGNED64`(x).

12.1.2 Arithmetics

Function	Argument Types	Result Type	Description
DEC(x)	x: Integer		decrement x by 1
DEC(x,n)	x: Integer, n: Integer		decrement x by n
EXCL(s,e)	s: SET, e: Integer		exclude element e from set s
INC(x)	x: Integer		increment x by 1
INC(x,n)	x: Integer, n: Integer		increment x by n
INCL(s,e)	s: SET, e: Integer		include element e in set s
MAX(T)	Number or Set type T	Number	return maximal number of basic type t
MIN(T)	Number or Set type T	Number	return minimal number of basic type t

12.1.3 Shifts

Function	Argument Types	Result Type	Description
ASH(x,y)	x: Integer or Set, y: Integer	Integer or Set	return arithmetic shift of x by y bits (shifts right for n < 0)
LSH(x,n)	x: Integer or Set, y: Integer	Integer or Set	Returns value x logically shifted left n bits (shifts right for n < 0)
ROL(x,y)	x: Integer or Set, y: Integer	Integer or Set	return rotate left of x by y bits.
ROR(x,y)	x: Integer or Set, y: Integer	Integer or Set	return rotate right of x by y bits.
ROT(x,n)	x: Integer or Set, y: Integer	Integer or Set	Returns value x rotated left by n bits (rotates right for n < 0)
SHL(x,y)	x: Integer or Set, y: Integer	Integer or Set	return shift left of x by y bits.
SHR(x,y)	x: Integer or Set, y: Integer	Integer or Set	return shift right of x by y bits. Type of x determines if this is logical or artihmetic shift

12.1.4 Arrays and Math Arrays

Function	Argument Types	Result Type	Description
DIM(a)	a: Math Array	SIZE	number of dimensions
INCR(a,d)	a: Math array, d: SIZE	SIZE	return increment of dimension d
LEN(x)	x: ARRAY OF	SIZE	return length of x
LEN(x,d)	x: Math Array or Array	SIZE	return length of dimension d of x

12.1.5 Addresses, Memory and Types

Function	Argument Types	Result Type	Description
ADDRESSOF(v)	v: any designator	ADDRESS	returns the address of v
ADDRESS OF v	v: any designator	ADDRESS	returns the address of v
COPY(x,y)	x,y: ARRAY OF CHAR		0X-terminated copy of x to y
NEW(x,...)	x: pointer type		allocate x
NEW T(...)	pointer type T	T	allocate an instance of T
SIZEOF(T)	any type T	SIZE	returns the size of type T

Function	Argument Types	Result Type	Description
SIZE OF T	any type T	SIZE	returns the size of type T

12.1.6 Traps

Function	Argument Types	Result Type	Description
ASSERT(x)	x: BOOLEAN		raise trap, if x not true
HALT(n)	n: Integer		generate a trap with number n

12.1.7 Atomic Operations

Atomic Operations allow to read and modify data that is shared between two or more activities without requiring that data to be protected using exclusive blocks:

Function	Argument Types	Result Type	Description
CAS(x,y,z)	x,y,z: same T	T	compare-and-swap

The compare-and-swap procedure compares the value of the variable named in the first argument with the value of the second argument. If the two values of non-structured type match, the variable is overwritten with the value of the third argument. The result is equal to the original value of the variable. The whole operation is executed atomically and never interrupted by any other activity. If the second and third argument are the same, the whole operation effectively equals to an atomic read of a shared variable.

Other atomic operations like test-and-set can be implemented on top of the CAS procedure:

```
PROCEDURE TAS* (VAR value: BOOLEAN): BOOLEAN;
BEGIN RETURN CAS (value, FALSE, TRUE);
END TAS
```

12.2 The Module SYSTEM

The (pseudo-)module SYSTEM contains definitions that are necessary to directly refer to resources particular to a given computer and/or implementation. These include facilities for accessing devices that are controlled by the computer, and facilities to override the data type compatibility rules otherwise imposed by the language definition. The functions and procedures exported by this module should be used with care! It is recommended to restrict their use to specific low-level modules. Such modules are inherently non-portable and easily recognized due to the identifier SYSTEM appearing in their import list.

12.2.1 BIT Manipulation

Function	Argument Types	Result Type	Description
BIT(adr,n)	adr: ADDRESS; n: INTEGER	BOOLEAN	Returns TRUE if bit n at adr is set, FALSE otherwise

12.2.2 SYSTEM Types

Type	Description
BYTE	Representation of a single byte.

12.2.3 Unsafe Typecasts

Function	Argument Types	Result Type	Description
VAL(T,x)	T: Type; x: ANY	T	Unsafe type cast. Returns x interpreted as type T with no conversion

12.2.4 Direct Memory Access Functions

Function	Argument Types	Result Type	Description
PUT(adr,x)	adr: ADDRESS; x: Type		Mem[adr] := x where the size of type x is 8, 16, 32 or 64 bits

PUT8(adr,x)
PUT16(adr,x)
PUT32(adr,x)
PUT64(adr,x)
	adr: ADDRESS; x: (UN)SIGNED8
	adr: ADRESS; x: (UN)SIGNED16
	adr: ADDRESS; x: (UN)SIGNED32
	adr: ADDRESS; x: (UN)SIGNED64
			Mem[adr] := x

GET(adr,x)	adr: ADDRESS; VAR x: Type		x := Mem[adr] where the size of type x is 8, 16, 32 or 64 bits

GET8(adr)
GET16(adr)
GET32(adr)
	adr: ADDRESS
	adr: ADDRESS
	adr: ADDRESS
		SIGNED8
		SIGNED16
		SIGNED32
			RETURN Mem[adr]

Function	Argument Types	Result Type	Description
GET64(adr)	adr: ADDRESS	SIGNED64	
MOVE(src, dst,n)	dst: ADDRESS; n: SIZE		Copy "n" bytes from address "src" to address "dst"

12.2.5 Access to Registers

Function	Argument Types	Result Type	Description
GetStackPointer()		ADDRESS	return value of stack pointer register
SetStackPointer(x)	x: ADDRESS		set value of stack pointer register
GetFramePointer()		ADDRESS	return value of frame pointer register
SetFramePointer(x)	x: ADDRESS		set value of frame pointer register
SYSTEM.GetActivity		ADDRESS	return value of activity pointer register
SYSTEM.SetActivity	x: ADDRESS		set value of activity pointer register

12.2.6 Miscellaneous

Function	Argument Types	Result Type	Description
NEW(p,s)	p: any pointer, s: SIZE		allocate a piece of memory
TYPECODE(T)	type T	ADDRESS	return address of type descriptor of T
HALT(n)	n: Integer		Raise a trap with number n (unrestricted)
MSK(x,y)	x: Integer, y: Integer		Mask bits of y out of x
Time		ARRAY OF CHAR	return the time at compilation
Date		ARRAY OF CHAR	return the date at compilation

13 Systems Programming with Oberon

The most often observed problem of systems programmers new to Oberon is the lack of logical operators on integer numbers. There is no `a & b` and no `a | b`. Furthermore, bit-shifts have to be expressed either using functions such as `SHL` or `SHR` or using division. The following strategies are recommended for newbies in systems programming with Oberon.

- Use `DIV` and Multiplication when you want to shift integer numbers.
- Use `SHL` and `SHR`. Use an unsigned type when it is important that the sign bit is not propagated for right-shifts.
- Use `MOD` for masking lower bits.
- Use set operations when you want to operate on bit masks.
- Use `ODD` for bit-tests on integers.
- Use `SYSTEM.MOVE` when you want to copy large areas of memory.
- Use `SYSTEM.PUT` or `SYSTEM.GET` to read or write data byte-wise.
- Use `POINTER {UNSAFE} TO ...` when you want to access memory directly. Values compatible to type **`ADDRESS`** can be assigned to unsafe pointers.

Procedures, types and symbols can be flagged with special properties that influence the behavior of compiler and linker:

- Use **`PROCEDURE {PLAIN} X(...)`** to avoid that procedure X contains a procedure activation frame. Procedures marked **`NOPAF`** cannot provide variables or parameters.
- Use **`PROCEDURE {OPENING} X(...)`** to declare that a procedure should be linked first in an image. An opening procedure is always PLAIN.
- Use **`PROCEDURE {CLOSING} X(...)`** to declare that a procedure should be linked after all module bodies in an image. A closing procedure is always PLAIN.
- Use **`VAR a {UNTRACED}: POINTER ...`** in order to declare a pointer that is not traced by the Garbage Collector.
- Use **`a {ALIGNED(32)}: ...`** in order to make sure a symbol a gets aligned in memory accordingly.
- Use **`x EXTERN 100000H ...`** in order to make sure a symbol x gets pinned in memory accordingly.
- Use **`BEGIN {UNCHECKED} ... END`** in order to emit code without any checks such as stack-, null-pointer- or index bound checks.
- Use **`POINTER {UNSAFE} TO RECORD`** or **`POINTER {UNSAFE} TO ARRAY`** to declare a pointer that is inherently unsafe. An unsafe pointer is assignment compatible with

an address. Clearly, unsafe pointers cannot be type guarded or checked. Unsafe pointer to open arrays have no length and cannot be passed as normal array.

Finally, you can write entire procedures or portions thereof using assembler.

- Use `CODE ... END` in order to write inline assembler code

Example:

```
PROCEDURE {OPENING} KernelBegin;
CODE
  MOV bootFlag, EAX
  LEA EAX, initRegs
  MOV [EAX + 0], ESI
  MOV [EAX + 4], EDI
END KernelBegin;
```

Fig. 13.1: Code that is linked to the front of a kernel image.

14 Type Compatibility

There are only few cases of type compatibility that have to be checked by the compiler:

1. *Assignment Compatibility* is used in the following cases
   (a) in assignments `d := e`: The designator d must designate a non-read-only variable (or field) and the type of e must be assignment compatible to the type of d
   (b) in return statements within procedures: The type of the parameter a in the statement `return a` must be assignment compatible to the return type of the respective procedure.
   (c) in type guards `a(Tf)` and type checks `a IS Tf`: since records Ra and Rf are assignment compatible if Ra is an extension of Rf, the parameter compatibility of Ra and Rf equals the extension compatibility in a type guard and type test.
2. *Value Parameter Compatibility* is used in the following case
   (a) in procedure calls `P(...,a,...)` on value parameters: if P is defined as

   ```
   PROCEDURE P(..., [CONST] f: Tf, ...)
   ```

   then the type of the actual parameters a must be value parameter compatible to the formal (value) parameter Tf.

   Value parameter compatibility is equivalent with assignment compatibility if the formal type is not an open array.
3. *Variable Parameter Compatibility* is used in the following case
   (a) in procedure calls `P(...,a,...)` on variable parameters: if P is defined as

```
PROCEDURE P(...,VAR f: Tf, ...)
```

then the type of the actual parameter a must be variable parameter compatible to the formal (variable) parameter Tf.

4. *Expression Compatibility* is used for binary operators on expressions and needs a special treatment.

14.1 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 to equal Tb 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 and are not open arrays.

14.2 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.

14.3 Type Inclusion

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

$$\texttt{FLOAT64} \supseteq \texttt{FLOAT32} \supseteq \texttt{SIGNED64} \supseteq \texttt{SIGNED32} \supseteq \texttt{INTEGER} \supseteq \texttt{SIGNED8}$$

14.4 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.

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.

14.5 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:

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

14.6 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 the same type, or
2. Tf is an open array, Ta is any array, and their element types are array compatible, or
3. Tf is `ARRAY OF CHAR` and a is a string, or
4. Tf is an open enhanced array, Ta is any enhanced array, and their element types are array compatible, or
5. Tf is an open enhanced array, Ta is a tensor and the element type of Tf is an enhanced array and is array compatible to Ta
6. Tf is a tensor, Ta is any enhanced array and Tf

14.7 Expression Compatible

For a given operator, the types of its operands are expression compatible if they conform to the following table (which shows also the result type of the expression). Type T1 must be an extension of type T0:

operator	first operand	second operand	result type
$+ - *$	numeric	numeric	smallest numeric type including both operands
/	numeric	numeric	smallest FLOAT32 type including both operands

+ − * /	`SET`	`SET`	`SET`
`DIV` `MOD`	integer	integer	smallest integer type including both operands
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 T0 or T1	`NIL`, pointer type T0 or T1	`BOOLEAN`
	procedure type T, NIL	procedure type T, NIL	`BOOLEAN`
`IN`	integer	SET	`BOOLEAN`
`IS`	type T0	type T1	`BOOLEAN`

14.8 Variable Compatible

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

1. Tf and Ta are of the same type or
2. Tf and Ta are array compatible.

14.9 Parameter Compatible

In a procedure call the actual parameters must be parameter compatible with the formal parameters. Consider a parameter in a procedure with formal parameter type Tf and a call P(...,e,...) with actual parameter (expression) e with type Ta. Then the actual parameter a (an expression with type Ta) is parameter compatible to the formal parameter

1. if in the case of a value parameter, Ta is assignment compatible to Tf
2. if in the case of a variable parameter, Ta is variable compatible to Tf.

14.10 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 EBNF of Active Oberon

```
Module = 'MODULE' [TemplateParameters] Identifier ['IN' Identifier] ';'
         {ImportList} DeclarationSequence [Body]
        'END' Identifier '.'.

TemplateParameters = '(' TemplateParameter {',' TemplateParameter} ')'.

TemplateParameter = ('CONST' | 'TYPE') Identifier.

ImportList = 'IMPORT' Import { ',' Import } ';'.

Import = Identifier [':=' Identifier] ['(' ExpressionList ')' ] ['IN' Identifier].

DeclarationSequence = {
          'const' [ConstDeclaration] {';' [ConstDeclaration]}
         |'type' [TypeDeclaration] {';' [TypeDeclaration]}
         |'var'  [VariableDeclaration] {';' [VariableDeclaration]}
         | ProcedureDeclaration
         | OperatorDeclaration
         | ';'
         }

ConstantDeclaration = [IdentifierDefinition '=' ConstantExpression].

ConstantExpression = Expression.

VariableDeclaration = VariableNameList ':' Type.

VariableNameList = VariableName {"," VariableName}.

VariableName = IdentifierDefinition [Flags]
              [':=' Expression | 'EXTERN' String].

Flags = '{' [ Flag {',' Flag} ] '}'.

Flag = Identifier ['(' Expression ')' | '=' Expression].

ProcedureDeclaration = 'PROCEDURE' ['^'|'&'|'~'|'-'|Flags ['-']]
                       ['(' ParameterDeclaration ')']
                       IdentifierDefinition [FormalParameters] ';'
                       DeclarationSequence [Body]
                     'END' Identifier.

OperatorDeclaration = 'OPERATOR' [Flags] ['-'] String ['*'|'-'] FormalParameters ';'
                       DeclarationSequence
                       [Body]
                     'END' String.

FormalParameters = '(' [ParameterDeclaration {';' ParameterDeclaration}] ')'
                    [':' [Flags] Type].

ParameterDeclaration = ['VAR'|'CONST'] Identifier [Flags] ['=' Expression]
           {',' Identifier [Flags] ['=' Expression]} ':' Type.
```

```
Body = 'BEGIN' [Flags] StatementSequence ['FINALLY' StatementSequence]
          | 'CODE' Code.

TypeDeclaration = IdentifierDefinition '=' Type ';'.

Type = ArrayType | MathArrayType | RecordType | PointerType | ObjectType
       | ProcedureType | EnumerationType | QualifiedIdentifier
       | CellType | CellnetType | PortType.

ArrayType = 'ARRAY' [Expression {',' Expression}] 'OF' Type.

MathArrayType = 'ARRAY' '[' MathArraySize {',' MathArraySize} ']' 'OF' Type.

MathArraySize = Expression | '*' | '?'.

RecordType = 'RECORD' ['(' QualifiedIdentifier ')']
           [VariableDeclaration {';' VariableDeclaration}]
           {ProcedureDeclaration [';']| OperatorDeclaration [';']}
           'END'.

PointerType = 'POINTER' [Flags] 'TO' Type.

ProcedureType = 'PROCEDURE' [Flags] [FormalParameters].

ObjectType = 'OBJECT'
          | 'OBJECT' [Flags] ['(' QualifiedIdentifier ')']
             DeclarationSequence
             [Body]
            'END' [Identifier].

EnumerationType = 'ENUM' ['('QualifiedIdentifier')']
                  IdentifierDefinition ['=' Expression]
                  {',' IdentifierDefinition ['=' Expression]}
                'END'.

CellType = ('CELL' | 'CELLNET') [Flags] ['(' PortList ')'] [';'] {ImportList}
            DeclarationSequence
          [Body] 'END' [Identifier].


PortList = [PortDeclaration {';' PortDeclaration}].

PortDeclaration = Identifier [Flags] {',' Identifier [Flags]}':' PortType.

PortType = 'PORT' ('IN'|'OUT') ['(' Expression ')']

QualifiedIdentifier = Identifier ['.' Identifier].

IdentifierDefinition = Identifier [ '*' | '-' ].

ArrayLiteral = '[' Expression {',' Expression} ']'.

Set    =  "{" [Element {"," Element}] "}".

Element = RangeExpression.
```

```
Statement = [
  Designator
    [':=' Expression
    | '!' Expression | '?' Expression | '<<' Expresssion | '>>' Expression
    ]
  | 'IF' Expression 'THEN' StatementSequence
    {'ELSIF' Expression 'THEN' StatementSequence}
    ['ELSE' StatementSequence]
    'END'
  | 'WITH' Identifier ':' QualifiedIdentifier 'DO' StatementSequence
    {'|' QualifiedIdentifier 'DO' StatementSequence}
    [ELSE StatementSequence]
    'END'
  | 'CASE' Expression 'OF' ['|'] Case
    {'|' Case}
    ['ELSE' StatementSequence]
    'END'
  | 'WHILE' Expression 'DO'
      StatementSequence
    'END'
  | 'REPEAT'
      StatementSequence
    'UNTIL' Expression
  | 'FOR' Identifier ':=' Expression 'TO' Expression ['BY' Expression] 'DO'
      StatementSequence
    'END'
  | 'LOOP' StatementSequence 'END'
  | 'EXIT'
  | 'RETURN' [Expression]
  | 'AWAIT' Expression
  | StatementBlock
  | 'CODE' {any} 'END'
  | 'IGNORE' Designator
].

Case = RangeExpression {',' RangeExpression} ':' StatementSequence.

StatementBlock = 'BEGIN' [Flags] StatementSequence 'END'.

StatementSequence = Statement {';' Statement}.

Expression = RangeExpression [RelationOp RangeExpression].

RelationOp = '=' | '#' | '<' | '<=' | '>' | '>=' | 'IN' | 'IS'
           | '.=' | '.#' | '.<' | '.<=' | '.>' '.>='
           | '??' | '!!' | '<<?' | '>>?'.

RangeExpression = SimpleExpression
         | [SimpleExpression] '..' [SimpleExpression]['by' SimpleExpression]
         | '*' .

SimpleExpression = ['+'|'-'] Term {AddOp Term}.
AddOp          = '+' | '-' | 'or'.

Term = Factor {MulOp Factor}.
MulOp = '*' | '/' | 'div' | 'mod' | '&'
```

```
        | ’.∗’ | ’./’ | ’\’ | ’∗∗’ | ’+∗’ .

Factor = Number [Guard]
         | Character | String | Set | ArrayLiteral
         | Designator
         | ’(’ Expression ’)’ [Guard]
         | ’~’ Factor | Factor ’‘’
         | ’NIL’ | ’IMAG’ | ’TRUE’ | ’FALSE’ |
         | ’SIZE’ ’OF’ Designator | ’ADDRESS’ ’OF’ Designator
         | ’ALIAS’ OF Expression

Guard = "(" ExpressionList ")".

ExpressionList = Expression { ’,’ Expression }.

Designator   = DesignatorName [DesignatorOperations] [Flags].

DesignatorName = ’SELF’ | ’RESULT’ | ’ADDRESS’ | ’SIZE’ | Identifier
               | ’NEW’ QualifiedIdentifier ’(’ ExpressionList ’)’.

DesignatorOperations = { "(" [ExpressionList] ")"
                        | "." Identifier
                        | ’[’ ExpressionList ’]’
                        | ’^’
                       } .

IndexList = ’?’ [’,’ ExpressionList]
          |  ExpressionList [’,’ ’?’ [’,’ ExpressionList] ]


String = ’"’ {Character} ’"’ | "’" {Character} "’" | ’\"’ {Character} ’\"’.

Number     =  Integer | Real.

Integer    =  Digit {["’"]Digit} | Digit {["’"]HexDigit} ’H’
              | ’0x’ {["’"]HexDigit} | ’0b’ {["’"]BinaryDigit}.

Real       =  Digit {["’"]Digit} ’.’ {Digit} [ScaleFactor].

ScaleFactor = (’E’ | ’D’) [’+’ | ’−’] digit {digit}.

HexDigit   =  Digit | ’A’ | ’B’ | ’C’ | ’D’ | ’E’ | ’F’
              | ’a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ .

Identifier = Letter {Letter | Digit | ’_’ }.

Letter = ’A’ | ’B’ | .. |’Z’ | ’a’ | ’b’ | .. | ’z’ .

Digit = ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’.
```