ETH Oberon (2019) Language Report

Felix Friedrich, Florian Negele

October 12, 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 B in the appendix on page 73.

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 **FLOAT**32 (and as such assignment compatible to any floating point variable) but it is represented as **FLOAT**64 by the compiler. This implies that constant folding is applied with highest implemented accuracy and conversion to **FLOAT**64 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. Strings of length 2 can be used wherever a character constant is allowed and vice versa.

A string of length 1 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). In such a string, the double backslash \\ can be used to denote a single backslash.

```
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

```
Array = '[' 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

2.6.1 Difference to original Oberon

Array expressoins and array literals were not present in original Oberon and have been added.

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 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;
  Table = ARRAY N OF REAL
  Tree = POINTER TO Node
  Node = RECORD
    key : INTEGER;
    left, right: Tree
  END
  CenterTree = POINTER TO CenterNode
  CenterNode = RECORD (Node)
    width: INTEGER;
    subnode: Tree
  END
```

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

Fig. 7.1: Examples of Type Declarations

7.1 Categories of Types

The Active Oberon Language features the following classes of types:

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

7.1.1 Fundamental Types

Fundamental types are predefined by the Oberon language and can be addressed by the corresponding predefined identifiers.

Some of the types, shown in Table 1, 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).

Moreover, there are platform-dependent types, shown in Table 2 that grow or shrink with the target hardware. The platform-dependent types are provided by ways of implicit type declarations declaring each names as an alias to some fixed sized type.

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

Figure 7.2 shows the (implicit) compatibility of the integer types. (Sequence of) Arrows from A to B mean: 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 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 considered ok. The other direction does not work:

Type name	Size	Valid values
`BOOLEAN`	1 byte	`TRUE` or `FALSE`
`CHAR`	1 byte	characters of the extended ASCII set (`0X` ... `0FFX`)
`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: Fixed Size Fundamental Types

Type name	Size	Valid values
`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

Table 2: Platform Dependent Fundamental Types

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

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

Fig. 7.2: Integer Compatibilities

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 *)
```

7.1.2 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. **`SIGNED`**64), 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.

7.2 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.

Arrays can be declared in the following form:

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

There are 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. Open arrays are restricted to pointer base types, element types of open array types, and formal parameter types.

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)
```

7.3 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 of Tensors are not permitted.
- A length expression `x` in `ARRAY [x] OF` type must be a constant, positive integer or zero

Math arrays are considered *value types*. This implies that in an assignment `a := b` the data from b are *copied* to a. If a is an *open* mathematical array, then memory will automatically allocated. Mathematical arrays can thus also be used to declare open arrays. An open mathematical array is always initialized with lengths zero. A tensor is a mathematical array that not only has variable lenghts but even variable dimensions are possible. It is initialized with dimension 0. Lengths and dimension of mathematical arrays can be determined with the builtin functions **LEN** and **DIM**. The length and dimensions of a mathematical array can be set with the **NEW** operation.

```
VAR
  x: ARRAY [*] OF REAL;
  vec: ARRAY [4] OF REAL;
  matrix: ARRAY [*,*] OF REAL;
  array3: ARRAY [*,3] OF REAL;
  tensor: ARRAY [?] OF FLOAT32;
BEGIN
  NEW(x,5); (* x has now length 5 *)
  vec := x[0..3]; (* vec now has the content of x *)
  matrix := [x,x]; (* matrix of size 2 x 5 *)
  tensor := FLOAT32(matrix); (* tensor has dimension 2 *)
```

7.4 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.

A record can be marked with the `FINAL` modifier in which case it cannot be extended:

```
TYPE
   Date = RECORD {FINAL}
      day, month, year: INTEGER;
   END
```

7.5 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, unless `P` is an unsafe pointer (cf. below). 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 two kinds of (safe) 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;
```

7.6 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;
```

7.7 Object Types

Objects are basically *pointers to* 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.

Objects can have a body. This body is executed after the initializer. A body of an object can be *active*, in which case the body of the object is executed on a separate thread.

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

7.8 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*)
```

7.8.1 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.

7.9 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 ')'].
```

8 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.

The types of the listed variables can be omitted when initializers are present and the type should be inferred from the types.

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

8.0.1 Difference to original Oberon

The **`EXTERN`** flag was not present in Original Oberon. Moreover, we have added variable initializers and type inference from the expression type.

9 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;
```

If a procedure declaration specifies a receiver parameter, the procedure is considered to be bound to a type (see 9.2).

9.1 Formal Parameters

Formal parameters are identifiers declared in the formal parameter list of a procedure. They correspond to actual parameters specified in the procedure call. The correspondence between formal and actual parameters is established when the procedure is called. There are two kinds of parameters, value and variable parameters, indicated in the formal parameter list by the absence or presence of the keyword VAR. Value parameters are local variables to which the value of the corresponding actual parameter is assigned as an initial value. Variable parameters correspond to actual parameters that are variables, and they stand for these variables. The scope of a formal parameter extends from its declaration to the end of the procedure block in which it is declared. A function procedure without parameters must have an empty parameter list. It must be called by a function designator whose actual parameter list is empty too. The result type of a procedure can be neither a record nor an array.

Let Tf be the type of a formal parameter f and Ta the type of the corresponding actual parameter a. a needs to be compatible to f in a procedure calls. Compatibility rules are defined in 14.8.

9.2 Type Bound Procedures

Globally declared procedures may be associated with a record type declared in the same module. The procedures are said to be *bound to* the record type. The binding is expressed by the type of the receiver in the heading of a procedure declaration. The receiver may be either a variable or const parameter of record type `T` or a value parameter of type `POINTER TO T` (where `T` is a record type). The procedure is bound to the type `T` and is considered local to it.

If a procedure P is bound to a type T_0, it is implicitly also bound to any type T_1 which is an extension of T_0. However, a procedure P' (with the same name as P) may be explicitly bound to T_1 in which case it *overrides* the binding of P. P' is considered a redefinition of P for T_1. The signature of P and P' must match (see 14.2). If P and T_1 are exported P' must be exported too.

If v is a designator and P is a type-bound procedure, then `v.P` denotes that procedure P which is bound to the *dynamic type* of v. Note, that this may be a different procedure than the one bound to the static type of v. v is passed to P's receiver according to the parameter passing rules specified in Section 9.1.

If r is a receiver parameter declared with type T, `r.P^` denotes the (redefined) procedure P bound to the *base type* of T. The signature of both declarations must match (14.2).

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

```
END Insert;

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

PROCEDURE (t: TreeNodePointer) Copy (): TreeNodePointer; (
VAR c: TreeNodePointer;
BEGIN
NEW(c); (* ... *)
RETURN c;
END Copy;

PROCEDURE (t: CenterTreeNodePointer) Copy (): TreeNodePointer; (
VAR c: CenterTreeNodePointer;
BEGIN
NEW(c); (* ... *)
RETURN c;
END Copy;
```

Remark 4 *Note that the type of a receiver parameter can decide about the dynamic nature of the object that can receive a procedure call. It the receiver type is a pointer type, then the procedure can only be called on an object on the heap. If the receiver is a record type, then the procedure can be called on both, heap and stack, and it is not possible that the heap pointer escapes from the type.*

Constness of records: Naturally, if a variable of record type is const, only receivers with a const parameter can receive a call. This ensures that the const property of a record can not be undermined by a call to a type bound procedure.

9.3 Operator Declaration

Active Oberon, in the version presented here, supports operator overloading. Operators can be declared

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

10 Expressions

10.1 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.

10.2 Range Expressions

Range Expressions are defined as follows

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

10.2.1 Difference to original Oberon

The notion of Range Expressions has been introduced and thus the ranges used in the Set types has been elevanted to Expressions in order to be able to represent slices in Math Oberon.

10.3 Simple Expressions

```
SimpleExpression = Term {AddOp Term}.
AddOp            = '+' | '−' | 'or'.
```

10.3.1 Difference to original Oberon

The unary operator "-" and "+" were originally contained here in the simple expression. We have moved them to the Factor as we believe it should have higher precedence, similar to the logical not operator.

We found an example of an assertion (in code that was productive): the assertion

```
assert(−1 MOD 3 = −1); (∗ did not fail, but fails now ∗)
```

was used in order to determine that `a MOD 3` would be negative for $a < 0$, something that is indeed untrue for Oberon. In fact, `(−1) MOD 3 = 2`! The assertion held true because −`1 MOD 3` used to be −`(1 MOD 3)`.

10.4 Terms

```
Term  = Factor {MulOp Factor}.
MulOp = '∗' | '/' | 'DIV' | 'MOD' | '&'
       | '.∗' | './' | '\' | '∗∗' | '+∗' .
```

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

10.5 Factors

Factors are defined as

```
Factor = UnaryExpression | UnaryOperation Factor.
UnaryOperation = '~' | '+' | '−'.
```

10.5.1 Difference to original Oberon

The original Oberon specification contained keywords, literals and designators as parts of the Factor. We moved this to a separate productions: unary and primary expressions, to avoid ambiguities in the syntax.

10.6 Unary and Primary Expressions

Primary Expressions constitute expressions that are not combined via free standing binary or prefix-operators but rather formed by a designator, literal, keyword or special structural expression that may be suffixed by further operations such as a procedure call, an index operator, an up-call or dereference operator or selector.

Primary expressions can provide an expression with address (that can stand of the left hand side of an assignment, for instance) or an expression that only has a value.

```
UnaryExpression = PrimaryExpression [DesignatorOperations] [Flags].

PrimaryExpression = Number | Character | String | Set | Array
```

```
        | 'NIL' | 'IMAG' | 'TRUE' | 'FALSE' |
        | 'SELF' | 'RESULT' | 'ADDRESS' | 'SIZE'
        | 'SIZE' 'OF' Factor | 'ADDRESS' 'OF' Factor
        | 'ALIAS' OF Factor
        | 'NEW' QualifiedIdentifier '(' ExpressionList ')'
        | '(' Expression ')'
        | Identifier.

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

ExpressionList = Expression { ',' Expression }.

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

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

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

Fig. 10.1: Example of Designators

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;
```

10.6.1 Difference to original Oberon

The notion of a Designator, as it was present in previous descriptions of the language, has been replaced by a primary expression. Due to the more liberal handling of chained expressions like the example shown below, the distinction between expressions that can be used on the left hand side of an assignment operator and expressions that constitute only a value, cannot be made so clear any more.

```
expression(BinaryExpression).left.CompatibleTo(x)
```

Also, in order to avoid ambiguities in the EBNF, and to allow other features such as type guards on value types (such as literal numbers), we chose to unify such expressions and call them primary expressions.

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.

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

10.7 Operator Table

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). In the following, we assume that T1 is an extension of T0.

operator	first operand	second operand	result type
`+`, −, ∗	numeric	numeric	smallest numeric type including both operands
`/`	numeric	numeric	smallest `FLOAT`-type including both operands
`+`, −, ∗, `/`	`SET`n	`SET`m	Smallest `SET`-type including both operands
`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`	T0	type T1	`BOOLEAN`

11 Statements

```
Statement = [
  UnaryExpression
    [':=' Expression
    | '!' Expression | '?' Expression | '<<' Expresssion | '>>' Expression
    ]
  | 'VAR' Identifier [':=' Expression]
         {',' Identifier [':=' Expression]} [':' Type]
  | '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' Expression
].

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*.

```
UnaryExpression ':=' 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 Declaration Statement

```
'VAR' Identifier [':=' Expression]
     {',' Identifier [':=' Expression]} [':' Type];
```

Variables can be declared within a statement sequence. A declaration statement is indicated by the `VAR` keyword. Similar to variable declaration, the type of the variable can be omitted when an initializer is present and when the type of the initializer shall be taken for the variable type. The variable is visible within the statement sequence from the point of declaration to the end of the procedure scope. A variable can only be declared once in a procedure scope.

```
BEGIN
VAR a := 20, b: SIZE; (* both, a and b are of type SIZE *)
VAR c := a, d := 30; (* c is of type of a and d is SIGNED8 *)
```

```
...
END
```

11.3.1 Difference to original Oberon

We have introdcued the declaration statement motivated by the need to support a better way of initialization for value (record) types.

11.4 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.

```
UnaryExpression ["(" 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");
```

11.4.1 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.

11.5 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.

11.6 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
```

11.7 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;
```

11.8 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

11.8.1 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.

It should be noted that, technically, `WITH` is different from `CASE` in the following sense. While for `CASE` it is relatively straighforward to implement an optimized version with a branch table, this is not the case for `WITH`, particularly not with dynamic module loading. However, at least the indirect loading of the type tags as it would occur in a multi-case `IF` statement can be avoided here by keeping the type tag address in a register. This is a simple optimisation that can be implemented with little costs in a compiler.

11.9 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;
```

11.10 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 );
```

11.11 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;
```

11.12 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;
```

11.13 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;
```

11.14 Await Statement

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

```
'AWAIT' Expression
```

11.15 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 "-";
```

11.16 Ignore Statement

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

```
'IGNORE' Expression
```

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 **SIGNED**x or **UNSIGNED**x, **SIZE** or **INTEGER**.Float stands for any of **FLOAT**x or **REAL**. Number stands for Integers or Float. Set stands for any of **SET**x or **SET**. Complex stands for any of **COMPLEX**x 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

Function	Argument Types	Result Type	Description
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
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 **SIGNED**64(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

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

12.1.6 Procedure Access

The following built-in procedure allows accessing exported procedures without having to import it.

Function	Argument Types	Result Type	Description
GETPROCEDURE(x,y,z)	x and y array of characters, z procedure variable		get procedure y of type z from module x

The first two operands name the module and the procedure therein to look for. The third operand must be a procedure variable. The result is either NIL or a public procedure of the named module that matches the procedure name and the procedure type of the procedure variable. For fine-grained access control, global procedures like as variables can be exported either with an asterik or a minus sign. A procedure exported with a minus sign is still accessible by importing modules, but not using `GETPROCEDURE`.

`GETPROCEDURE` is useful for interactive programs, where the user types in or middle-clicks on commands to execute them. The following module provides an interface for executing standard Oberon commands given as a string whereas the module gets dynamically loaded if possible:

```
MODULE Commands;

IMPORT Strings;

PROCEDURE Execute− (CONST command: ARRAY OF CHAR);
VAR pos: SIZE; module, procedure: ARRAY 32 OF CHAR; result: PROCEDURE;
```

```
BEGIN
   IF Strings.FindCharacter ('.', command, pos) THEN
      Strings.Copy (command, module, 0, pos); INC (pos);
      Strings.Copy (command, procedure, pos, Strings.GetLength (command) − pos);
      GETPROCEDURE (module, procedure, result);
      IF result # NIL THEN result () END;
   END;
END Execute;

END Commands.
```

The following module show how that interface might be used:

```
MODULE Test;

IMPORT Commands;

PROCEDURE Hello*;
BEGIN
   TRACE ("Hello World!");
END Hello;

BEGIN
   Commands.Execute ("Test.Hello");
END Test.
```

12.1.7 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

12.1.8 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)
GET64(adr)
	adr: ADDRESS
	adr: ADDRESS
	adr: ADDRESS
	adr: ADDRESS
		SIGNED8
		SIGNED16
		SIGNED32
		SIGNED64
			RETURN Mem[adr]

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

Function	Argument Types	Result Type	Description
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 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 variable (or field) that is not read-only 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. For many arithmetic expressions, one of the two operands needs to be assigment compatible to the other. All other cases are described via the operator table in Section 10.7.

14.1 Equal Types

Two variables a and b with types T_a and T_b are of *equal type* if

1. T_a and T_b are both denoted by the same type identifier, or
2. T_a and T_b are declared to be equal in a type declaration of the form $\mathtt{T_a} = \mathtt{T_b}$, or
3. a and b appear in the same identifier list in a variable, field, or formal parameter declaration and are not open arrays, or
4. T_a and T_b are array types with coinciding lengths and same element type,

5. T_a and T_b are pointer types, both safe or both unsafe, with an equal pointer base type,
6. T_a and T_b are port types with the same direction and width
7. T_a and T_b are tensors with an equal element type
8. T_a and T_b are open math arrays with an equal element type
9. T_a and T_b are open math arrays with the same dimension and equal element type
10. T_a and T_b are static math arrays with the same dimensions, the same lengths and an equal element type
11. T_a and T_b are procedure types with a matching signature

Item 3. means that a and b have the same (potentially anonymous type). Example: `a, b: RECORD a: SIZE END;`

14.1.1 Comparison to original Oberon

Array types with the same length and element type were not considered same in the original Oberon. Port types are for Active Cells. Math array types are also new.

14.2 Matching Signature

Two procedure types have a *matching signature* if they both

1. have an equal return type
2. coincide in the following features: calling convention, delegate, no-return, interrupt, nestedness,
3. have the same number of formal parameters, each of the same kind (`VAR`, `CONST` or value) and with equal type.

14.3 Type Inclusion

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

$$\begin{aligned}&\texttt{FLOAT}64 \supseteq \texttt{FLOAT}32\\ \supseteq&\texttt{UNSIGNED}64 \supseteq \texttt{SIGNED}64\\ \supseteq&\texttt{UNSIGNED}32 \supseteq \texttt{SIGNED}32\\ \supseteq&\texttt{UNSIGNED}16 \supseteq \texttt{SIGNED}16\\ \supseteq&\texttt{UNSIGNED}8 \supseteq \texttt{SIGNED}8\end{aligned}$$

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 equal 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 l of type T_l is assignment compatible (`l := r` might be allowed) with a variable r of type T_r if one of the following conditions hold:

1. T_l and T_r are equal types and no open arrays;
2. T_l and T_r are numeric types and T_l includes T_r;
3. T_l and T_r are record types and T_r is an extension of T_l;
4. T_l and T_r are pointer types and T_r is an extension of T_l;
5. T_l is a pointer or a procedure type and r is `NIL`;
6. T_l is `ARRAY n OF CHAR`, r is a string constant with m characters, and $m < n$
7. T_l is `ARRAY OF CHAR` and r is a string constant;
8. T_l and T_r are equal procedure types;
9. T_l is an `ADDRESS` type and T_r is an unsafe pointer or vice versa
10. T_l and T_r are math array assignment compatible (cf. below).

14.6 Array Compatible

An actual parameter r of type T_r is array compatible with a formal parameter l of type T_l if

1. T_l and T_r are the same type, or
2. T_l is an open array, T_r is any array, and their element types are array compatible, or
3. T_l is `ARRAY OF CHAR` and r is a string, or
4. T_l is `ARRAY OF SYSTEM.BYTE` and T_r is any type.

14.7 Math Array Compatible

The base element type of a math array type T is defined recursively as follows: if T is a math array and its element type E is a math array, then B is the base element type of E, otherwise E is the base element type of T.

Math array variable compatible An actual parameter r of type T_r is *variable-compatible* to a formal parameter l of type T_l if they are both math arrays and the base element types B_l of T_l and B_r or T_r are equal types and T_r is shape-compatible to T_l-

Shape compatible A math array type T_r is *shape-compatible* to T_l if

1. T_l is a tensor
2. T_l is an open math array and T_r is a an open or static math array with the same number of dimensions.
3. T_l is a static math array and T_r is a static math array with the same number of dimensions and the same lenghts.

This definition of shape-compatibility obviously applies to the static properties of a math array. If a math array with dynamic features (such as variable lengths or dimension) turns out to be shape-incompatible during runtime, a trap is raised.

Math array assignment compatible An actual parameter r of type T_r is *math array asignment-compatible* to a formal parameter l of type T_l if they are both math arrays and the base element type B_l of T_l is assignment compatible to the base element type B_r of T_r and either of

1. T_r is shape compatible to T_l or
2. T_l is an open math array and T_r is a tensor

14.8 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 T_l and a call P(...,r,...) with actual parameter (expression) r with type T_r. Then the actual parameter r (an expression with type T_r) is parameter compatible to the formal parameter

1. if, in the case of a value or const parameter,
   (a) T_r is assignment compatible to T_l or
   (b) T_r is array compatible to T_l
2. if, in the case of a variable parameter, l can be modified, i.e. l has an address and l is not constant and either of
   (a) T_l and T_r are of equal type or
   (b) T_r is array compatible to T_l or
   (c) T_r is math array variable compatible to T_l.

A Math Array Types: Usage

Math Arrays are provided as a language extension to Oberon. Math arrays can be treated just like normal arrays in Active Oberon but they also extend the functionality considerably by the definition of array-substructures that can be accessed as parameters in procedure calls and as operands in arithmetic expressions.

Purpose of the math arrays is the possibility of intuitive and efficient mathematical programming, in particular in the field of (multi-)linear algebra. By a strict value-semantical approach the unnatural notion of pointers is consistently avoided.

A.1 Declaration

Math arrays are declared nearly like normal arrays in Active Oberon. The distinction is realized by additional square brackets in the array type. There are four different ways to declare a math array.

1. **Static arrays** are declared with a fixed number of dimensions and with constant lengths in the form `array [n1,n2,n3] OF BaseType` where `n1`, `n2` and `n3` must be constant.
2. **Dynamic arrays** are declared with a fixed number of dimensions and with open length field in the form `ARRAY [*,*] of BaseType`
3. **Tensor arrays** are declared with an open number of dimensions and with variable lengths in the form `ARRAY [?] of BaseType`. The number of dimensions is determined at runtime, for example in the allocation statement `NEW(a,1,2,3,4)` (4 dimensions).

The star ∗ stands for an arbitrary index while the question mark `?` stands for an arbitrary number of arbitrary indices. Therefore '`ARRAY [*,*] OF REAL` stands for a dynamic arrays of dimension two while `ARRAY [?] OF FLOAT32` stands for an array with arbitrary number of dimensions.

Some examples of declaration of enhanced arrays are given in the following figure.

```
VAR
S1: ARRAY [3,5] OF SIZE;
S2: ARRAY [3] OF ARRAY [5] OF SIZE; (* equivalent to S1 *)
D1: ARRAY [*,*] OF REAL;
D2: ARRAY [*] OF ARRAY [*] OF REAL; (* equivalent to D1 *)
T: ARRAY [?] of INTEGER;
```

Fig. A.1: Examples of enhanced array declaration

A.2 Single Element Access

Single element access is denoted just as in Active Oberon by square brackets. Single elements can be read or written. If an array is read-only or constant, then elements can only be read.

```
VAR
  A: ARRAY [*,*] of REAL;
  t: REAL; i,j: SIZE;
PROCEDURE p(VAR x: REAL);
BEGIN
    (* ... *)
    A[3,5] := t;
    t := A[3,5];
    P(a[i,j]);
```

Fig. A.2: Examples of single element access

A.3 Allocation

Dynamic enhanced arrays and arrays with a dynamic dimension have to be allocated before elements or substructures can be accessed. This can happen expressively with the built-in NEW-function or implicitly via assignment.

```
VAR
    a,b: ARRAY [*,*] of REAL;
    t,s: ARRAY [?] of REAL;
begin
    ...
    NEW(a,3,3); (* allocation *)
    NEW(t,3,3); (* allocation *)
    s := a; (* implicit allocation *)
    b := a; (* implicit allocation *)
    s := t; (* implicit allocation *)
    s := a; (* implicit allocation, dynamic dimension check *)
```

Fig. A.3: Examples of Allocation of arrays

A.4 Shape

The shape of arrays (geometric information) can be identified by the built-in **LEN** and **DIM**function. If an array has not been allocated yet, the built-in functions return values of zero.

`LEN` can be used with two or one parameters. Used with one parameter it returns a vector of lengths (that is an `ARRAY [*] OF SIZE`. With the built-in function `INCR` the internally used increments can be read for each dimension.

```
VAR
   a: ARRAY [*,*] of FLOAT32;
   t: ARRAY [?] of FLOAT32;
   d: SIZE;
   v: ARRAY [*] of SIZE;
begin
   ...
   i := LEN(a,d); (* number of elements along dimension d *)
   v := LEN(t); (* vector of lengths *)
   i := INCR(a,0); (* increment *)
   v := INCR(t); (* vector of increments *)
   d := DIM(s); (* number of dimensions of s *)
```

Fig. A.4: Examples how to retrieve the shape of arrays

A.5 Assignment and Constant Arrays

There is no notion of pointer to an array for math arrays. Math arrays are provided with value semantics. By the general determination of value-semantic design for such array types, assignment always denotes a copy of content ('deep copy') rather than a copy of references ('shallow copy'). In some cases the deep copy operation may – for optimization purposes – be a shallow copy, i.e. may be converted to a reference copy by the compiler. Preconditions for this is that the source operand cannot be reached any more after the assignment statement and that the destination operand allows overwriting of the descriptor. From an abstract point of view, however, the basic principle of value transfer persists.

The usage of assignment has already been indicated in Fig. A.3. With the introduction of value-copies also constant arrays make sense in the language.

```
VAR
   a: ARRAY [*,*] of REAL;
   t: ARRAY [?] of REAL;
CONST
   c = [[1,2,3],[4,5,6],[7,8,9]];
BEGIN
   ...
   a := c; (* assignment of constant array c to a *)
```

```
    t := [[1,2,3],[4,5,6],[7,8,9]]; (* assignment *)
```

Fig. A.5: Example of constant arrays and assignment

A.6 Ranges and Slices

Besides that arrays can be accessed element-wise, they can also be accessed in whole portions, namely in substructures that can be declared with so called *ranges*. Thereby indexing arrays with a list of ranges (and integers) results in a designator denoting a (rectangular, regular) part of the array.

```
VAR
    a: ARRAY [*,*] of REAL;
    t,s: ARRAY [?] of REAL;
CONST
    c = [[1,2,3],[4,5,6],[7,8,9]];
BEGIN
    ...
    a[1..3,2..4] := c; (* shape must match *)
    t := a[1..2,2..4]; (* allocation if necessary *)
    t[1..2,2..4] := a[1..2,2..4]; (* dynamic dimension check *)
    t[1,*] := [1,2,3];
    s[1,?] := a;
```

Fig. A.6: Example of ranges and their application

A.7 Parameters, Procedure Calls

Arrays can be used as constant and variable parameter for procedures or methods in Math Oberon. Variable parameters require that the array might be modified by the callee.

There are certain restrictions and runtime-checks when variables are passed. These will now be commented on in more detail in the following paragraphs.

Pass by Value. Pass by Value is forbidden for Math Arrays. It has been deliberately removed from the language and can be replaced without much overhead by assignments.

Pass by Variable. If there is a `VAR` modifier in the parameter declaration then the respective parameter is marked as a reference-parameter. If a parameter is passed by reference then it is in general not write-protected and can in some cases even be re-allocated. Modifications of the parameter do have effects on the caller's variable.

```
PROCEDURE VPS(VAR S: ARRAY [3,3] of Type)
PROCEDURE VPA(VAR A: ARRAY [*,*] of Type)
PROCEDURE VPT(VAR T: ARRAY [?] of Type)
```

Pass as constant. If there is the `const` keyword present in the parameter declaration then the parameter is marked as read-only parameter. It is therefore write protected within the procedure. This protection is transient, i.e. a write protected variable may not be used as reference parameter.

```
PROCEDURE PS(CONST S: ARRAY [3,3] of Type)
PROCEDURE PA(CONST A: ARRAY [*,*] of Type)
PROCEDURE PT(CONST T: ARRAY [?] of Type)
```

Return types. Math arrays can be returned by procedures. Procedures returning enhanced arrays can consequently also be passed as value- or constant parameters, but naturally not as variable parameters.

```
PROCEDURE P(): ARRAY [*,*] of Type;
PROCEDURE P(): ARRAY [3,3] of Type;
PROCEDURE P(): ARRAY [?] of Type;
```

Fig. A.7: Examples of array return types

A.7.1 Restrictions on the Callee side

By the high grade of interaction between the different array modes (static, dynamic, tensor), some checks have to be provided by the runtime. In this paragraph we comment on possible runtime-restrictions for the different possible declarations.

- **Pass static array, open array or tensor as constant** Array data and descriptor are immutable. The array is read-only and cannot be re-allocated nor modified in size.
- **Pass static array as variable** The array can be modified in content only with effect to the caller's variable.
- **Pass open array as variable** The array can in principle be modified both in content and size. However the allowance of a re-allocation of `A` is checked at runtime, since the passed variable might have been a static array or a range (as in `P(B[1..2,1..2])`). In this case a modification in size would not make sense.
- **Pass tensor array as variable** Array `A` can be modified both in size and content and even in the number of dimensions. However the allowance of both, the re-allocation

and a change of dimension, is checked at runtime. The re-allocation and change of dimension is forbidden, if a static array or a range was passed (as in `P(B[1..2,1..2])`). A modification of the number of dimensions is forbidden, if a static or dynamic array has been passed (as also in `P(B)` with `B: ARRAY [*,*] of Type`).

Remark 5 *Note that it is in general not guaranteed that the constant array* `A` *within the procedure* `P` *is physically identical with the caller's variable. This is particularly not the case, if a conversion of parameters has taken place. As an example consider the variable* `A: ARRAY [*] of INTEGER` *and the procedure* `PROCEDURE P(CONST A: ARRAY [*] of FLOAT32)`*. As explained in Section* **??***, it is possible to call* `P(A)` *with an automatic conversion of* `A` *resulting in a copy of content during the call.*

A.7.2 Restrictions on the Callers side

Additional to the possible restrictions within the called procedure (as described in the previous paragraph) there can also be (runtime-)restrictions on the caller's side due to the inter-operability of different array-modes.

Notation 1 *Let in the following the variables* `A`*,* `T`*,* `S`*,* `PA(...)`*,* `PS(...)`*,* `PT(...)` *be defined as follows:*

`VAR S: array [a,b] of Type`	*(static array)*
`VAR A: array [*,*] of Type`	*(open array)*
`VAR T: array [?]  of Type`	*(tensor)*
`PROCEDURE PS(...):  ARRAY [a,b] OF Type`	*(procedure returning static array)*
`PROCEDURE PA(...):  ARRAY [*,*] OF Type`	*(procedure returning open array)*
`PROCEDURE PT(...):  ARRAY [?]  OF Type`	*(procedure returning tensor)*

Additionally, we use the expression `A[a..b,c..d]` *for any range array substructure being passed to a procedure. Here* `A` *may freely be exchanged by* `T` *or* `S`*.*

Remark 6 *Note that the application of a range to an array in general incurs a runtime-check for the geometry, i.e. the lengths of each dimension. In case of tensors an additional runtime-check for the number of dimensions is adopted.*

Now using the notation from above, the following restrictions apply to the different displayed cases:

- **Pass static array as constant**

```
PROCEDURE P(CONST s: ARRAY [A,B] OF Type;
```

  - the type of `s` and the caller's variable have to match exactly. This is only possible if the types exactly match. Therefore only `P(S)` and `P(PS(...))` is possible. [1]
  - it is not possible to pass sub-array-structures

- **Pass static array as variable**

```
PROCEDURE P(VAR s: ARRAY [A,B] OF Type;
```

  - only allowed: `P(S)`.
  - restrictions same as above with the additional restriction that a procedure returning an array cannot be passed.

- **Pass open array as constant**

```
PROCEDURE P(CONST s: ARRAY [*,*] OF Type);
```

  - allowed: `P(A), P(A[a..b,c..d]), P(S), P(PA(...)), P(PS(...))`
  - if a tensor is passed (`P(T), P(PT(...))`) then the dimension is checked during runtime

- **Pass open array as variable**

```
PROCEDURE P(VAR s: ARRAY [*,*] OF Type);
```

  - allowed: `P(A), P(A[a..b,c..d]), P(T), P(S)`
  - a procedure returning an array cannot be passed
  - if a tensor is passed (`P(T), P(PT(...))`) then the dimension is checked at runtime

- **Pass tensor array as constant**

```
PROCEDURE P(CONST s: ARRAY [?] OF Type);
```

  - allowed: `P(A), P(A[a..b,c..d]), P(T), P(S), P(PA(...)), P(PT(...)), P(PS(...))`
  - no restrictions

- **Pass tensor array by reference**

```
PROCEDURE P(VAR s: ARRAY [?] OF Type);
```

  - allowed: `P(A), P(A[a..b,c..d]), P(T), P(S)`
  - a procedure returning an array cannot be passed

A.7.3 The Return Parameter

If a procedure returns a static array then the actual return value must be of the exact return type. Some forbidden cases are displayed in the next example.

[1]The current implementation even rejects the displayed case if a=A and b=B since the types are still distinguishable.

```
PROCEDURE P(): ARRAY [5,5] OF Type;
VAR a: ARRAY [*,*] OF Type;
   b: ARRAY [5,5] OF Type;
BEGIN
   NEW(a,5,5);
   RETURN a; (* forbidden, will not compile *)
   RETURN a[0..4,0..4]; (* forbidden, will not compile *)
   RETURN b[0..4,0..4]; (* forbidden, will not compile *)
   RETURN b[*,*]; (* forbidden, will not compile *)
   RETURN b; (* ok *)
END P;
```

Fig. A.8: Examples of a procedure returning static arrays

In procedures returning open arrays any array can be returned if the number of dimensions and the base types match. If tensors are displayed then the number of dimensions is checked during runtime. Otherwise the compiler can check it.

```
PROCEDURE P(): ARRAY [*,*] OF Type;
VAR a: ARRAY [*,*] OF Type;
   t: ARRAY [?] OF Type;
   s: ARRAY [3,5] OF Type;
BEGIN
   NEW(a,5,5); NEW(t,5,5);
   RETURN a; (* ok *)
   RETURN a[*,*]; (* ok *)
   RETURN s; (* ok *)
   RETURN s[0..2,0..2]; (* ok *)
   RETURN t; (* runtime check for the number of dimensions *)
   RETURN t[*,*]; (* runtime check already in the range *)
END P;
```

Fig. A.9: Examples of a procedure returning open arrays

Procedures returning tensors do not incur any restriction to the return values, neither at compile- nor during runtime.

A.8 Built-in Operators

A large set of mathematical operators have been defined and implemented for the new enhanced array types. In the following code examples, the following types are used: `A:array [?]  OF Type`, `v:array [*] OF integer`, `s:number`, `b:boolean`

Array ↦ Scalar

```
s := MIN(A); s := MAX(A); s := SUM(A);
```

Array ↦ *Array*
element-wise operators:

```
A := −B;
A := ~ B;
A := ABS(B);
A := SIGNED64(B); A := ENTIER(B);
```

other operator

```
A := B‘; transposition
```

Array × *Array* ↦ *Scalar*

```
s := B +∗ C;
b := B = C; b := B < C; b := B <= C;
b := B > C ; b := B >= C; b := B \# C;
```

Array × *Scalar* ↦ *Array* | *Scalar* × *Array* ↦ *Array*

```
A := B + s; A := s + B; A := B − s; A := s − B;
A := s ∗ B; A := B ∗ s;
A := B / s ; A := s / B;
A := s DIV B; A := B DIV s;
A := s MOD B; A := B MOD s;
```

Array × *Array* ↦ *Array*
element-wise operators:

```
A := B DIV C; A := B MOD C;
A := B + C; A := A − C;
A := B .∗ C; A := B ./ C;
A := B OR C; A := B & C;
A := B .= C; A := B .< C; A := B .<= C;
A := B .> C; A := B .>= C; A := B .# C;
```

other operators

```
A := B ∗ C; matrix / vector product
A := B ∗∗ C; tensor product
A := RESHAPE(B,v); reshape operation
```

A.9 Type Compatibilities

Due to the automatic conversions that can be performed during runtime (behind the scenes), enhanced arrays are subject to different type compatibility rules than traditional arrays. Naturally, elements of arrays are accessed just as ordinary data types and therefore this statement only applies to expressions containing (substructures) of arrays (called array designators). Array designators are used in assignments, as variable or value parameter in procedure call and (implicitly) for the type compatibility of procedure variables.

A.9.1 Math Array Assignment Compatibility

Arrays and substructures of arrays are assignment compatible if for

```
dest := src;
```

the following conditions hold.

1. – either `dim(src)=dim(dest)`: the number of dimensions of dest equals the one of src
   – or `dest` is permitted to be made dimension compatible by reallocation (i.e. `dest` must be a tensor type).

   For enhanced arrays this may be checked at compile-time whereas for tensor types this can in general only be determined during runtime.
2. – either `len(src,i)=dim(dest,i)` for all *i*: the shapes of dest and src coincide
   – or the shape of `dest` is permitted to be made compatible to the shape of `src` by reallocation.

   If source and destination are both static arrays or static substructures then this may be checked during compile time, otherwise it can only made sure during runtime.
3. The basetypes of dest and src have to be assignment compatible.

A.9.2 Value Parameter Compatibility

Arrays and substructures of arrays are value parameter compatible if for

```
PROCEDURE P(CONST a: FormalType);

P(A);
```

the (actual) type of `A` is assignment compatible to the (formal) type of `a` in the definition of `P`. Thus the expression `P(A)` is (statically and dynamically) admissible, if the expression `a := A` would be admissible in the given context.

A.9.3 Variable Parameter Compatibility

Arrays and substructures of arrays are variable parameter compatible if for

```
PROCEDURE P(var f: FormalType);

P(A);
```

the (actual) type of `A` and the (formal) type of `f` are

- either identical
- or it holds that
  1. the base type of `A` and `f` are identical and
  2. for the shapes `Shape(a)` and `Shape(A)` of `a` and `A`, respectively, it is true that `shape(a)` $\in$ `shape(A)`. The following different cases have to be distinguished:
     (a) the shape of `f` is static. Then the shape of the type of `A` has to be static also and the lengths must coincide.
     (b) the shape of `f` is open but of fixed dimension. Then the shape of the type of `A` must be of the same dimension or dynamic.
     (c) the shape of `f` is a tensor. Then the shape of `A` may be arbitrary.

A.9.4 Procedure Type Compatibility

Two procedures `P` and `Q` are procedure type compatible (i.e. for a variable `p` of a type identical to that of `P` it would be allowed to assign `p := Q`) with respect to enhanced arrays, if corresponding array parameters `a` and `b` of `P` and `Q` have the same base types and if they are identical in shape, i.e. `Base(a)=Base(b)` and `Shape(a)=Shape(b)`.

A.9.5 Formalization of the array types. Domains and Shapes

Although we do not like over-formalization very much and want to avoid it as much as possible in favor of examples and intuition, here we think that a formal treatment of array types can bring us considerably forward with respect to the general understanding of structured types.

As already noted previously, there are two equally important views on an array that have to be taken into account to understand the matter in full detail. On the one hand, arrays contain data and therefore a sufficiently large storage for base type values has to be provided by the system. On the other hand the geometric structure of an array plays the very important role of structuring the data just as it is the case with records or the like. The significant difference between arrays and records is that arrays can have a dynamic structure whereas records / objects are in general statically sized and structured.

With a mathematical formalization we now try to get some insight into the structuring of arrays. We introduce the notions *domain*, *type-domain*, *shape* and *type-shape* of arrays. Informally speaking, the domain of an array variable is the set of possible indexes that can be applied to access a single array element. The domain of an array variable is not necessarily the same as the one of the respective type of the array. This is due to the fact that an array declaration such as `var a: array [*,*] of FLOAT32` leaves open the set of possible indices, while during runtime the statement `new(a,3,3)` substantiates the set of admissible values of `i` and `j` within `v := a[i,j]`. The type-domain of `a` is the set containing all potential domains of `a`. In the given example this is $\bigcup_{i,j\in\mathbb{N}} S_{ij}$ with $S_{ij} = \{1,\dots,i\}\times\{1,\dots,j\}$. With

$i = j = 3$ the domain of `a` is then $\{1,\dots,3\} \times \{1,\dots 3\}$. In the following table we specify the domain and shape for different types and variables.

Declaration	Domains	Shapes
`M:array n of B`	$D(M) = \{\{1,\dots,n\}\}$	$S(M) = \{n\}$
`r:M`	$D(r) \in D(r) = \{1,\dots,n\}$	$S(r) = n$
`M:array [*] of B`	$D(M) = \bigcup_{n\in\mathbb{N}}\{\{1,\dots,n\}\}$	$S(M) = \mathbb{N}$
`r:M`	$D(r) = \{1,\dots,n\}$ for one n	$S(r) \in \mathbb{N}$
`M:array [*,*] of B`	$D(M) = \bigcup_{n,m\in\mathbb{N}}\{\{1,\dots,n\} \times \{1,\dots,m\}\}$	$S(M) = \mathbb{N}^2$
`r:M`	$D(r) = \{\{1,\dots,n\} \times \{1,\dots,m\}\}$ for one (n,m)	$S(r) \in \mathbb{N}^2$
`M:array [?]  of B`	$D(M) = \bigcup_{d\in\mathbb{N}}\bigcup_{k\in\mathbb{N}^d} S_d(k)$	$S(M) = \bigcup_d \mathbb{N}^d$
`r:M`	$D(r) = \{1,\dots,k_1\} \times \cdots \times \{1,\dots,k_d\}$ for one k, d	$S(r) \in \mathbb{N}^d$ for one $d \in \mathbb{N}$

We have used the abbreviation $S_d(k) = \{1,\dots,k_1\} \times \cdots \times \{1,\dots,k_d\}$.

With this tools we can express the value parameter compatibility of an array `a` of (actual) type `A` with a procedure parameter `f` with (formal) type `F` very easily. Consider the following code

```
VAR a: A;
PROCEDURE P(f: F);
...
P(a); (* is this a valid call ? *)
```

The expression `P(a)` is valid, if the base types are compatible and if $D(a) \in D(F)$, equivalently if $S(a) \in S(F)$. Since $S(a)$ is not known at compile time, the best that the compiler can do is check if $S(A) \subseteq S(F)$. We have implemented this type check with one exception: in the case of `A` being a tensor type and `F` being an open array type, we only check $S(A) \cap S(F) \neq \emptyset$ at compile time, the rest is checked during run time.

A.10 Accessibility of return parameters

To give programmers the possibility to optimize code with regards to the allocation of arrays, the return parameter of a procedure returning an array or tensor can now be accessed like a reference parameter. It is named **RESULT**.

```
PROCEDURE Test(CONST a: ARRAY [?] OF FLOAT32): ARRAY [?] OF FLOAT32;
BEGIN
    if LEN(a)#LEN(RESULT) then
        NEW(RESULT,len(a));
    end;
    return RESULT; (* does nothing, prevents copying *)
END Test;
```

B 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 [ '*' | '-' ].

Statement = [
  UnaryExpression
    [':=' 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' Expression
].

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 = UnaryExpression | UnaryOperation Factor.

UnaryOperation = '~' | '+' | '-'.

UnaryExpression = PrimaryExpression [DesignatorOperations] [Flags].

PrimaryExpression = Number | Character | String | Set | Array
       | 'NIL' | 'IMAG' | 'TRUE' | 'FALSE' |
       | 'SELF' | 'RESULT' | 'ADDRESS' | 'SIZE'
       | 'SIZE' 'OF' Factor | 'ADDRESS' 'OF' Factor
       | 'ALIAS' OF Factor
       | 'NEW' QualifiedIdentifier '(' ExpressionList ')'
       | '(' Expression ')'
```

```
        | Identifier.

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

ExpressionList = Expression { ’,’ Expression }.

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

Array = ’[’ Expression {’,’ Expression} ’]’.

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

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’.
```