2.1.7.10 Integer literals are unsigned, and there are different suffix types.

2.1.7.11 Octal and binary integer literals are supported in addition to hexadecimal.

2.1.7.12 Only one-dimensional arrays (but array x of array y of T is still possible.

2.1.7.13 There is a goto statement (mostly to easy porting existing C code)

2.1.7.14 String concatenation only allowed for string and character literals.

2.1.7.15 Symbols, declared via enumeration type

## 2.2 Syntax

2.2.1 An extended Backus-Naur Formalism (EBNF) is used to describe the syntax of Micron:

2.2.1.1 Alternatives are separated by **|**.

2.2.1.2 Brackets **[** and **]** denote optionality of the enclosed expression.

2.2.1.3 Braces **{** and **}** denote its repetition (possibly 0 times).

2.2.1.4 Syntactic entities (non-terminal symbols) are denoted by English words expressing their intuitive meaning.

2.2.1.5 Symbols of the language vocabulary (terminal symbols) are denoted by strings formatted in bold face.

## 2.3 Vocabulary and Representation

2.3.1 Micron source code is a string of characters encoded using the UTF-8 variable-width encoding as defined in ISO/IEC 10646. Identifiers, numbers, operators, and delimiters are represented using the ASCII character set; strings and comments can be either represented in the ASCII or Latin-1 (as defined in ISO/IEC 8859-1) character set.

2.3.2 The following lexical rules apply: blanks and line breaks must not occur within symbols (except in block comments, and blanks in strings); they are ignored unless they are essential to separate two consecutive symbols. Capital and lower-case letters are considered as distinct.

### 2.3.3 Identifiers

2.3.3.1 Identifiers are sequences of letters, digits and underscore. The first character must be a letter or an underscore.

2.3.3.2 Syntax:

2.3.3.2.1

```
ident  = ( letter | '_' ) { letter | digit | '_' }

letter = 'A' ... 'Z' | 'a' ... 'z'

digit  = '0' ... '9'
```

2.3.3.3 Examples:

2.3.3.3.1

```
x

Scan

Oberon_2

_y

firstLetter
```

### 2.3.4 Numbers

2.3.4.1 Number literals are integer or real constants.

2.3.4.2 The type of an integer constant is the minimal unsigned integer type to which the constant value belongs (see 2.6.4 Basic types).

2.3.4.3 If the literal is specified with the suffix 'H', the representation is hexadecimal, if it is specified with suffix 'O', the representation is ocal, or if it is specified with suffic 'B', the representation is binary, otherwise the representation is decimal.

2.3.4.4 The type of an integer constant can be explicitly set with a suffix. The U8, U16, U32 or U64 suffices correspond to the UINT8, UINT16, UINT32 or UINT64 unsigned integer types. If the given constant value cannot be represented by the explicit type, the compiler reports an error.

2.3.4.5 **NOTE** Number literals can be prefixed with a sign. But the sign is not part of the literal, but of a simple expression including the literal, see 2.8.3 Operators. Use the SIGNED() predeclared function for a signed integer reinterpretation of the unsigned integer (e.g. SIGNED(0FFH) is an INT8 with value -1). Otherwise the term -0FFH becomes an INT16 with value -255.

2.3.4.6 Signed integer types are represented in two's complement form.

2.3.4.7 A real number always contains a decimal point and at least one digit before the point. Optionally it may also contain a decimal scale factor. The letter 'E' means *times ten to the power of*.

2.3.4.8 The type of a floating-point constant is the minimal Floating-point types to which the constant value belongs.

2.3.4.9 A real number is of type `LONGREAL`, if it has a scale factor containing the letter 'D', or of type `REAL`, if it has a scale factor containing the letter 'F'. If the scale factor contains the letter 'E', the type is `LONGREAL` if the mantissa or exponent are too

large to be represented by REAL.

2.3.4.10 Syntax:

2.3.4.10.1
```
number   = integer | real

integer  = (digit {digit} ['O'|'B' | digit {hexDigit} 'H')
           ['U8'|'U16'|'U32'|'U64']

real     = digit {digit} '.' {digit} [Exponent]

Exponent = ('E' | 'D' | 'F' ) ['+' | '-'] digit {digit}

hexDigit = digit | 'A' ... 'F'

digit    = '0' ... '9'
```

2.3.4.11 Even though only the upper-case version is shown here, also the lower-case versions of the suffices and scale factor letters are supported.

2.3.4.12 Examples:

2.3.4.12.1
```
1234

0dh              0DH

12.3

4.567e8          4.567E8

0.57712566d-6    0.57712566D-6
```

### 2.3.5 Characters

2.3.5.1 Character constants are denoted by the ordinal number of the character in hexadecimal notation followed by the letter X (or x).

2.3.5.2 Syntax:

2.3.5.2.1
```
character = digit {hexDigit} ('X' | 'x')
```

2.3.5.3 A character is encoded as a 8-bit code value using the ISO/IEC 8859-1 Latin-1 encoding scheme.

### 2.3.6 Strings

2.3.6.1 Strings are sequences of printable characters enclosed in single (') or double (") quote marks. The opening quote must be the same as the closing quote and must not occur within the string. A string must not extend over the end of a line. The number of characters in a string is called its length. A string of length 1 can be used wherever a character constant is allowed and vice versa.

2.3.6.2 Syntax:

2.3.6.2.1
```
string = ''' {character} ''' | '"' {character} '"'
```

2.3.6.3 Examples:

2.3.6.3.1
```
'Oberon'

"Don't worry!"

'x'
```

#### 2.3.6.4 Hex Strings

2.3.6.4.1 Hex strings are sequences of bytes encoded in hexadecimal format and enclosed in dollar signs. The number of hex digits in the string must be even, two hex digits per byte. The number of bytes in a hex string is called its length. Line breaks and other white space between the dollar signs is ignored. Both upper and lower-case hex digits are supported.

2.3.6.4.2 Syntax:

2.3.6.4.2.1
```
hexstring = '$' {hexDigit} '$'
```

2.3.6.4.3 Examples:

2.3.6.4.3.1
```
const arrow = $0F0F 0060 0070 0038 001C 000E 0007 8003 C101 E300
               7700 3F00 1F00 3F00 7F00 FF00$
```

### 2.3.7 Operators and Delimiters

2.3.7.1 Operators and delimiters are the special characters, or character pairs listed below.

2.3.7.2

| | | | | | |
|---|---|---|---|---|---|
| - | , | ; | : | := | . |
| .. | ( | ) | [ | ] | { |
| } | * | / | # | ^ | + |
| <= | = | >= | \| | ~ | |

### 2.3.8 Reserved Words

2.3.8.1 The reserved words consist of either all capital or all lower case letters and cannot be used as identifiers. All words listed below are reserved (only capital letter versions shown).

2.3.8.2

| | | | | |
|---|---|---|---|---|
| ARRAY | BEGIN | BY | CASE | CONST |
| | DIV | DO | ELSE | ELSIF |
| END | EXIT | FALSE | FOR | IF |
| IMPORT | IN | INLINE | LOOP | MOD |
| MODULE | NIL | OF | OR | POINTER |
| PROC | PROCEDURE | RECORD | REPEAT | RETURN |
| THEN | TO | TRUE | TYPE | UNTIL |
| VAR | WHILE | | | |

### 2.3.9 Comments

2.3.9.1 Comments are arbitrary character sequences opened by the bracket `(*` and closed by `*)`. Comments may be nested. They do not affect the meaning of a program. Oberon+ also supports line comments; text starting with `//` up to a line break is considered a comment.

## 2.4 Declarations and scope rules

2.4.1 Every identifier occurring in a program must be introduced by a declaration, unless it is a predeclared identifier. Declarations also specify certain permanent properties of an object, such as whether it is a constant, a type, a variable, or a procedure. The identifier is then used to refer to the associated object.

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

2.4.2.1 No identifier may denote more than one object within a given scope (i.e. no identifier may be declared twice in a block);

2.4.2.2 An object may only be referenced within its scope;

2.4.2.3 A type*T* of the form POINTER TO *T1* (see 2.6.7 Pointer types) can be declared at a point where *T1* is still unknown. The declaration of *T1* must follow in the same block to which *T* is local;

2.4.2.4 Identifiers denoting record fields (see 2.6.6 Record types) are valid in record designators only.

2.4.3 An identifier declared in a module block may be followed by an export mark (" * " or " - ") in its declaration to indicate that it is exported. An identifier *x* exported by a module *M* may be used in other modules, if they import *M* (see 2.12 Modules). The identifier is then denoted as *M.x* in these modules and is called a *qualified identifier*. Identifiers marked with " - " in their declaration are *read-only* in importing modules.

2.4.4 Syntax:

2.4.4.1

```
qualident = [ident '.'] ident

identdef  = ident ['*' | '-']
```

2.4.5 The following identifiers are predeclared; their meaning is defined in the indicated sections; either all capital or all lower case identifiers are supported (only capital versions shown).

2.4.6

| | | | | |
|---|---|---|---|---|
| ABS | ANY | ASSERT | BITAND | BITASR |
| | | | | |

| | | | | |
|---|---|---|---|---|
| BITNOT | BITOR | BITS | BITSHL | BITSHR |
| BITXOR | BOOLEAN | BYTE | CAP | CAST |
| CHAR | CHR | DEC | DEFAULT | DISPOSE |
| EXCL | F32 | F64 | FLOOR | FLT |
| HALT | INC | INCL | INT16 | INT32 |
| INT64 | INT8 | INTEGER | LEN | LONG |
| LONGINT | LONGREAL | MAX | MIN | NEW |
| ODD | ORD | PCALL | RAISE | REAL |
| SET | SHORT | SHORTINT | SIGNED | SIZE |
| STRLEN | SYMBOL | UINT16 | UINT32 | UINT64 |
| UINT8 | UNSIGNED | VA_LIST | VARARG | VARARGS |

## 2.5 Constant declarations

2.5.1 A constant declaration associates an identifier with a constant value.

2.5.2 Syntax:

2.5.2.1
```
ConstDeclaration = identdef '=' ConstExpression

ConstExpression  = expression
```

2.5.3 A constant expression is an expression that can be evaluated by a mere textual scan without actually executing the program. Its operands are constants (see 2.8.2 Operands) or predeclared functions (see 6 Predeclared Procedure Reference) that can be evaluated at compile time. Examples of constant declarations are:

2.5.4 Examples:

2.5.4.1
```
N = 100

limit = 2*N - 1

fullSet = {min(set) .. max(set)}
```

2.5.5 **NOTE** For compile time calculations of values the same rules as for runtime calculation apply. The ConstExpression of ConstDeclaration behaves as if each use of the constant identifier was replaced by the ConstExpression. An expression like `MAX(INTEGER)+1` thus causes an overflow of the INTEGER range. To avoid this either `LONG(MAX(INTEGER))+1` or `MAX(INTEGER)+1L` has to be used.

## 2.6 Type declarations

2.6.1 A data type determines the set of values which variables of that type may assume, and the operators that are applicable. A type declaration associates an identifier with a type. In the case of structured types (arrays and records) it also defines the structure of variables of this type. A structured type cannot contain itself.

2.6.2 Syntax:

2.6.2.1
```
TypeDeclaration = identdef '=' type

type            = NamedType | ArrayType | RecordType | PointerType | ProcedureType | enumeration

NamedType       = qualident
```

2.6.3 Examples:

2.6.3.1
```
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
```

### 2.6.4 Basic types

2.6.4.1 The basic types are denoted by predeclared identifiers. The associated operators are defined in 2.8.3 Operators and the predeclared function procedures in 6 Predeclared Procedure Reference. Either all capital or all lower case identifiers are supported (only capital versions shown). There are fixed and variable size basic types. For the fixed size basic types the byte widths and ranges are explicitly specified herein. The variable size basic types are just alternative names for the fixed size integer types.

2.6.4.2 The values of the given fixed size basic types are the following:

2.6.4.3

| | | |
|---|---|---|
| BOOLEAN | 1 byte | the truth values true and false |
| CHAR | 1 byte | the characters of the Latin-1 set (0x .. 0ffx) |
| BYTE, UINT8 | 1 byte | the integers between 0 and 255 |
| INT8 | 1 byte | the integers between -128 and 127 |
| INT16 | 2 byte | the integers between -32'768 and 32'767 |
| UINT16 | 2 byte | the integers between 0 and 65'535 |
| INT32 | 4 byte | the integers between -2'147'483'648 and 2'147'483'647 |
| UINT32 | 4 byte | the integers between 0 and 4'294'967'295 |
| INT64 | 8 byte | the integers between -9'223'372'036'854'775'808 and 9'223'372'036'854'775'807 |
| UINT64 | 8 byte | the integers between 0 and 18'446'744'073'709'551'615 |
| REAL, F32 | 32 bit | an IEEE 754 floating point number |
| LONGREAL, F64 | 64 bit | an IEEE 754 floating point number |
| SET | 4 byte | the sets of integers between 0 and MAX(SET) |

2.6.4.4 The values of the given variable size basic types are the following:

2.6.4.5

| | |
|---|---|
| SHORTINT | the integers between MIN(SHORTINT) and MAX(SHORTINT) |
| INTEGER | the integers between MIN(INTEGER) and MAX(INTEGER) |
| LONGINT | the integers between MIN(LONGINT) and MAX(LONGINT) |

2.6.4.6 INT64, INT32, INT16, INT8, LONGINT, INTEGER, SHORTINT are signed integer types, UIN64, UINT32, UINT16, UINT8, and BYTE are unsigned integer types, REAL and LONGREAL are Floating-point types, and together they are called numeric types. The larger type includes (the values of) the smaller type according to the following relations:

2.6.4.7

```
INT64 >= INT32 >= INT16 >= INT8

UNT64 >= UINT32 >= UINT16 >= UINT8

LONGREAL >= REAL

LONGINT >= INTEGER >= SHORTINT
```

2.6.4.8 **NOTE** Oberon and Oberon+ both support implicit type casts from integer to floating point and vice versa. Micron requres an explicit cast. To convert an integer to a floating point type, the FLT() built-in function should be used. To convert a numeric type to a signed integer, the SIGNED() function should be used. To convert a numeric byp to an unsigned integer, the UNSIGNED() function should be used.

2.6.4.9 A compiler may map the variable size integer names to any of the fixed size signed integers as long as the inclusion relations are obeyed. By default a correspondence of LONGINT with INT64, INTEGER with INT32 and SHORTINT with INT16 is assumed.

### 2.6.5 Array types

2.6.5.1 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 length is a positive integer. The elements of the array are designated by indices, which are integers between 0 and the length minus 1.

2.6.5.2 Syntax:

2.6.5.2.1
```
ArrayType  = ARRAY [ length ] OF type | '[' [ length ] ']' type

length     = ConstExpression | VAR varlength

varlength  = expression
```

2.6.5.3 **NOTE** In contrast to Oberon and Oberon+, Micron only supports one-dimensional length lists. But it is still possible that the type of the array is yet another array.

2.6.5.4 Arrays declared without length are called *open arrays*. They are restricted to pointer base types (see 2.6.7 Pointer types), element types of open array types, and formal parameter types (see 2.11.10 Formal parameters).

2.6.5.5 Examples:

2.6.5.5.1
```
array 10 of integer

array of char

[N] T
```

2.6.5.6 Local variables of array type can have variable lengths calculated at runtime; in this case the LengthList is prefixed with the VAR reserved word; the expression cannot reference other local variables of the same scope.

2.6.5.7 **NOTE** In contrast to array pointers allocated with NEW(), variable length arrays (VLA) can be allocated on the stack instead of the heap (depending on the compiler and supported options), which makes them attractive to low-resource embedded applications where dynamic memory allocation is not feasible. It is also interesting to note that already the length/range of ALGOL 60 arrays was defined using an ordinary arithmetic expression and thus could be calculated at runtime; even ALGOL W had this feature, but unfortunately it was removed in Pascal, and even Oberon-07 still uses a const expression for array lengths evaluated at compile time.

2.6.5.8 Array lengths of at least MAX(INT32) shall be accepted by a compiler, for constant, dynamic and variable lengths.

### 2.6.6 Record types

2.6.6.1 A record type is a structure consisting of a fixed number of elements, called fields, with possibly different types. The record type declaration specifies the name and type of each field. The scope of the field identifiers extends from the point of their declaration to the end of the record type, but they are also visible within designators referring to elements of record variables (see 2.8.2 Operands). 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.

2.6.6.2 The syntax for a record type also makes provisions for an optional variant part, started by the CASE reserved word. The variant part implies that a record type may be specified as consisting of several variants. This means that different variables, although said to be of the same type, may assume structures which differ in a certain manner. The differences may consist of a different number and different types of components. Each variant is characterised by a field of record type, according to which the variants are discriminated.

2.6.6.3 Syntax:

2.6.6.3.1
```
RecordType = RECORD [FixedPart][VariantPart] END

FixedPart = FieldList { [';'] FieldList}

FieldList  = [ IdentList ':' type ] | INLINE identdef ':' type

VariantPart   = CASE { ['|'] [INLINE] identdef ':' type }

IdentList  = identdef { [','] identdef }
```

2.6.6.4 Examples:

2.6.6.4.1
```
record

  day, month, year: integer

end
```

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

2.6.6.5 Fields of record type of the fixed part or the fields of the variant part can be prefixed by the INLINE (or abbreviated only IN) reserved word, in which case the name of the field or variant becomes transparent, and the fields of the corresponding record type are directly embedded in the enclosing record.

2.6.6.6 Each field or variant of a record must have a name which is unique within the record; if a field of a record embedded with the INLINE prefix has the same name as a field or variant of the embedding record, then the name of the prefixed field or variant must be used to access it.

### 2.6.7 Pointer types

2.6.7.1 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 can be of any type.

2.6.7.2 Syntax:

2.6.7.2.1
```
PointerType = ( POINTER TO | '^' ) type
```

2.6.7.3 If p is a variable of type `P = POINTER TO T`, a call of the predeclared procedure `NEW(p)` (see 6 Predeclared Procedure Reference) 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 open array type, the allocation has to be done with `NEW(p, e)` where T is allocated with the length given by the expression e. In either case a pointer to the allocated variable is assigned to `p`. `p` is of type P. The referenced variable `p^` is of type T. Any pointer variable may assume the value NIL, which points to no variable at all. All pointer fields or elements of a newly allocated record or array are set to NIL. If an allocated record or array is no longer used, it has to be explicitly deallocated with `DISPOSE(p)`.

### 2.6.8 Procedure types

2.6.8.1 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 and result types (see 2.11.10 Formal parameters) of P and T must *match* (see 3 Definition of terms). A procedure P assigned to a variable or a formal parameter must not be a predeclared, nor an inlined or const procedure, nor may it access local variables or parameters declared in outer procedures, or call procedures which access local variables or parameters declared in outer procedures.

2.6.8.2 Syntax:

2.6.8.2.1
```
ProcedureType = PROCEDURE [FormalParameters]
```

2.6.8.3 **NOTE** Traditional Oberon versions don't support assignment of procedures local to another procedure to a procedure type variable. Oberon+ as well as Micron don't make this restriction, as long as the local procedure (or one of its nested procedures) isn't nested and doesn't depend on local variables or parameters declared in its enclosing procedure.

### 2.6.9 Enumeration and symbol types

2.6.9.1 An enumeration is a list of identifiers that denote the elements which constitute the type. They, and no other values, belong to this type. These identifiers are either constants or symbols. Accordingly, the enumeration is either a constant or a symbol enumeration.

2.6.9.2 The export mark of the enumeration type declaration applies to each element of the enumeration.

2.6.9.3 Syntax:

2.6.9.3.1
```
enumeration ::= '(' ( constEnum | symbolEnum ) ')'
constEnum ::= ident [ '=' ConstExpression ] { [','] ident }
symbolEnum ::= ':' ident { [','] ':' ident }
```

2.6.9.4 The values of an enumeration are ordered, and the ordering relation is defined by their sequence in the enumeration. If T is an enumeration type then `MIN(T)` returns the first and `MAX(T)` the last element of the enumeration.

2.6.9.5 The ordinal number of a constant enumeration element can be obtained using the `ORD` predeclared procedure. `CAST` is the reverse operation. The ordinal number of the first element is determined by the optional constant expression, or 0 by default. `INC` returns the next and `DEC` the previous element. `INC(MAX(T))` and `DEC(MIN(T))` are undefined and terminate the program.

2.6.9.6 Examples:

2.6.9.6.1
```
(red, green, blue)

(club, diamond, heart, spade)

(Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)
```

2.6.9.7 The elements of a symbol enumeration are of the predeclared type SYMBOL. Values of the SYMBOL type have a runtime representation and can be stored in variables of SYMBOL type, just like integers. An invalid symbol has the value NIL. An ident of a symbol enumeration is declared the first time it appears in the source text. It can reappear later in the source text in another symbol enumeration declaration; in that case, both idents represent the same symbol. Elements of symbol enumerations can be assigned to variables of SYMBOL type. Variables of SYMBOL type are initialized with NIL.

2.6.9.8 TBD: symbol enumerations support the IN operator the check wheter a given value of SYMBOL type is part of the enum; this could be used to simulate inheritance relations.

2.6.9.9 TBD: do we need INC and DEC for the elements of symbol enumerations?

## 2.7 Variable declarations

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

2.7.2 Syntax:

2.7.2.1
```
VariableDeclaration = IdentList ":" type
```

2.7.3 Examples:

2.7.3.1
```
i, j, k: integer

x, y: real

p, q: bool

s: set

F: Function

a: array 100 of real

w: array 16 of record

     name: arra 32 of char

     count: integer

   end

t, c: Tree
```

## 2.8 Expressions

2.8.1 Expressions are constructs denoting rules of computation whereby constants and current values of variables are combined to compute other values by the application of operators and function procedures. Expressions consist of operands and operators. Parentheses may be used to express specific associations of operators and operands.

### 2.8.2 Operands

2.8.2.1 With the exception of constructors and literal constants (numbers, character constants, or strings), operands are denoted by designators. A designator consists of an identifier referring to a constant, variable, or procedure. This identifier may possibly be qualified by a module identifier (see 2.4 Declarations and scope rules and 2.12 Modules) and may be followed by selectors if the designated object is an element of a structure.

2.8.2.2 Syntax:

2.8.2.2.1
```
designator = qualident {selector}

selector   = '.' ident | '[' expression ']' | '^' | '(' qualident ')'

ExpList    = expression {',' expression}
```

2.8.2.3 If `a` designates an array, then `a[e]` denotes that element of `a` whose index is the current value of the expression `e`. The type of `e` must be an *integer type*.

2.8.2.4 If `r` designates a record, then `r.f` denotes the field `f` of `r`. If `p` designates a pointer, `p^` denotes the variable which is referenced by `p`. The designators `p^.f` and `p^[e]` may be abbreviated as `p.f` and `p[e]`, i.e. record and array selectors imply dereferencing.

2.8.2.5 Dereferencing is also implied if a pointer is assigned to a variable of a record or array type, if a pointer is used as actual parameter for a formal parameter of a record or array type, or if a pointer is used as argument of the predeclared procedure LEN().

2.8.2.6 If `a` or `r` are read-only, then also `a[e]` and `r.f` are read-only.

2.8.2.7 A type cast `v(T)` converts the original type of `v` to T. Within the designator, `v` is then regarded as having the type T instead of the original type. The cast is applicable, if `v` is a pointer to record type.

2.8.2.8 If the designated object is a constant or a variable, then the designator refers to its current value. If it is a procedure, the designator refers to that procedure unless it is followed by a (possibly empty) parameter list in which case it implies an

activation of that procedure and stands for the value resulting from its execution. The actual parameters must correspond to the formal parameters as in proper procedure calls (see 2.11.10 Formal parameters).

2.8.2.9 Examples:

2.8.2.9.1

```
i

a[i]

w[3].name[i]

t.left.right

t(CenterTree).subnode
```

### 2.8.3 Operators

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

2.8.3.2 Syntax:

2.8.3.2.1

```
expression       = SimpleExpression [ relation SimpleExpression ]
relation         = '=' | '#' | '<' | '<=' | '>' | '>=' | IN
SimpleExpression = ['+' | '-'] term { AddOperator term }
AddOperator      = '+' | '-' | OR
term             = factor {MulOperator factor}
MulOperator      = '*' | '/' | DIV | MOD | '&'
literal          = number | string | hexstring | hexchar
                   | NIL | TRUE | FALSE | set
constructor      = NamedType '{' [ component {',' component} ] '}'
component        = ConstExpression
factor           = literal | designator [ActualParameters]
                | '(' expression ')' | '~' factor
                | '@' designator | constructor
ActualParameters = '(' [ ExpList ] ')'
set              = '{' [ element {',' element} ] '}'
element          = expression ['..' expression]
```

#### 2.8.3.3 Logical operators

2.8.3.3.1

| | | | |
|---|---|---|---|
| OR | logical disjunction | `p or q` | *if p then TRUE, else q* |
| & | logical conjunction | `p & q` | *if p then q, else FALSE* |
| ~ | negation | ~p | *not p* |

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

#### 2.8.3.4 Arithmetic operators

2.8.3.4.1

| | |
|---|---|
| + | sum |
| - | difference |
| * | product |
| / | real quotient |
| DIV | integer quotient |
| | |

| | |
|---|---|
| MOD | modulus |

2.8.3.4.2 The operators +, -, *, and / apply to operands of numeric types.

2.8.3.4.3 When used as a binary operator, both operands must be of either of signed integer, unsigned integer or floating point type.

2.8.3.4.4 **NOTE** This is the same rule as in Modula-2, *where both operands must be either of type CARDINAL [...], in which case the result is of the type CARDINAL, or they must both be of type INTEGER [...], in which case the result is of type INTEGER*. The Ada language has a similar rule. The rules of the C language in contrast are much more complicated (see e.g. here), but it is always possible to add explicit conversions when migrating C to Micron.

2.8.3.4.5 The type of the result is the type of that operand which includes the type of the other operand (see 2.6.4.7), except for division (/), where the result is the smallest Floating-point types which includes both operand types

2.8.3.4.6 The operators `DIV` and `MOD` apply to integer operands only. They are related by the following formulas defined for any `x` and positive divisors `y`:

2.8.3.4.6.1
```
x = (x DIV y) * y + (x MOD y)

0 <= (x MOD y) < y
```

2.8.3.4.7 When used as monadic (unary) operators, '-' denotes sign inversion and '+' denotes the identity operation. If applied to an operand of unsigned integer type, the result is of the signed integer type with the same byte width as the operand. Otherwise the type of the result is the same as the type of the operand.

2.8.3.4.8 Examples:

2.8.3.4.8.1
```
x    y    x DIV y    x MOD y

5    3    1          2

-5   3    -2         1
```

2.8.3.4.9 **NOTE** Micron doesn't require overflow checks. If the representation of the result of an arithmetic operation would require a wider integer type than provided by the type of the expression, the behaviour is undefined; e.g. `MAX(INTEGER)+1` causes an overflow, i.e. the result could be MIN(INTEGER) or anything else.

### 2.8.3.5 Set Operators

2.8.3.5.1

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

2.8.3.5.2 Set operators apply to operands of type SET and yield a result of type SET. The monadic minus sign denotes the complement of `x`, i.e. `-x` denotes the set of integers between 0 and `MAX(SET)` which are not elements of `x`. Set operators are not associative (`(a+b)-c # a+(b-c)`).

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

### 2.8.3.6 Relations

2.8.3.6.1

| | |
|---|---|
| = | equal |
| # | unequal |
| < | less |
| <= | less or equal |
| > | greater |
| >= | greater or equal |
| IN | set membership |

2.8.3.6.2 Relations yield a BOOLEAN result. The relations =, #, <, <=, >, and >= apply to the numeric types, as well as enumerations, CHAR, strings, and CHAR arrays containing `0x` as a terminator.

2.8.3.6.3 The relations = and # also apply to BOOLEAN, SYMBOL and SET, as well as to pointer and procedure types.

2.8.3.6.4 `x IN s` stands for *x is an element of s*. `x` must be of an integer type, and `s` of type SET.

2.8.3.6.5 Examples:

2.8.3.6.5.1
```
1991                    // integer
i div 3                 // integer
~p or q                 // boolean
(i+j) * (i-j)           // integer
s - {8, 9, 13}          // set
i + x                   // real
a[i+j] * a[i-j]         // real
(0<=i) & (i<100)        // boolean
t.key = 0               // boolean
k in {i..j-1}           // boolean
w[i].name <= "John"     // boolean
t is CenterTree         // boolean
```

### 2.8.3.7 String operators

2.8.3.7.1

| | |
|---|---|
| + | concatenation |

2.8.3.7.2 The concatenation operator applies to string and character literals. The resulting string consists of the characters of the first operand followed by the characters of the second operand.

### 2.8.3.8 Address operator

2.8.3.8.1 The address operator '@' can be applied to a variable, record field or array element, which is visible and not read-only, and yields the address of the object.

2.8.3.8.2 If `v` designates a variable of type T, then `@v` denotes the address of `v` with the type `POINTER TO T.`

2.8.3.8.3 If `r.f` designates a record field `f` of type T, then `@r.f` denotes the address of field `f` with the type `POINTER TO T`.

2.8.3.8.4 If `a[e]` designates an array element of type T at index `e`, then `@a[e]` denotes the address of the element at index `e`; the address is of type `POINTER TO T`.

### 2.8.3.9 Function Call

2.8.3.9.1 A function call is a factor in an expression. In contrast to 2.9.6 Procedure calls in a function call the actual parameter list is mandatory. Each expression in the actual parameters list (if any) is used to initialize a corresponding formal parameter. The number of expressions in the actual parameter list must correspond the number of formal parameters. See also 2.11.10 Formal parameters.

2.8.3.9.2 Syntax:

2.8.3.9.2.1
```
FunctionCall            = designator ActualParameters
```

## 2.8.4 Constructors

2.8.4.1 With constructors, record, array and pointer literals can be declared.

2.8.4.2 Syntax:

2.8.4.2.1
```
constructor ::= NamedType '{' [ component {',' component} ] '}'
component ::= ConstExpression
```

2.8.4.3 If NamedType is a record type, then there is a component for each field of the record in the order of declaration.

2.8.4.4 If NamedType is an array type, then there is a componend for each element of the array. The array type may be an open array in which case the number of elements is determined by the number of components.

2.8.4.5 If NamedType is a pointer type, then there is exactly one component which is an unsigned integer type constant representing the address.

2.8.4.6 For each field or element which is of record, array or pointer type, an embedded constructor is declared. Since the exact type of the field or element is known, the NamedType prefix can be left out.

2.8.4.7 Example:

2.8.4.7.1
```
myVal := Rect{0,0,x1,y1};
```

# 2.9 Statements

2.9.1 Statements denote actions. There are elementary and structured statements. Elementary statements are not composed of any parts that are themselves statements. They are the assignment, the procedure call, the return, and the `exit` statement. Structured statements are composed of parts that are themselves statements. They are used to express sequencing and conditional, selective, and repetitive execution.

2.9.2 Syntax:

2.9.2.1
```
statement = [ assignment | ProcedureCall | IfStatement
            | CaseStatement  | WithStatement | LoopStatement
            | ExitStatement | GotoStatement | gotoLabel
        | ReturnStatement | RepeatStatement | ForStatement ]
```

### 2.9.3 Statement sequences

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

2.9.3.2 Syntax:

2.9.3.2.1
```
StatementSequence = statement { [";"] statement}
```

### 2.9.4 Statement block

2.9.4.1 A statement sequence enclosed between a BEGIN and END reserved word pair is called a statement block. Statement blocks appear on module level and in procedure bodies.

2.9.4.2 Syntax:

2.9.4.2.1
```
block = BEGIN StatementSequence END
```

### 2.9.5 Assignments

2.9.5.1 Assignments replace the current value of a variable by a new value specified by an expression. The expression must be *assignment compatible* with the variable (see 3 Definition of terms). The assignment operator is written as `:=` and pronounced as *becomes*.

2.9.5.2 Syntax:

2.9.5.2.1
```
assignment = designator ':=' expression
```

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

2.9.5.3.1 if $T_v$ and $T_e$ are the same record types, all fields of $T_e$ are assigned to the corresponding fields of $T_v$.

2.9.5.3.2 if $T_v$ is `ARRAY n OF CHAR` and e is a string of length m < n, `v[i]` becomes $e_i$ for i = 0..m-1 and `v[m]` becomes 0X;

2.9.5.3.3 if $T_v$ and $T_e$ are open or non-open CHAR arrays, `v[i]` becomes `e[i]` for i = 0..STRLEN(e); if LEN(v) <= STRLEN(e) or e is not terminated by 0X the program halts;

2.9.5.3.4 if $T_v$ is an open CHAR array and e is a string `v[i]` becomes `e[i]` for i = 0..LEN(e)-1 and `v[LEN(e)]` becomes 0X; if LEN(v) <= LEN(e) the program halts.

2.9.5.4 Examples:

2.9.5.4.1
```
i := 0
p := i = j
x := i + 1
k := log2(i+j)
F := log2
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y) * (x-y)
t.key := i
w[i+1].name := "John"
t := c
```

### 2.9.6 Procedure calls

2.9.6.1 A procedure call activates a procedure. It may contain a list of actual parameters which replace the corresponding formal parameter list defined in the procedure declaration (see 2.11 Procedure declarations). The correspondence is established

by the positions of the parameters in the actual and formal parameter lists.

2.9.6.2 Each actual parameter must be an expression. This expression is evaluated before the procedure activation, and the resulting value is assigned to the formal parameter (see also 2.11.10 Formal parameters).

2.9.6.3 Syntax:

2.9.6.3.1
```
ProcedureCall = designator [ ActualParameters ]
```

2.9.6.4 Examples:

2.9.6.4.1
```
WriteInt(i*2+1)

inc(w[k].count)

t.Insert("John")
```

### 2.9.7 If statements

2.9.7.1 If statements specify the conditional execution of guarded statement sequences. The boolean expression preceding a statement sequence is called its guard. The guards are evaluated in sequence of occurrence, until one evaluates to TRUE, whereafter its associated statement sequence is executed. If no guard is satisfied, the statement sequence following ELSE is executed, if there is one.

2.9.7.2 Syntax:

2.9.7.2.1
```
IfStatement    = IF expression THEN StatementSequence

             {ElsifStatement} [ElseStatement] END

ElsifStatement = ELSIF expression THEN StatementSequence

ElseStatement  = ELSE StatementSequence
```

2.9.7.3 Example:

2.9.7.3.1
```
if (ch >= "A") & (ch <= "Z") then ReadIdentifier

elsif (ch >= "0") & (ch <= "9") then ReadNumber

elsif (ch = "'") OR (ch = '"') then ReadString

else SpecialCharacter

end
```

### 2.9.8 Case statements

2.9.8.1 Case statements specify the selection and execution of a statement sequence according to the value of an expression. First the case expression is evaluated, then that statement sequence is executed whose case label list contains the obtained value. The case expression must either be of an integer type that includes the types of all case labels, or an enumeration type with all case labels being valid members of this type, or both the case expression and the case labels must be of type CHAR. Case labels are constants, and no value must occur more than once. If the value of the expression does not occur as a label of any case, the statement sequence following ELSE is selected, if there is one, otherwise the program is aborted.

2.9.8.2 Syntax:

2.9.8.2.1
```
CaseStatement = CASE expression OF ['|'] Case { '|' Case }

            [ ELSE StatementSequence ] END

Case          = [ CaseLabelList ':' StatementSequence ]

CaseLabelList = LabelRange { ',' LabelRange }

LabelRange    = label [ '..' label ]

label         = ConstExpression
```

2.9.8.3 Examples:

2.9.8.3.1
```
case ch of

  "A" .. "Z": ReadIdentifier

| "0" .. "9": ReadNumber

| "'", '"': ReadString
```

```
else SpecialCharacter
end
```

### 2.9.9 While statements

2.9.9.1 While statements specify the repeated execution of a statement sequence while the Boolean expression (its guard) yields TRUE. The guard is checked before every execution of the statement sequence. The ELSIF part is integrated in the loop; as long as any of the Boolean expressions (either the WHILE or ELSIF guard) yields TRUE, the corresponding statement sequence is executed; repetition only terminates, when all guards are FALSE.

2.9.9.2 Syntax:

2.9.9.2.1
```
WhileStatement = WHILE expression DO StatementSequence
             {ELSIF expression DO StatementSequence} END
```

2.9.9.3 Examples:

2.9.9.3.1
```
// Euclidean algorithm to compute the greatest common divisor of m and n:
while m > n do
        m := m – n
elsif n > m do
        n := n – m
end
// is equivalent to:
loop
        if m > 0 then
                m := m – n
        elsif n > m then
                n := n – m
        else
                exit
        end
end
```

2.9.9.4 **NOTE** The ELSIF part was added to Oberon-07. It is noteably Dijkstra's form of the WHILE loop. Contrary to intuition, the ELSIF part is not executed only if the first check of the WHILE guard evaluates to FALSE; instead, both parts are checked and executed until both guards evaluate to FALSE.

### 2.9.10 Repeat statements

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

2.9.10.2 Syntax:

2.9.10.2.1
```
RepeatStatement = REPEAT StatementSequence UNTIL expression
```

### 2.9.11 For statements

2.9.11.1 A for statement specifies the repeated execution of a statement sequence while a progression of values is assigned to a control variable of the for statement. Control variables can be of integer or enumeration types. An explicit BY expression is only supported for integer control variables.

2.9.11.2 Syntax:

2.9.11.2.1
```
ForStatement = FOR ident ':=' expression TO expression
           [BY ConstExpression]
           DO StatementSequence END
```

2.9.11.3 The statement

2.9.11.3.1
```
for v := first to last by step do statements end
```

2.9.11.4 is equivalent to

2.9.11.4.1
```
temp := last; v := first
if step > 0 then
    while v <= temp do statements; INC(v,step) end
else
    while v >= temp do statements; DEC(v,-step) end
end
```

2.9.11.5 temp has the same type as v. For integer control variables, step must be a nonzero constant expression; if step is not specified, it is assumed to be 1. For enumeration control variables, there is no explicit step, but the INC or DEC version of the while loop is used depending on ORD(first) ⇐ ORD(last).

2.9.11.6 Examples:

2.9.11.6.1
```
for i := 0 to 79 do k := k + a[i] end
for i := 79 to 1 by -1 do a[i] := a[i-1] end
```

### 2.9.12 Loop statements

2.9.12.1 A loop statement specifies the repeated execution of a statement sequence. It is terminated upon execution of an exit statement within that sequence (see 2.9.13 Return, exit and goto statements).

2.9.12.2 Syntax:

2.9.12.2.1
```
LoopStatement = LOOP StatementSequence END
ExitStatement = EXIT
```

2.9.12.3 Example:

2.9.12.3.1
```
loop
  ReadInt(i)
  if i < 0 then exit end
  WriteInt(i)
end
```

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

### 2.9.13 Return, exit and goto statements

2.9.13.1 A return statement indicates the termination of a procedure. It is denoted by RETURN, followed by an expression if the procedure is a function procedure. The type of the expression must be assignment compatible (see 3 Definition of terms) with the result type specified in the procedure heading (see 2.11 Procedure declarations).

2.9.13.2 Syntax:

2.9.13.2.1
```
ReturnStatement = RETURN [ expression ]
```

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

2.9.13.4 **NOTE** The optional expression causes an LL(k) ambiguity which can be resolved in that the parser expects a return expression if the procedure has a return type and vice versa.

2.9.13.5 An exit statement is denoted by EXIT. It specifies termination of the enclosing loop statement and continuation with the statement following that loop statement. Exit statements are contextually, although not syntactically associated with the loop statement which contains them.

2.9.13.6 Syntax:

2.9.13.6.1
```
ExitStatement   = EXIT
```

2.9.13.7 The goto statement causes an unconditional jump to another statement in the same block (see 2.9.4 Statement block). The destination of the jump is specified by the name of a label. Goto labels are declared in the statement sequence and are referenced in the goto statements. It is illegal to define a label that is never used. Goto labels do not conflict with identifiers that are not labels. The scope of a goto label is the enclosing block.

2.9.13.8 A goto statement cannot jump into a structured statement or outside of the enclosing block.

2.9.13.9 Syntax:

2.9.13.9.1
```
GotoStatement = 'GOTO' ident

gotoLabel = ident ':'
```

2.9.13.10 **NOTE** Goto statements are considered bad practice and should be avoided as much as possible. They were mostly added to the language to ease migration of existing C code. It is always possible to replace a goto statement by an alternative implementation that doesn't need a goto.

## 2.10 Exception handling

2.10.1 Exception handling in Micron is implemented using the predeclared procedures PCALL and RAISE (see 6 Predeclared Procedure Reference), without any special syntax. There are no predefined exceptions.

2.10.2 An exception is just a value of type SYMBOL The symbol representing the exception is passed as an actual argument to RAISE. RAISE may be called without an argument in which case the compiler provides an internal symbol. RAISE never returns, but control is transferred from the place where RAISE is called to the nearest dynamically-enclosing call of PCALL. When calling RAISE without a dynamically-enclosing call of PCALL the program execution is aborted.

2.10.3 PCALL executes a protected call of the procedure or procedure type P. P is passed as the second argument to PCALL. P cannot have a return type. P can be a nested procedure, even if it accesses local variables or parameters of an outer procedure. If P has formal parameters the corresponding actual parameters are passed to PCALL immediately after P. The actual parameters must be *parameter compatible* with the formal parameters of P (see 3 Definition of terms). The first parameter R of PCALL is a variable of type SYMBOL; if RAISE(E) is called in the course of P, then R is set to E; otherwise R is set to NIL. The state of VAR parameters of P or local variables or parameters of an outer procedure accessed by P is non-deterministic in case RAISE is called in the course of P.

2.10.4 Example:

2.10.4.1
```
module ExceptionExample
  type Exception = record end
  proc Print(in str: array of char)
    var e: pointer to Exception
  begin
    println(str)
    new(e)
    raise(e)
    println("this is not printed")
  end Print
  var res: pointer to anyrec
begin
  pcall(res, Print, "Hello World")
  case res of
  | Exception: println("got Exception")
  | anyrec: println("got anyrec")
  | nil: println("no exception")
  else
    println("unknown exception")
    // could call raise(res) here to propagate the exception
  end
end ExceptionExample
```

## 2.11 Procedure declarations

2.11.1 A procedure declaration consists of a procedure heading and a procedure body. The heading specifies the procedure identifier and the formal parameters (see 2.11.10 Formal parameters). The body contains declarations and statements. The procedure identifier must be repeated at the end of the procedure declaration unless it has no body.

2.11.2 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. Each control path of a function procedure must return a value.

2.11.3 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.

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

2.11.5 **NOTE** Procedures can be nested, and inner procedures have access to the parameters or local variables of outer procedures ("non-local access"). This feature was already supported in ALGOL 60 and adopted by Wirth in Pascal; it is also supported by Oberon, Oberon-2, and Oberon+, but not by Oberon 07.

2.11.6 A procedure body may have no statements in which case the ident after the END reserved word can also be left out; in a function procedure with no statements a return statement with a default value is assumed.

2.11.7 A procedure can be declared INLINE (or abbreviated IN) in which case the code of the procedure is embedded at the call site, and no call actually happens. Procedures declared INLINE don't support recursion, neither directly nor indirectly.

2.11.8 A procedure can be declared CONST in which case it can be used in CONST declarations to derive a constant value at compile time. CONST procedures can only call other CONST procedures or predeclared procedures not depending on runtime information (TBD: why not all predeclareds?). CONST procedures cannot access module variables, but they can access CONST declarations in the same or other modules.

2.11.9 Syntax:

2.11.9.1
```
ProcedureDeclaration = ProcedureHeading [';'] ProcedureBody END [ ident ]

ProcedureHeading     = ( PROCEDURE | PROC ) [ INLINE | CONST ]

                     identdef [ FormalParameters ]

ProcedureBody        = DeclarationSequence

                       [ BEGIN StatementSequence

                       | ReturnStatement [';'] ]

DeclarationSequence  = { CONST { ConstDeclaration [';'] }

                     | TYPE { TypeDeclaration [';'] }

                     | VAR { VariableDeclaration [';'] }

                     | ProcedureDeclaration [';'] }
```

### 2.11.10 Formal parameters

2.11.10.1 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.

2.11.10.2 **NOTE** Oberon and Oberon+ support VAR parameters, which are not necessary in Micron, because the formal parameter can be a pointer instead and the '@' operator can be applied to the designator passed as actual parameter.

2.11.10.3 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 cannot be an open array.

2.11.10.4 Syntax:

2.11.10.4.1
```
FormalParameters = '(' [ FPSection { [';'] FPSection } [ [';'] '..'] ] ')'

                   [ ':' ReturnType ]

ReturnType       = [ POINTER TO | '^' ] NamedType

FPSection        = ident { [','] ident } ':' FormalType

FormalType       = type
```

2.11.10.5 Module level procedures can have a variable number of parameters, which is indicated by `'..'` (ellipsis). In that case the additional parameters (called "variable parameters") can be accessed in the procedure body using the VARARG() and VARARGS() predeclared procedures. Only actual parameters of basic or pointer type (including enumerations and procedure types) can be passed as variable parameters.

2.11.10.6 **NOTE** Nested procedures cannot have a variable number of arguments because this would interfere with the access to non-local variables and parameters, see 2.11 Procedure declarations.

2.11.10.7 Examples:

2.11.10.7.1
```
proc ReadInt(x: pointer to integer)
```

```
  var i: integer; ch: char
begin i := 0; Read(ch)
  while ("0" <= ch) & (ch <= "9") do
    i := 10*i + (ord(ch)-ord("0")); Read(ch)
  end
  x^ := i
end ReadInt
```

### 2.11.11 Predeclared procedures

2.11.11.1 Predeclared procedures are provided by the compiler and accessible without extra imports. Some are generic procedures, i.e. they apply to several types of operands.

2.11.11.2 See 6 Predeclared Procedure Reference for the list of predeclared procedures each Micron compiler has to provide.

## 2.12 Modules

2.12.1 A module is a collection of declarations of constants, types, variables, and procedures, together with a sequence of statements for the purpose of assigning initial values to the variables. A module constitutes a text that is compilable as a unit (compilation unit).

2.12.2 Syntax:

2.12.2.1

```
module     = MODULE ident [ MetaParams ] [';']
               { ImportList | DeclarationSequence }
           [ BEGIN StatementSequence ] END ident ['.']
ImportList = IMPORT import { [','] import } [';']
import     = [ ident ':=' ] ImportPath ident [ MetaActuals ]
ImportPath = { ident '.' }
```

2.12.3 The import list specifies the names of the imported modules. If a module A is imported by a module M and A exports an identifier x, then x is referred to as `A.x` within M.

2.12.4 If A is imported as `B := A`, the object x must be referenced as `B.x`. This allows short alias names in qualified identifiers.

2.12.5 In Micron, as in Oberon+, the import can refer to a module by means of a module name optionally prefixed with an import path. There is no requirement that the import path actually exists in the file system, or that the source files corresponding to an import path are in the same file system directory. It is up to the compiler how source files are mapped to import paths. An imported module with no import path is first looked up in the import path of the importing module.

2.12.6 A module must not import itself (neither directly nor indirectly).

2.12.7 Identifiers that are to be exported (i.e. that are to be visible in client modules) must be marked by an export mark in their declaration (see Chapter 2.4 Declarations and scope rules).

2.12.8 The statement sequence following the symbol BEGIN is executed when the module is loaded, which is done after the imported modules have been loaded. It follows that cyclic import of modules is illegal.

## 2.13 Generics

2.13.1 Micron, as Oberon+, supports generic programming. Modules can be made generic by adding formal meta parameters. Meta parameters represent types or constants; the latter include procedures. Meta parameters default to types, but can be explicitly prefixed with the TYPE reserved word; the CONST prefix designates a constant meta parameter. A meta parameter can be constrained with a named type, in which case the actual meta parameter must correspond to this type; the correspondence is established when the generic module is instantiated; the type of the actual meta parameter must be assignment compatible with the constraint type (see 3 Definition of terms).

2.13.2 Generic modules can be instantiated with different sets of meta actuals which enables the design of reusable algorithms and data structures. The instantiation of a generic module occurs when importing it. A generic module can be instantiated more than once in the same module with different actual meta parameters. See also 2.12 Modules.

2.13.3 Syntax:

2.13.3.1

```
MetaParams       = '(' MetaSection { [';'] MetaSection } ')'
MetaSection      = [ TYPE | CONST ] ident { [','] ident } [ ':' TypeConstraint ]
TypeConstraint   = NamedType
MetaActuals      = '(' ConstExpression { [','] ConstExpression } ')'
```

2.13.4 Meta parameters can be used within the generic module like normal types or constants. If no type constraint is present, the types and constants can be used wherever no information about the actual type is required; otherwise the type constraint

determines the permitted operations. The rules for *same types* and *equal types* apply analogously to meta parameters, and subsequently also the corresponding assignment, parameter and array compatibility rules.

## 2.14 Source code directives

2.14.1 Source code directives are used to set configuration variables in the source text and to select specific pieces of the source text to be compiled (conditional compilation). Oberon+ uses the syntax recommended in 7.2 Kirk, B. et al. (1995). The Oakwood Guidelines....

### 2.14.2 Configuration Variables

2.14.2.1 Configuration variables can be set or unset in the source code using the following syntax:

2.14.2.2 Syntax:

2.14.2.2.1
```
directive = '<*' ident ( '+' | '-' ) '*>'
```

2.14.2.3 Each variable is named by an ident which follows the syntax specified in 2.3.3 Identifiers. Variable names have compilation unit scope which is separate from all other scopes of the program. Configuration variable directives can be placed anywhere in the source code. The directive only affects the present compilation unit, starting from its position in the source code.

2.14.2.4 Example:

2.14.2.4.1
```
<* WIN32+ *>
<* WIN64- *>
```

2.14.2.5 **NOTE** Usually the compiler provides the possibility to set configuration variables, e.g. via command line interface.

2.14.2.6 TBD: import configuration files with a list of variables and values for project wide configuration

### 2.14.3 Conditional compilation

2.14.3.1 Conditional compilation directives can be placed anywhere in the source code. The following syntax applies:

2.14.3.2 Syntax:

2.14.3.2.1
```
directive  = '<*' [ scIf | scElsif | scElse | scEnd ] '*>'
scIf       = IF scExpr THEN
scElsif    = ELSIF condition THEN
scElse     = ELSE
scEnd      = END
condition  = scTerm { OR scTerm }
scTerm     = scFactor {'&' scFactor}
scFactor   = ident | '(' condition ')' | '~' scFactor
```

2.14.3.3 An ELSIF or ELSE directive must be preceded by an IF or another ELSIF directive. Each IF directive must be ended by an END directive. The directives form sections of the source code. Only the section the condition of which is TRUE (or the section framed by ELSE and END directive otherwise) is visible to the compiler. Conditions are boolean expressions. Ident refers to a configuration variable. When a configuration variable is not explicitly set it is assumed to be FALSE. Each section can contain nested conditional compilation directives.

2.14.3.4 Example:

2.14.3.4.1
```
<* if A then *>
  println("A")
<* elsif B & ~C then *>
  println("B & ~C")
<* else *>
  println("D")
<* end *>
```

# 3 Definition of terms

## 3.1 Integer types

3.1.1 Signed integer types, Unsigned integer types

## 3.2 Signed integer types

3.2.1 INT8, INT16, INT32, INT64, SHORTINT, INTEGER, LONG...

**3.3 Unsigned integer types**

3.3.1 BYTE, UINT8, UINT16, UINT32, UINT64

**3.4 Floating-point types**

3.4.1 REAL, LONGREAL, F32, F64

**3.5 Numeric types**

3.5.1 Integer types, Floating-point types

**3.6 Same types**

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

3.6.1.1 $T_a$ and $T_b$ are both denoted by the same type identifier, or

3.6.1.2 $T_a$ is declared to equal $T_b$ in a type declaration of the form $T_a = T_b$, or

3.6.1.3 a and b appear in the same identifier list in a variable, record field, or formal parameter declaration and are not open arrays.

**3.7 Equal types**

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

3.7.1.1 $T_a$ and $T_b$ are the *same type*, or

3.7.1.2 $T_a$ and $T_b$ are open array types with *equal element types*, or

3.7.1.3 $T_a$ and $T_b$ are procedure types whose formal parameters *match*, or

3.7.1.4 $T_a$ and $T_b$ are pointer types with *equal* base types.

**3.8 Type inclusion**

3.8.1 Numeric types include (the values of) smaller numeric types. There is a separate type inclusion hierarchy for signed integeres, unsigned integers and floating point types. See 2.6.4 Basic types for more information.

**3.9 Assignment compatible**

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

3.9.1.1 $T_e$ and $T_v$ are the *same type*;

3.9.1.2 $T_e$ and $T_v$ are numeric types and $T_v$ *includes* $T_e$

3.9.1.3 $T_e$ and $T_v$ are pointer types and the pointers have *equal* base types;

3.9.1.4 $T_e$ and $T_v$ are pointer types and $T_v$ is a POINTER TO ANY;

3.9.1.5 $T_v$ is a pointer or a procedure type and e is NIL;

3.9.1.6 $T_v$ is an enumeration type and e is a valid element of the enumeration;

3.9.1.7 $T_e$ is an open array and $T_v$ is an array of *equal* base type;

3.9.1.8 $T_v$ is an array of CHAR, $T_e$ is a Latin-1 string literal or character array, and STRLEN(e) < LEN(v);

3.9.1.9 $T_v$ is a procedure type and e is the name of a procedure whose formal parameters *match* those of $T_v$.

**3.10 Parameter compatible**

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

3.10.1.1 $T_f$ and $T_a$ are *equal* types, or

3.10.1.2 $T_a$ is *assignment compatible* with $T_f$

**3.11 Array compatible**

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

3.11.1.1 $T_f$ and $T_a$ are the *equal type*, or

3.11.1.2 $T_f$ is an open array, $T_a$ is any array, and their element types are *array compatible*, or

3.11.1.3 $T_f$ is an open array of CHAR and $T_a$ is a Latin-1 string literal, or

3.11.1.4 $T_f$ is an open array of BYTE and $T_a$ is a hex string literal.

**3.12 Expression compatible**

3.12.1 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). CHAR arrays that are to be compared must contain 0X as a terminator.

3.12.2

| operator | first operand | second operand | result type |
|---|---|---|---|
| + - * | signed integer | signed integer | smallest signed integer type including both operands |
| | unsigned integer | unsigned integer | smallest unsigned integer type including both operands |
| | floating point | floating point | smallest floating point type including both operands |
| / | numeric | numeric | smallest floating point type type including both operands |
| + - * / | SET | SET | SET |
| DIV MOD | signed integer | signed integer | smallest signed integer type type including both operands |
| | unsigned integer | unsigned integer | smallest unsigned integer type type including both operands |
| OR & ~ | BOOLEAN | BOOLEAN | BOOLEAN |
| = # < <= > >= | signed integer | signed integer | BOOLEAN |
| | unsignednteger | unsigned integer | BOOLEAN |
| | floating point | floating point | BOOLEAN |
| | CHAR | CHAR | BOOLEAN |
| | CHAR array, string | CHAR array, string | BOOLEAN |
| = # | BOOLEAN | BOOLEAN | BOOLEAN |
| | SET | SET | BOOLEAN |
| | NIL, pointer type | NIL, pointer type | BOOLEAN |
| | procedure type, NIL | procedure type, NIL | BOOLEAN |
| IN | integer | SET | BOOLEAN |

## 3.13 Matching formal parameter lists

3.13.1 Two formal parameter lists match if

3.13.1.1 they have the same number of parameters, and

3.13.1.2 parameters at corresponding positions have *equal types*, and

3.13.1.3 both parameter lists either end with an ellipsis or not

## 3.14 Matching result types

3.14.1 The result types of two procedures match if they are either the *same type* or none.

# 4 Complete Micron Syntax

4.1 ↳

```
ident  = ( letter | '_' ) { letter | digit | '_' }

letter = 'A' ... 'Z' | 'a' ... 'z'

digit  = '0' ... '9'
```

4.2 ↳

```
number   = integer | real

integer  = (digit {digit} ['O'|'B' | digit {hexDigit} 'H')
           ['U8'|'U16'|'U32'|'U64']

real     = digit {digit} '.' {digit} [Exponent]
```

```
Exponent = ('E' | 'D' | 'F' ) ['+' | '-'] digit {digit}

hexDigit = digit | 'A' ... 'F'

digit    = '0' ... '9'
```

4.3 ↳

```
character = digit {hexDigit} ('X' | 'x')
```

4.4 ↳

```
string = ''' {character} ''' | '"' {character} '"'
```

4.5 ↳

```
hexstring = '$' {hexDigit} '$'
```

4.6 ↳

```
qualident = [ident '.'] ident

identdef  = ident ['*' | '-']
```

4.7 ↳

```
ConstDeclaration = identdef '=' ConstExpression

ConstExpression  = expression
```

4.8 ↳

```
TypeDeclaration = identdef '=' type

type            = NamedType | ArrayType | RecordType | PointerType | ProcedureType | enumeration

NamedType       = qualident
```

4.9 ↳

```
ArrayType  = ARRAY [ length ] OF type | '[' [ length ] ']' type

length     = ConstExpression | VAR varlength

varlength  = expression
```

4.10 ↳

```
RecordType = RECORD [FixedPart][VariantPart] END

FixedPart = FieldList { [';'] FieldList}

FieldList  = [ IdentList ':' type ] | INLINE identdef ':' type

VariantPart   = CASE { ['|'] [INLINE] identdef ':' type }

IdentList  = identdef { [','] identdef }
```

4.11 ↳

```
PointerType = ( POINTER TO | '^' ) type
```

4.12 ↳

```
ProcedureType = PROCEDURE [FormalParameters]
```

4.13 ↳

```
enumeration ::= '(' ( constEnum | symbolEnum ) ')'
constEnum ::= ident [ '=' ConstExpression ] { [','] ident }
symbolEnum ::= ':' ident { [','] ':' ident }
```

4.14 ↳

```
VariableDeclaration = IdentList ":" type
```

4.15 ↳

```
designator = qualident {selector}
```

```
selector   = '.' ident | '[' expression ']' | '^' | '(' qualident ')'
ExpList    = expression {',' expression}
```

4.16 ↳

```
expression       = SimpleExpression [ relation SimpleExpression ]
relation         = '=' | '#' | '<' | '<=' | '>' | '>=' | IN
SimpleExpression = ['+' | '-'] term { AddOperator term }
AddOperator      = '+' | '-' | OR
term             = factor {MulOperator factor}
MulOperator      = '*' | '/' | DIV | MOD | '&'
literal          = number | string | hexstring | hexchar
                   | NIL | TRUE | FALSE | set
constructor      = NamedType '{' [ component {',' component} ] '}'
component        = ConstExpression
factor           = literal | designator [ActualParameters]
                | '(' expression ')' | '~' factor
                | '@' designator | constructor
ActualParameters = '(' [ ExpList ] ')'
set              = '{' [ element {',' element} ] '}'
element          = expression ['..' expression]
```

4.17 ↳

```
FunctionCall           = designator ActualParameters
```

4.18 ↳ constructor ::= NamedType '{' [ component {',' component} ] '}'
component ::= ConstExpression

4.19 ↳

```
statement = [ assignment | ProcedureCall | IfStatement
            | CaseStatement  | WithStatement | LoopStatement
            | ExitStatement | GotoStatement | gotoLabel
        | ReturnStatement | RepeatStatement | ForStatement ]
```

4.20 ↳

```
StatementSequence = statement { [";"] statement}
```

4.21 ↳

```
assignment = designator ':=' expression
```

4.22 ↳

```
ProcedureCall = designator [ ActualParameters ]
```

4.23 ↳

```
IfStatement    = IF expression THEN StatementSequence
             {ElsifStatement} [ElseStatement] END
ElsifStatement = ELSIF expression THEN StatementSequence
ElseStatement  = ELSE StatementSequence
```

4.24 ↳

```
CaseStatement = CASE expression OF ['|'] Case { '|' Case }
            [ ELSE StatementSequence ] END
```

```
Case          = [ CaseLabelList ':' StatementSequence ]
CaseLabelList = LabelRange { ',' LabelRange }
LabelRange    = label [ '..' label ]
label         = ConstExpression
```

4.25 ↳

```
WhileStatement = WHILE expression DO StatementSequence
             {ELSIF expression DO StatementSequence} END
```

4.26 ↳

```
RepeatStatement = REPEAT StatementSequence UNTIL expression
```

4.27 ↳

```
ForStatement = FOR ident ':=' expression TO expression
           [BY ConstExpression]
           DO StatementSequence END
```

4.28 ↳

```
LoopStatement = LOOP StatementSequence END
ExitStatement = EXIT
```

4.29 ↳

```
ReturnStatement = RETURN [ expression ]
```

4.30 ↳

```
ExitStatement   = EXIT
```

4.31 ↳

```
GotoStatement = 'GOTO' ident
gotoLabel = ident ':'
```

4.32 ↳

```
ProcedureDeclaration = ProcedureHeading [';'] ProcedureBody END [ ident ]
ProcedureHeading     = ( PROCEDURE | PROC ) [ INLINE | CONST ]
                   identdef [ FormalParameters ]
ProcedureBody        = DeclarationSequence
                       [ BEGIN StatementSequence
                       | ReturnStatement [';'] ]
DeclarationSequence  = { CONST { ConstDeclaration [';'] }
                   | TYPE { TypeDeclaration [';'] }
                   | VAR { VariableDeclaration [';'] }
                   | ProcedureDeclaration [';'] }
```

4.33 ↳

```
FormalParameters = '(' [ FPSection { [';'] FPSection } [ [';'] '..'] ] ')'
                   [ ':' ReturnType ]
ReturnType       = [ POINTER TO | '^' ] NamedType
FPSection        = ident { [','] ident } ':' FormalType
FormalType       = type
```

4.34 ↳

```
module     = MODULE ident [ MetaParams ] [';']
               { ImportList | DeclarationSequence }
           [ BEGIN StatementSequence ] END ident ['.']
ImportList = IMPORT import { [','] import } [';']
import     = [ ident ':=' ] ImportPath ident [ MetaActuals ]
ImportPath = { ident '.' }
```

4.35 ↳

```
MetaParams       = '(' MetaSection { [';'] MetaSection } ')'
MetaSection      = [ TYPE | CONST ] ident { [','] ident } [ ':' TypeConstraint ]
TypeConstraint   = NamedType
MetaActuals      = '(' ConstExpression { [','] ConstExpression } ')'
```

4.36 ↳

```
directive = '<*' ident ( '+' | '-' ) '*>'
```

4.37 ↳

```
directive  = '<*' [ scIf | scElsif | scElse | scEnd ] '*>'
scIf       = IF scExpr THEN
scElsif    = ELSIF condition THEN
scElse     = ELSE
scEnd      = END
condition  = scTerm { OR scTerm }
scTerm     = scFactor {'&' scFactor}
scFactor   = ident | '(' condition ')' | '~' scFactor
```

# 5 Predeclared Types

## 5.1 ANY

5.1.1 Opaque type only usable as a pointer base type which is assignment compatible with all pointer types, like void* in C.

## 5.2 BOOLEAN

5.2.1 see 2.6.4 Basic types.

## 5.3 BYTE, UINT8, UINT16, UINT32, UINT64

5.3.1 Unsigned integer types, see 2.6.4 Basic types.

## 5.4 CHAR

5.4.1 see 2.6.4 Basic types.

## 5.5 INT8, INT16, INT32, INT64, SHORTINT, INTEGER, LONGINT

5.5.1 Signed integer types, see 2.6.4 Basic types.

## 5.6 REAL, LONGREAL, F32, F64

5.6.1 Floating-point types, see 2.6.4 Basic types.

## 5.7 SET

5.7.1 see 2.6.4 Basic types.

## 5.8 SYMBOL

5.8.1 Opaque type representing an internalized identifier. Values of the type exist at runtime and can be NIL. See 2.6.9 Enumeration and symbol types.

## 5.9 VA_LIST

5.9.1 Opaque type representing the list of variable parameters, accessible by <null reference>, see 2.11.10 Formal parameters.

# 6 Predeclared Procedure Reference

## 6.1 Predeclared function procedures

6.1.1

| Name | Argument type | Result type | Function |
|---|---|---|---|
| ABS(x) | numeric type | type of x | absolute value |
| CAP(x) | CHAR | CHAR | corresponding capital letter (only for the ASCII subset of the CHAR type) |
| BITAND(x,y) | x, y: INT32 or INT64 | INT32 or INT64 | bitwise AND; result is INT64 if x or y is INT64, else INT32 |
| BITASR(x,n) | x: INT32 or INT64, n: INT32 | INT32 or INT64 | arithmetic shift right by n bits, where n >= 0 and n < SIZE(x)*8; result is INT64 if x is INT64, else INT32 |
| BITNOT(x) | x: INT32 or INT64 | INT32 or INT64 | bitwise NOT; result is INT64 if x or y is INT64, else INT32 |
| BITOR(x,y) | x, y: INT32 or INT64 | INT32 or INT64 | bitwise OR; result is INT64 if x or y is INT64, else INT32 |
| BITS(x) | x: INT32 | SET | set corresponding to the integer; the first element corresponds to the least significant digit of the integer and the last element to the most significant digit. |
| BITSHL(x,n) | x: INT32 or INT64, n: INT32 | INT32 or INT64 | logical shift left by n bits, where n >= 0 and n < SIZE(x)*8; result is INT64 if x is INT64, else INT32 |
| BITSHR(x,n) | x: INT32 or INT64, n: INT32 | INT32 or INT64 | logical shift right by n bits, where n >= 0 and n < SIZE(x)*8; result is INT64 if x is INT64, else INT32 |
| BITXOR(x,y) | x, y: INT32 or INT64 | INT32 or INT64 | bitwise XOR; result is INT64 if x or y is INT64, else INT32 |
| CAST(T,x) | T:enumeration type x:ordinal number | enumeration type | the enum item with the ordinal number x; halt if no match |
| | T,x: integer type | T | convert integer types, accept possible loss of information |
| CHR(x) | integer type | CHAR | Latin-1 character with ordinal number x |
| DEFAULT(T) | T = basic type | T | zero for numeric and character types, false for boolean, empty set |
| | T = enumeration type | T | same as MIN(T) |
| | T = pointer/proc type | T | nil |
| | T = record/array type | T | all fields/elements set to their DEFAULT type |
| FLOOR(x) | x: REAL or LONGREAL | INT32 or INT64 | largest integer not greater than x; result is INT64 if x is LONGREAL, else INT32 |
| FLT(x) | x: INT32 or INT64 | REAL or LONGREAL | Convert integer to real type; result is LONGREAL if x was INT64, else REAL, accepting potential loss of information |
| LEN(v, n) | v: array n: INT32 | INT32 | length of v in dimension n (first dimension = 0) |
| LEN(v) | v: array | INT32 | equivalent to LEN(v, 0) |
| | v: string | INT32 | length of string (including the terminating 0X) |
| LONG(x) | x: INT8 or BYTE | INT16 | identity |

| Name | Argument type | Result type | Function |
|---|---|---|---|
| | x: INT16 | INT32 | |
| | x: INT32 | INT64 | |
| | x: REAL | LONGREAL | |
| | x: CHAR | WCHAR | projection |
| MAX(T) | T = basic type | T | maximum value of type T |
| | T = SET | INT32 | maximum element of a set |
| | T = enumeration type | T | last element of the enumeration |
| MAX(x,y) | x,y: numeric type | numeric type | greater of x and y, returns smallest numeric type including both arguments |
| | x,y: character type | character type | greater of x and y, returns smallest character type including both arguments |
| MIN(T) | T = basic type | T | minimum value of type T |
| | T = SET | INT32 | 0 |
| | T = enumeration type | T | first element of the enumeration |
| MIN(x,y) | x,y: numeric type | numeric type | smaller of x and y, returns smallest numeric type including both arguments |
| | x,y: character type | character type | smaller of x and y, returns smallest character type including both arguments |
| ODD(x) | integer type | BOOLEAN | x MOD 2 = 1 |
| ORD(x) | x: CHAR or WCHAR | BYTE or SHORT | ordinal number of x |
| | x: enumeration type | INT32 | ordinal number of the given identifier |
| | x: BOOLEAN | BYTE | TRUE = 1, FALSE = 0 |
| | x: set type | INT32 | number representing the set; the first element corresponds to the least significant digit of the number and the last element to the most significant digit. |
| SHORT(x) | x: INT64 | INT32 | identity |
| | x: INT32 | INT16 | identity |
| | x: INT16 | INT8 | identity |
| | x: LONGREAL | REAL | identity (truncation possible) |
| | x: WCHAR | CHAR | projection (0x if there is no projection) |
| SIGNED(x) | x: unsigned integer T | signed integer T | interprets the raw bytes of x as signed integer of the same byte width |

| Name | Argument type | Result type | Function |
|---|---|---|---|
| SIZE(T) | any type | INT32 | number of bytes required by T |
| STRLEN(s) | s: array of char or wchar | INT32 | dynamic length of the string up to and not including the terminating 0X |
| | s: string literal | | |
| UNSIGNED(x) | x: signed integer T | unsigned integer T | interprets the raw bytes of x as unsigned integer of the same byte width |
| VARARG(T,n) | T: target type, n: argument number | value of type T | Returns the value of the nth variable argument. Only valid in a function declared with variable arguments. |
| VARARG(v,T,n) | v: VA_LIST, T: target type, n: argument number | value of type T | Returns the value of the nth variable argument of the VA_LIST. Can be used in procedures with a VA_LIST formal parameter. |
| VARARGS() | | VA_LIST | Returns an opaque value of the built-in type VA_LIST which can be passed as argument to another procedure. Only valid in a function declared with variable arguments. |

## 6.2 Predeclared proper procedures

6.2.1

| Name | Argument types | Function |
|---|---|---|
| ASSERT(x) | x: Boolean expression | terminate program execution if not x |
| ASSERT(x, n) | x: Boolean expression | terminate program execution if not x |
| | n: integer constant | |
| DEC(v) | integer type | v := v - 1 |
| | enumeration type | previous ident in enumeration |
| DEC(v, n) | v, n: integer type | v := v - n |
| DISPOSE(p) | p: pointer | free the memory allocated before using NEW() |
| EXCL(v, x) | v: SET; x: integer type | v := v - {x} |
| HALT(n) | integer constant | terminate program execution |
| INC(v) | integer type | v := v + 1 |
| | enumeration type | next ident in enumeration |
| INC(v, n) | v, n: integer type | v := v + n |
| INCL(v, x) | v: SET; x: integer type | v := v + {x} |
| NEW(p) | p: pointer to T | allocate p^ of type T |
| NEW(p,n) | p: pointer to open array | allocate p^ with length n |
| | x: integer type | |
| PCALL(e,p,$a_0$, …,$a_n$) | VAR e: SYMBOL; p: proper procedure type; $a_i$: actual parameters | call procedure type p with arguments $a_0$…$a_n$ corresponding to the parameter list of p; e becomes nil in normal case and gets the SYMBOL to RAISE() otherwise |

| Name | Argument types | Function |
|---|---|---|
| RAISE(e) | e: SYMBOL | terminates the last protected function called and returns e as the exception value; RAISE() never returns |

## 7 References


7.1 **[Mo91]** Mössenböck, H.; Wirth, N. (1991). The Programming Language Oberon-2. Structured Programming, 12(4):179-195, 1991. http://www.ssw.uni-linz.ac.at/Research/Papers/Oberon2.pdf (accessed 2020-11-16).

7.2 **[Oak95]** Kirk, B. et al. (1995). The Oakwood Guidelines for Oberon-2 Compiler Developers. Revision 1A. https://web.archive.org/web/20171226172235/https://www.math.bas.bg/bantchev/place/oberon/oakwood-guidelines.pdf (accessed 2022-04-26).

7.3 **[OBX]** Keller, R. (2021): The Programming Language Oberon+, github.com/.../The_Programming_Language_Oberon+.adoc (accessed 2024-02-03).