

DRAFT

MiGOberon™ Language Report

Michael Griebling

April 2010

Table of Contents

MiGOberon Language Report.....	5
1 Introduction.....	5
1.1 Acknowledgements.....	5
2 Program Composition.....	5
3 Syntax Notation.....	5
3.1 Definition of Extended Backus-Naur Formalism.....	5
3.2 EBNF defined in EBNF.....	6
3.3 Description of EBNF.....	6
3.3.1 Sequence.....	6
3.3.2 Repetition.....	6
3.3.3 Selection.....	6
3.3.4 Option.....	6
3.3.5 Quotes and bold font	6
4 Language Symbols and Identifiers.....	7
4.1 Vocabulary and Representation.....	7
4.2 Identifiers.....	7
4.3 Modifiers and Specifiers.....	7
4.4 Numeric Constants.....	7
4.5 Character Constants.....	8
4.6 String Constants.....	8
4.7 Reserved Words, Delimiters and Operators.....	8
4.7.1 Reserved Words.....	8
4.7.2 Delimiters.....	9
4.7.3 Predefined Operators.....	9
4.7.4 User-Defined Operators.....	9
4.8 Comments.....	9
5 Declarations.....	9
5.1 Identifier Declarations and Scope Rules.....	9
5.1.1 Declaration Modifiers.....	10
5.2 Constant Declarations.....	10
5.3 Type Declarations.....	10
5.3.1 Basic Types.....	10
5.3.2 Type Widths.....	11
5.3.3 Enumeration Types.....	11
5.3.5 Array Types.....	11
5.3.6 The string Type.....	12
5.3.7 Object Types.....	12
5.3.8 Record Types.....	13
5.3.10 Procedure Types.....	13
5.3.12 Conversion between Types.....	14
5.3.12.1 Type name used as conversion function (predefined types).....	14

5.3.12.2 Type name used as conversion function (object types).....	15
5.3.12.3 Implicit type of constant.....	15
5.4 Variable declarations.....	15
6 Expressions.....	16
6.1 Operands and Designators.....	16
6.2 Predefined Operators.....	16
6.2.1 Logical operators.....	17
6.2.2 Arithmetic operators.....	17
6.2.3 Set Operators.....	17
6.2.4 Relations.....	18
6.3 User-Defined Operators and Operator Declarations.....	18
6.3.1 Operators overloading.....	18
6.3.2 New Operator Declarations.....	18
6.3.3 Rules governing overloading.....	19
6.4 Operator Precedence.....	20
6.5 Numeric resolution within expressions.....	21
7 Statements.....	21
7.1 The Assignment Statement.....	21
7.1.1 Indexer Assignments.....	22
7.1.2 Abstract Assignments.....	22
7.2 The Procedure Call.....	23
7.3 The if Statement.....	23
7.4 The case Statement.....	23
7.5 The while Statement.....	24
7.6 The repeat Statement	24
7.7 The for Statement.....	25
7.8 The loop Statement	25
7.9 The return Statement.....	25
7.10 The Block Statement.....	26
7.10.1 Exception Handling.....	26
8 Procedure and Method Declarations and Formal Parameters.....	27
8.1 Procedure Modifiers.....	27
8.2 Method-level Protection.....	28
10 Mathematical extensions.....	28
10.1 Data structures.....	28
10.1.1 Expression arrays.....	28
10.2 Indices.....	29
10.2.1 Simple indices.....	29
11 Modules.....	29
11.1 The Module.....	29
11.2.1 Inheritance and Multiple Inheritance.....	30
11.2.2 Polymorphism.....	30
13 Definition of Terminology.....	30
13.1 Numeric Types.....	30
13.2 Same Types.....	31
13.3 EqualTypes.....	31
13.4 Assignment Compatible.....	31

13.5 ArrayCompatible.....	31
13.6 Compatible for Expressions and Operator Overloading.....	31
13.7 Matching Formal Parameter Lists.....	32
14 Predefined Procedures.....	32
15 Input and Output Procedures.....	33
15.1 Parameters and special syntax.....	33
15.2 Input Procedures.....	33
15.2.1 The read procedure.....	33
15.2.2 The readln procedure.....	33
15.3 Output Procedures.....	33
15.3.1 The write procedure.....	33
15.3.2 The writeln procedure.....	34
15.3.3 Default values of widths in write and writeln.....	34
16 Syntax.....	34

MiGOberon Language Report

1 Introduction

MiGOberon is a new programming language in the Pascal, Modula-2, Oberon, and Zonnon family. It retains an emphasis on simplicity, clear syntax and separation of concerns. Although more compact than languages such as C#, Java and Ada, it is a general-purpose language suited to a wide range of applications. Typically this includes component-oriented composition, algorithms and data structures, object-oriented and structured programming, graphics, mathematical programming and low-level systems programming. MiGOberon provides a rich object model with encapsulated behavior. It may be used to write programs in procedural and object-oriented styles. MiGOberon is also well suited for teaching purposes, from basic principles right through to advanced concepts.

The object model in MiGOberon is based on the notion that ‘everything is an object’. Many of the concepts in MiGOberon have been drawn from its heritage. The intention has been to offer expressive and cohesive features which have proved their worth. MiGOberon also introduces some new features such as operator overloading for representing mathematical and other expressions in a natural way and exception handling for improving reliability. Some features such as built-in input/output features have been reintroduced from earlier members of the Pascal language family and enumeration types from Modula-2. When choosing a language for building modern systems achieving interoperability between programs written in different languages within the same system is an important consideration. The MiGOberon language is specifically designed to be platform-independent whilst supporting interoperability.

1.1 Acknowledgements

Thanks to Jörg Gutknecht, Brian Kirk, and David Lightfoot for their excellent Zonnon Language Report from which I have borrowed heavily in developing the MiGOberon language report. Any mistakes introduced are mine and apologies for removing some of what they must have spent much time developing.

2 Program Composition

MiGOberon programs are composed from modules object, definition and implementation. More precisely:

A *module* has a dual nature: it declares a syntactic container for logically cohesive program declarations and it simultaneously declares an object whose lifecycle is controlled by the system. So the module provides the mechanism for the textual partitioning of a source program and also the dynamic loading at execution time of a part of a program, in the form of an instantiated object.

Any number of dynamically created objects may have their life cycles managed by a program, however only a single instance of each module’s object may be instantiated by the system at any given time. Because the module forms a unit of encapsulation and data hiding, it is also ideal as a container for implementing abstract data types.

3 Syntax Notation

The syntax of the language is defined in an *Extended Backus-Naur Formalism* (EBNF). The complete syntax of MiGOberon is defined in section 17. Relevant fragments of the syntax are also provided in the text as each feature of the language is defined.

3.1 Definition of Extended Backus-Naur Formalism

The EBNF notation used in this report has the following features:

- Alternatives are separated by |.
- Brackets [and] denote that the enclosed expression is optional.
- Braces { and } denote its repetition (possibly 0 times).
- Parentheses (and) are used to form groups of items.
- Non-terminal symbols start with an upper-case letter (e.g. *Statement*).
- Terminal symbols either start with a lower-case letter (e.g. *letter*), or are written in bold letters (e.g. **begin**), or are denoted by strings (e.g. ":=").
- Comments start with // and continue to the end of the line.

3.2 EBNF defined in EBNF

It is possible to define the EBNF syntax using EBNF as an example

```
Syntax= {Production}.
Production= NonTerminalSymbol "=" Expression ".".
Expression= Term {"|" Term}.
Term = Factor {Factor}.
Factor = terminalSymbol | NonTerminalSymbol | "(" Expression ")" | "[" Expression "]" | "{" Expression "}" .
```

3.3 Description of EBNF

The EBNF constructs are described below:

3.3.1 Sequence

$A = BC.$

An A consists of a B followed by a C

Examples:

```
Sentence = Subject Predicate.
FileName = Name '.' Extension.
Name = FirstName Surname.
```

3.3.2 Repetition

$A = \{B\}.$

An A consists of zero or more B 's.

Examples:

```
File = {Record}.
Bill = {Item Price}.
```

3.3.3 Selection

$A = B \mid C.$

An A consists of a B or a C .

Examples:

```
Fork = Resource \mid Data.
Meal = Breakfast \mid Lunch \mid Dinner.
```

3.3.4 Option

$A = [B].$

An A consists of a B or nothing.

Example:

```
SelectedDrink = [ Tea \mid Coffee \mid Chocolate ]. // Possibly none!
```

3.3.5 Quotes and bold font

Text in quotes or in a **bold** font stands for itself.

Examples:

```
ImportDeclaration = import Import {"," Import}.
OwnSymbol = "me" \mid self.
```

4 Language Symbols and Identifiers

4.1 Vocabulary and Representation

Symbols are identifiers, numbers, strings, operators, and delimiters. There are some lexical rules:

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

4.2 Identifiers

Identifiers are sequences of letters and digits and underscores ‘_’. The first character must be a letter or an underscore.

```
ident = ( letter | "_" ) { letter | digit | "_" }.  
letter = "A" | ... | "Z" | "a" | ... | "z" | // any other "culturally-defined" letter
```

The case of letters is significant in identifiers, except in predefined identifiers which may be written *either* entirely in lower-case letters *or* entirely in upper-case letters (see 5.3.1 and section 17)

Examples:

```
X      Scan  MiGOberon GetSymbol firstLetter  
_external_package27 (* underscore typically used for interoperability with other languages*)
```

4.3 Modifiers and Specifiers

A *modifier* is used to indicate alternative semantics, where the same syntax is used for more than one purpose. It is a list of identifiers contained in braces and separated by commas. The valid identifiers depend on the context in which the modifier is applied.

```
Modifiers = "{" IdentList "}".  
IdentList = ident { "," ident }.  
ident = ( letter | "_" ) { letter | digit | "_" }.  
letter = "A" | ... | "Z" | "a" | ... | "z" | // any other "culturally-defined" letter
```

Examples:

```
{ value } { public }      { public, value }
```

A *specifier* is used to provide additional information such as the type of an expected object, or a width. It comprises a list of words or numbers contained in braces { } or an EBNF protocol specification (See also 5.3.4)

Examples:

```
var r: real{32}; (* real in 32-bit format *)  
i := integer(t); (* the value of t expressed as an integer *)
```

4.4 Numeric Constants

Numbers are (unsigned) integer, cardinal or real constants. If the constant is specified with the suffix *H*, the representation is hexadecimal, otherwise the representation is decimal. A real number always contains a decimal point and optionally it may also contain a decimal scale factor. The letter *E* means ‘times ten to the power of’. A numeric constant may optionally be followed by a width modifier which is the number of bits to be used for its representation (surrounded by braces). If no width is specified then the default value defined in the *MiGOberon Programmers’ Manual* is used [Compiler]. For further information on types see 12.1.

```
number = (whole | real) [ "{" Width "}"].  
whole = digit {digit} | digit {hexDigit} "H".  
real = digit {digit} "." {digit} [ScaleFactor].  
ScaleFactor = "E" ["+" | "-"] digit {digit}.  
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".  
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Width = ConstExpression.

A *whole* constant is compatible with both integer (signed) types and cardinal (unsigned) types.

Examples:

<i>constant</i>	<i>type</i>	<i>value</i>
1991	integer or cardinal	1991
0DH{8}	integer{8} or cardinal{8}	13
12.3	real	12.3
4.567E8	real	456700000
0.57712566E-6{64}	real{64}	0.00000057712566

4.5 Character Constants

A character constant is a character enclosed in single (') or double (") quote marks. The opening quote must be the same as the closing quote and must not be the character itself. Character constants may also be denoted by the ordinal number of the character in hexadecimal notation followed by the letter X.

```
CharConstant = ''' character ''' | ''' character ''' | digit { HexDigit } "X".
character = // Any character from the alphabet except the current delimiter character
```

This is useful for expressing special characters that are either non-printable or that are part of an extended character set.

Examples:

```
"a" 'n' ''' ''' 20X
```

4.6 String Constants

String constants are sequences of characters enclosed in single (') or double (") quote marks; the opening quote must be the same as the closing quote. A string constant may not contain its delimiting quote character or a line break. The number of characters in a string is called its length. A single character string constant (of length 1) can be used wherever a character constant is allowed and vice versa. String constants can be assigned to variables of type *string* (see 5.3.1).

```
StringConstant = ''' { character } ''' | ''' { character } '''.
character = // Any character from the alphabet except the current delimiter character
```

Examples:

```
"MiGOberon" "Don't worry!" "x" 'hello world'
```

4.7 Reserved Words, Delimiters and Operators

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

4.7.1 Reserved Words

The following reserved words (shown in **bold** in this report) may not be used as identifiers and are written *either* entirely in lower-case letters:

```
array begin by case const div do else elsif end exception exit false for if import in is loop mod module new nil object of operator or  
procedure record read readln repeat return termination then to true type until var while write writeln
```

or entirely in upper-case letters:

```
ARRAY BEGIN BY CASE CONST DIV DO ELSE ELSIF END EXCEPTION EXIT FALSE FOR IF IMPORT IN IS LOOP MOD MODULE  
NEW NIL OBJECT OF OPERATOR OR PROCEDURE RECORD READ READLN REPEAT RETURN TERMINATION THEN TO TRUE  
TYPE UNTIL VAR WHILE WRITE WRITELN
```

4.7.2 Delimiters

The delimiter characters are:

```
( ) [ ] { } . (dot) , (comma) ; (semicolon) : (colon) .. (range)
| (case separator) ' (single quote) " (double quote)
```

4.7.3 Predefined Operators

The predefined operators are:

```
- (unary minus) + (unary plus) ~ (negation)
** (exponentiation)
+ - * / div mod & or
:= (assignment) = (equality) # (not equal) < <= > >= in
```

4.7.4 User-Defined Operators

MiGOberon introduces the concept of user-defined operators. They are declared like procedures. (See 6.3).

4.8 Comments

Comments may be inserted between any two symbols in a program. They are arbitrary character sequences opened by the bracket (*** and closed by ***). Comments may be nested. They do not affect the meaning of a program. They are shown in italics in this report.

```
(* this is a comment *)
(* This comment continues
   on to the next line *)
(* Nested comments (* look like *) this *)
```

5 Declarations

Declarations are used to introduce identifiers and to indicate their type.

```
Declarations = { SimpleDeclaration } { ProcedureDeclaration }.
SimpleDeclaration = ( const [ Modifiers ] { ConstantDeclaration ";" }
                     | type [ Modifiers ] { TypeDeclaration ";" }
                     | var [ Modifiers ] { VariableDeclaration ";" } ).
```

5.1 Identifier Declarations and Scope Rules

Every identifier occurring in a program must be introduced by a declaration, unless it is predefined. Declarations also specify certain permanent properties of an item, such as whether it is a constant, a type, a variable (see 5.4), or a procedure (see section 1). The identifier is then used to refer to the associated item.

The scope of an identifier is the scope to which its declaration belongs and hence to which it is local. It excludes the scopes of identically named identifiers which are declared in nested blocks. The scope rules are:

- No identifier may denote more than one item within a given scope (i.e. no identifier may be declared more than once in a block).
- An identifier may only be referenced within its scope.
- Identifiers denoting object fields or methods/procedures are valid only in object designators, where they must be qualified by the name of the object.
- Related declarations within a scope may be declared in any order.

```
IdentList = ident { "," ident }.
```

Examples:

```
Month.Oct (* see 5.3.3 *)
NameSpace.Program
```

5.1.1 Declaration Modifiers

Declarations may have optional modifiers which are defined as follows:

- *private*: the identifiers are visible only in the scope of their declarations.
- *public*: the identifiers are visible in the scope in which they are declared and in any constructs that explicitly import the program construct that contains its declaration.
- *immutable*: is used in conjunction with *public* and indicates that values are read-only from outside the scope in which of declaration.

Declaration modifiers may be used with declarations in the inner scope of a procedure/method.

```
SimpleDeclaration = ( const [ Modifiers ] { ConstantDeclaration ";" }  
                    | type [ Modifiers ] { TypeDeclaration ";" }  
                    | var [ Modifiers ] { VariableDeclaration ";" } ).
```

Examples:

```
var {private} flag, statusWord: boolean; (* flag and statusWord are both private *)  
var {public, immutable} refCount: integer; (* read only access *)
```

5.2 Constant Declarations

A constant declaration associates an identifier with a constant value.

```
ConstantDeclaration = ident "=" ConstExpression. ConstExpression = Expression.
```

Examples:

```
const N = 10;  
      limit=2*N-1; (*see 6.2.2*)  
      fullSet = { min(set) .. max(set) }; (* see 5.3.1 *)
```

A constant expression is an expression that can be evaluated solely by a textual scan without actually executing the program. Its operands must be constants or calls of predefined functions.

5.3 Type Declarations

A data type determines the set of values variables of that type may assume and the operators that are applicable to them. A type declaration associates an identifier with a type. In the case of the structured types (arrays and objects) it also defines the structure of variables of this type. Object types are defined in 5.3.6 and 10.1

```
TypeDeclaration = ident "=" Type.
```

Example:

```
Type = (TypeName [ "{" Width "}" ] | EnumType | ArrayType | ProcedureType | ObjectType | RecordType ).
```

5.3.1 Basic Types

The basic types are denoted by predefined identifiers. The associated operators are defined in 6.2 and the predefined function procedures in section 13. The values of the basic types are the following:

- **boolean** the truth values *true* and *false*
- **char** the underlying character set of the environment
- **integer** the integers between *min(integer)* and *max(integer)*
- **cardinal** positive whole numbers between *min(cardinal)* and *max(cardinal)*
- **fixed** large numbers with fixed precision between *min(fixed)* and *max(fixed)*
- **real** the real numbers between *min(real)* and *max(real)*

- `set` the set of whole numbers (integer or cardinal) between 0 and *max(set)*
- `string` character strings type

5.3.2 Type Widths

For types *char*, *integer*, *cardinal*, *real* and *set* the number of bits required to contain the value can be specified by a modifier stating a whole number of bits as a constant value in braces { } after the type name. The default type widths are:

```
char{16}, set{32}, integer{32}, real {32}, cardinal {32}, fixed{128}
```

For conversion between different types see section 5.3.12.

5.3.3 Enumeration Types

An enumeration is a type that comprises a named list of identifiers denoting the values which constitute the type. These identifiers are qualified by the type name when used as named constants in the program. The values are ordered and their ordering relation is defined by their textual sequence in the enumeration list. No other values belong to the type. The ordinal number of the first value is zero and increases by one for each subsequent identifier.

```
EnumType = "(" IdentList ")". IdentList = ident { "," ident }.
```

Examples:

```
type NumberKind = (Bin, Oct, Dec, Hex);
Month = (Jan, Feb, Mar, Apr, May, Jun, July, Sep, Oct, Nov, Dec);
```

Names in separate enumerations need not be different as their use is always qualified. So for example *NumberKind.Oct* is distinct from *Month.Oct*.

Values of expressions can be converted to a different type. (See section 5.3.12).

The predefined function *pred* returns the value of the predecessor of the enumeration value given as its parameter, for all except the first value of the enumeration. The predefined function *succ* returns the value of the successor of the enumeration value given as its parameter, for all except the last value of the enumeration.

5.3.5 Array Types

An array is a structure consisting of a number of elements that are all of the same type, called the element type. Arrays can be indexed either by a positive whole number or by a value of an enumeration type. In the first case, the number of elements in the array's declaration determines its length. The array's elements are designated by indices, which are whole-number values between 0 and the array length minus 1. In the second case the name of the enumeration type is used in the declaration and the array's elements are designated by values of the enumeration type.

The syntax rules for the array type are:

```
ArrayType = array Length {"," Length} of Type.
Length = ConstExpression | "*".
```

Arrays can be multidimensional; that is, the array elements may themselves be arrays, and mixing the different length specification forms is acceptable in principle. An example and a counter example are:

```
type Acceptable=array * of array 42 of T;(*array*,42ofT*)
Jagged = array 42 of array * of T; (* 'jagged' array *)
```

The declaration *array m, n of T* is textually equivalent to *array m of array n of T*.

For example

```
array * of array 42 of T
```

can be written

```
array *, 42 of T
```

The expression *len(a, n)* returns the number of elements in dimension *n* of the array *a*. The expression *len(a)* is a shorthand for *len(a, 0)*.

In an array the number of elements in any dimension may be variable and is then denoted by an asterisk. It is the programmer's responsibility to allocate storage space on the heap for an array by using the reserved word *new* for each instance of the array:

```
arrayVariable := new ArrayType(length0, length1, ... );
```

The length values must be expressed by positive expressions of integer or cardinal type and the number of such expressions must correspond to the number of dimensions of the variable. An array must be declared as either fully static or fully dynamic.

Examples of the use of arrays are:

```
type Vector = array * of integer;

procedure CreateAndReadVector(var a: Vector); var i, n: integer;
begin
  read(n); a := new Vector(n);
  for i := 0 to len(a) - 1 do
    read(a[i])
  end
end CreateAndReadVector;

procedure InitializeMatrix(var mat: array *, * of real); var i, j: integer;
begin
  for i := 0 to len(mat, 0) - 1 do
    for j := 0 to len(mat, 1) - 1 do
      mat[i, j] := 0.0
    end
  end
end InitializeMatrix;

...

var m: array 10, 10 of real;

...

InitializeMatrix(m);
```

For information about math arrays, see 10.

5.3.6 The string Type

Variables of type *string* represent immutable sequences of characters. Strings can be compared for equality and inequality by using the '=' and '#' operators. The operator '+' signifies concatenation of strings and ':=' signifies assignment. The predefined procedure *copy* converts between *string* type and *array of char* representation and vice versa and the predefined function *len* delivers the length of a string (see Section 13). The properties of the string type are not defined as part of the MiGOberon language. An example of a library module *Strings* is shown in Section 15. String syntax and constants are defined in section 4.6.

5.3.7 Object Types

An object is a data type template comprising fields and methods. The fields represent the object's state and the methods its functionality and the activities its concurrent activities. It exposes its interface to its system environment by the interface of its intrinsic type (referred to as its intrinsic interface), that is the set of all the elements which the programmer chooses to make public rather than keep private.

```
ObjectType = object ObjectDefinition ident.
ObjectDefinition = [ FormalParameters ] ";,"
  [ ImportDeclaration ]
  { SimpleDeclaration | ProcedureDeclaration | ( UnitBody | end ) }.
Object = object [ Modifiers ] ObjectName ObjectDefinition SimpleName. // when declared as a unit

ImportDeclaration = import Import { ";," Import } ";,".
```

```

Import = ImportedName [ as ident ].
ImportedName = ( ModuleName | ObjectName ).
UnitBody = begin [ StatementSequence ] end.

```

An object is composed of declarations including constants, types, variables (referred to as fields), and procedures (referred to as methods). Variables which are reference objects provide references to objects which are created dynamically during program execution within the program using *new*. An object may optionally have parameters which are used in the body of the object to initialize fields when the object is instantiated using *new*.

The modifiers *public* and *private* can be used to declare the visibility of the contents of an object. If no modifier is present then the default is *private*. Individual items may be made public by explicit use of the modifier *public* following their declaration. The object itself can also have a modifier which denotes it as either a value object or a reference object using the modifier values *value* and *ref* respectively. The default modifier is *value*.

Examples:

```

object {ref} Box(w, h: integer);
  var width, height: integer;

  procedure Area( ): integer;
  begin
    return width * height
  end Area;

begin
  self.width := w; self.height := h (* self is optional in both cases here *)
end Box.

...
var box: Box;
...
box := new Box(3, 7); (* makes new Box object with width 3 and height 7 *)

```

See 10.2 on *objects* as separately compiled program units; object declarations cannot be nested.

5.3.8 Record Types

A *record* is a value object type. It can be used to encapsulate variable declarations. The keyword *record*. Record declarations cannot be nested.

```

RecordType = record ident { VariableDeclaration ";" } end ident.

```

Examples:

```

record Position (* declares the record-type Position *)
  x, y: integer
end Position;

record Date; (* declares the record-type Date *)
  year: integer{8};
  month: Month;
  day: integer{8}
end Date;

```

5.3.10 Procedure Types

A procedure type is a template for a procedure which can be referenced via a procedure variable. A variable of a procedure type *T* has a procedure or method *P* or *nil* as its value. If *P* is assigned to a variable of type *T*, the formal parameter lists of *P* and *T* must match according to a set of rules. (See 12.4). *P* must not be a predefined procedure nor may it be local to another procedure. However, the sole exception is a global module procedure, which may be used with or without qualification within the module in which it is declared.

When a method is assigned to a variable of type procedure it must be prefixed by (the designator of) an object instance that contains it. This may also be referred to as a 'delegate'. When any procedure is called it shares the thread of execution with its caller, on termination of the procedure the caller resumes execution from the statement immediately following the call.

```

ProcedureType = procedure [ ProcedureTypeFormals ].
ProcedureTypeFormals = "(" [ PTFSection { ";" PTFSection } ] ")" [ ":" FormalType ].
PTFSection = [ var ] FormalType { "," FormalType }.

```

FormalType = { **array** "*" **of** } (TypeName | InterfaceType).

Example:

```
type Delegate=procedure;
  Action = procedure (n: integer);
  Function = procedure (n: integer): integer;
```

5.3.12 Conversion between Types

In MiGOberon, type conversions within a ‘family’ (such as *integer*) are implicit when guaranteed to be safe. However, conversions between families must be explicit (because a change of internal representation is involved). Inverse conversions (for example, *integer*{32} to *integer*{16}) must always be explicit. The exception mechanism detects conversion anomalies (see 7.10.1).

The interoperability between types is summarized in the table below:

Type family	width in bits							
	8		16		32		64	128
fixed								M
real					M	→	M	↗
integer	M	→	M	→	M	→	M	
cardinal	M	→	M	→	M	→	M	
char	M	→	M					

M mandatory type for conforming implementation
 → implicit conversion always allowed (within same family)
 ↗, ↑ explicit conversion always allowed (change of representation)

Note that implicit conversions are transitive. Inverse conversion (in the opposite direction of the arrows) requires an explicit conversion and may result in truncation or an exception.

5.3.12.1 Type name used as conversion function (predefined types)

To achieve a type conversion, the name of the destination type is regarded as a built-in function which takes an expression of the source type as a parameter and returns the converted value. An optional second parameter indicates the desired width of the result.

Syntax:

TypeName "(" Expression ["," Size] ")"

Examples:

```
integer(x + e/f, 16)
```

is the value of the expression $x + e/f$ represented as a 16-bit integer (exception may be raised if conversion not possible).

```
integer(x + e/f)
```

is the value of the expression $x + e/f$ represented as a 32-bit integer (assuming that 32 is the implementation’s default width for integer).

Note that integers cannot be implicitly conversion to real and so:

```
var count, sum: integer; mean: real;
...
mean := sum / count
```

is *not* syntactically allowed and requires explicit conversions:

```
mean := real(sum) / real(count)
```

5.3.12.2 Type name used as conversion function (object types)

MiGOberon supports both object-oriented programming and operator-style programming:

In object-oriented programming, the desired definition (interface) of a servant object needs to be known but not its full type. If an object type X implements definitions D and E , instances of X can be regarded as being of types D or E respectively, depending on the client's perspective. So, if D exports a method f and E exports a method g and x is a variable of type X , we can write $D(x).f$ and $E(x).g$, for example. In operator-style programming, we apply operators to operands of a certain statically known type (strong typing). For example, we might want to apply the operator procedure:

```
operator "*" (x: X; y: Y): Z; ...
```

to generic objects:

```
var s, t: object;
```

We need to use a type cast which takes the function-like *type-name(expression)* in this case:

```
if (s is X) & (t is Y) then z := X(s) * Y(t) else (* type error *) end
```

5.3.12.3 Implicit type of constant

The type of a simple numeric constant is determined by the declaration of the variable to which it is assigned. So for instance, given the declaration:

```
var i: integer {16};
```

then the assignment

```
i := 1
```

is actually treated by the compiler as being

```
i := 1{16}
```

If no width is specified, then the default width for that type is assumed (see 5.3.2)

Other type conversions are achieved by means of predefined procedures (see Section 13).

5.4 Variable declarations

A variable holds a value that can be assigned to it from an expression in an assignment operation (see 7.1). A variable is defined to have a type, which may not change, and which defines the set of values that it may hold. Variable declarations introduce variables by defining an identifier and a data type for each one.

```
VariableDeclaration = IdentList ":" Type.
```

Examples:

```
var i, j, k: integer;  
    x, y: real;  
    p, q: boolean;  
s: set {32};  
a: array 100 of real;  
name: array 32 of char;  
size, count: integer;  
mousePosition: Position;  
dateOfBirth, today: Date;
```

6 Expressions

An expression is a construct which specifies a computation. In an expression constants and current values of variables are combined to compute other values by the application of operators and function procedures. An expression consists of

operands and operators; parentheses may be used to express specific associations of operators and operands. The types of intermediate values used during expression evaluation are the responsibility of the implementation (see [Compiler]). The type of the result of an expression is defined in the section on expression compatibility (see 12.6).

```

Expression = SimpleExpression [ ( "=" | "#" | "<" | "<=" | ">" | ">=" | in ) SimpleExpression ].
SimpleExpression = [ "+" | "-" ] Term { ( "+" | "-" | or ) Term }.
Term = Factor { ( "*" | "/" | div | mod | "&" ) Factor }.
Factor = number
        | CharConstant
        | string
        | nil
        | Set
        | Designator
        | new TypeName [ "(" ActualParameters ")" ]
        | "(" Expression ")"
        | "~" Factor
        | "!" Factor
        | Factor "~~~" Factor.

```

6.1 Operands and Designators

With the exception of set constructors (see 6.2.3) and literal constants (numbers, character constants, or strings constants), 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 an identifier denoting a module, definition, implementation or object and may be followed by selectors if object is an element of a structure.

```

ExpressionRange = Expression | Range.
Range = [ Expressions ] ".." [ Expression ] [ "by" Expression ] .
Designator = Instance
            | TypeName "(" Expression [ "," Size ] ")"           // Conversion
            | Designator "[" ExpressionRange { "," ExpressionRange } "]" // Array elements
            | Designator "(" [ ActualParameters ] ")"           // Function call
            | Designator "." MemberName                         // Member selector
Instance = ( self | InstanceName | DefinitionName "(" InstanceName ")" ).
Size = ConstantExpression.
ActualParameters = Actual { "," Actual }.
Actual = Expression [ "{" [ var ] FormalType "}" ] . // Argument with type signature

```

Examples:

<i>designator</i>	<i>type</i>	<i>meaning</i>
size	integer	value of the variable called <i>size</i>
a[i]	real	the element of the array <i>a</i> at position <i>i</i>
dateOfBirth.day	integer{8}	the <i>day</i> field of the object called <i>dateOfBirth</i>
w[3].name[i]	char	the element at position <i>i</i> in the field called <i>name</i> of the element at position 3 of the array called <i>w</i>

If *a* designates an array, then *a[e]* denotes that element of *a* whose index is the current value of the expression *e*. The expression *e* must be of an enumeration, a cardinal or an integer type. A designator of the form *a[e₀, e₁, ..., e_n]* stands for *a[e₀][e₁]...[e_n]*. For further information about array indices see 10.2.

If *obj* designates an object, then *obj.f* denotes the field *f* of *obj* or the method *f* of the object *obj*, (see 11.1).

If the designated object is a constant or a variable, then the designator refers to its current value. If it is a procedure without any parameter list, the designator refers to the procedure itself. However, if it is a function procedure and is followed by a (possibly empty) parameter list it causes an activation of that procedure and stands for its resulting value. The actual parameters must correspond to the formal parameters as in proper (non-function) procedure calls. (See 7.2).

6.2 Predefined Operators

Predefined operators are fixed and built into the language. For further information about operators on math arrays see 10.3.

6.2.1 Logical operators

These operators apply to *boolean* operands and yield a *boolean* result.

or	logical disjunction	$p \text{ or } q$	'if p then <i>true</i> , else q '
&	logical conjunction	$p \text{ \& } q$	'if p then q , else <i>false</i> '
~	negation	$\sim p$	'not p '

6.2.2 Arithmetic operators

The operators $+$, $-$, and $*$ apply to operands of numeric types in an expression. (See 6.3.1). The division operator $/$ applies only to operands of type *real* and produces a result of type *real*. When used as monadic operators, $-$ denotes sign inversion and $+$ denotes the identity operation.

+	sum
-	difference
*	product
/	real quotient (of reals)
**	power ($x**y$ signifies x_y)

Examples:

```
i := j + k; x := real(i) / real(j); (* see section 5.3.12 *)
```

The operators *div* and *mod* apply to integer and cardinal operands only.

div	integer quotient
mod	modulus

They are related by the following formulas defined for any x and positive divisors y :

$$x = (x \text{ div } y) * y + (x \text{ mod } y)$$

$$0 \leq (x \text{ mod } y) < y$$

If the value of the divisor y is negative then the meanings of the operators *div* and *mod* are mathematically ambiguous and so are left undefined; their effect is implementation specific. It is recommended that programmers test for this condition and employ mathematics to ensure that only positive divisors values are used. For example:

x	y	$x \text{ div } y$	$x \text{ mod } y$
5	3	1	2
-5	3	-2	1

6.2.3 Set Operators

Set operators apply to operands of type *set* and yield a result of type *set*. The declared bit widths of the operand *SETs* must be identical. The monadic minus sign denotes the complement of x , that is, $-x$ denotes the set of integers between 0 and $\text{max}(\text{set})$ which are *not* elements of x .

+	union	bitwise or
-	difference ($x - y = x * (-y)$)	bitwise subtraction
*	intersection	bitwise and
/	symmetric set difference ($x / y = (x - y) + (y - x)$)	bitwise exclusive or

A set constructor defines the value of a set by listing its elements, if any, between braces. The elements must be integers in the range $0 \dots \text{max}(\text{set})$. A range $m \dots n$ denotes all integers in the interval starting with element m and ending with element n , inclusive of m and n . If $m > n$ then $m \dots n$ denotes an empty set.

```
Set = "{" [ SetElement { "," SetElement } ] "}".
SetElement = Expression [ ".." Expression ].
```

Examples of the use of sets:

```
const left = 0; right = 1; top = 2; bottom = 3;
var edges: set; x, y: integer;

begin
```

```

edges := { }; (* the empty set *)
if x < xMin then edges := edges + {left}
...
if left in edges then...(*clipatleft*) ...

const opCodemask = {0..3};
var opCode, word: set;
...
opCode := word * opCodeMask; (* extract the op-code *)

```

6.2.4 Relations

Relations yield a *boolean* result. The relations =, #, <, <=, >, and >= apply to the numeric types and *char*. The relations = and # also apply to *boolean*, *set* and *string*, as well as to procedure types (including the value *nil*). *x in s* stands for 'x is an element of s'. *x* must be of an integer type, and *s* of type *set*.

=	equal
#	unequal
<	less
<=	less or equal
>	greater
>=	greater or equal
in	set membership
is	<i>x is T</i> is true if and only if the intrinsic type of <i>x</i> is <i>T</i> , for any type <i>T</i>

Examples of expressions

<i>expression</i>	<i>type</i>	<i>meaning</i>
1991	integer	simple constant value
<i>i</i> div 3	integer	integer division of <i>i</i> by 3
~wellFormed or outOfRange	boolean	(not well-formed) or out-of-range
(<i>i</i> + <i>j</i>) * (<i>i</i> - <i>j</i>)	integer	arithmetic expression
<i>s</i> - {8, 9, 13}	set{8}	<i>s</i> with 8, 9, 13 removed
keys in {left, right}	boolean	<i>keys</i> is <i>left</i> or <i>right</i> or <i>both</i>
('0' <= <i>ch</i>) & (<i>ch</i> <= '9')	boolean	<i>ch</i> is a digit

6.3 User-Defined Operators and Operator Declarations

Operator overloading introduces the notion of user-defined operators and the opportunity to use familiar syntax in expressions involving them. Operators are defined only in a module implementing an abstract data type i.e. one that defines a new user-defined type and that implements a set of operations on it. Typically this can be used when introducing new data types such as complex numbers or matrices.

6.3.1 Operators overloading

The set of predefined operators that can be overloaded is as follows:

```

- (unary minus)
+ (unary plus)
~ ** (exponentiation)
+ - * / div mod & or = # < <= > >= in := (assignment is a special case, see 6.3.3)

```

The precedence of operators is defined in 6.4. Note that the *is* operator cannot be overloaded, see 10.1.

6.3.2 New Operator Declarations

Overloaded and new operators are introduced as operator declarations. The syntax of the declaration is as follows:

```

OperatorDeclaration = operator [ Modifiers ] OpSymbol [ FormalParameters ] ";" OperatorBody ";".
OperatorBody = Declarations UnitBody OpSymbol.
OpSymbol = string. // A 1, 2 or 3-character string; the set of possible symbols is restricted

```

Example:

```

record Complex;
  re, im: real;
end Complex;

operator '+' (x1, x2: Complex): Complex;
var res: Complex;
begin
  res.re := x1.re + x2.re;
  res.im := x1.im + x2.im;
  return res
end '+';

```

For all overloaded operators parameters are passed by value (as for all predefined operators) and the operator must produce a result. The sole exception is for the assignment operator where the first parameter must be passed by reference and the operator must *not* produce a result.

For overloaded operators the number of parameters in an operator declaration must be the same as that of the predefined operator with the same symbol. For a new operator declaration the number of parameters in an operator declaration must be one or two, depending on whether it is a unary or binary operation.

It is only possible to declare overloaded and new operators in a *module*. The reason is to enable complete overloading resolution statically at compile time to support strong type checking. This is also intended to clearly separate two concepts: objects implementing interfaces (definitions) and abstract data types with associated operators. If necessary type conversion must be used at runtime, for example: to apply the operator procedure “*”($x: X; y: Y$): Z to the generic objects **var** s, t : *object* would require conversion between types, so if s is of type X and t is of type Y then $z := f(X(s), Y(t))$.

Operator declaration can be made available outside the module where it is declared. In that case, it is legal to use those operators in units importing the module in normal expressions, together with the predefined operators. The compiler is responsible for selecting the right version of the operator in each case.

It is possible to define operators in a module to extend an abstract data type. These operators must be defined in terms of the operations already defined in the module where the abstract data type is declared.

Normally, all imported entities should be qualified by the name of the imported unit. This is also possible, but not required, for operators. For example, there are two legal ways to use ‘new addition’ for operands of some type T .

```

module M;
  type {public} T = ...;
  operator {public} "+" ( a, b: T): T; begin ... end "+";
end M.

module Obj;
  import M;
  var x, y : T;
  begin
    x := x + y;           (* like a normal expression *)
    x := x M."+" y;       (* fully qualified, but less conventional *)
  end Obj.

```

An operator procedure cannot be called as a normal function:

```

x := M."+"(x, y); (* not legal; must use expression notation *)

```

6.3.3 Rules governing overloading

The following set of rules applies to overloaded operators:

1. Operators can only be introduced to define previously undefined operations, but *not* to refine previously defined operations
2. The type of at least one operand of an overloaded operator must be a user-defined type (an array type defined without *{math}* modifier, an object type, a procedure type, an enumeration type). It is illegal to introduce user-defined operator versions for ‘basic’ types such as *integer*, *real*, and *boolean*.
3. Specifying an object type with a specified interface (such as *object { D }*) as the operator’s parameter is not allowed. The reason is that it must be possible to resolve operator overloading completely at compile time (i.e. statically).
4. The number of arguments, the precedence of an overloaded operator and the form (prefix or postfix) of unary operators, must be the same as those features for predefined operators with the same symbols.
5. All operators except assignment must produce a result, which may be of any type.
6. It is also possible to overload assignment. In this case, the assignment symbol is considered as a special operator

with the symbol ‘:=’ performing a certain side effect and producing no value. Note that the assignment operator can be used to copy the value of an object of the same type. If it is overloaded with parameters of the same object type then it will be used instead of the predefined := operator for that type. In any case the value is copied by default by the predefined assignment operator semantics.

7. In the overloaded operator for assignment there must be two parameters, and the first one must be passed by reference.
8. The number of operands of a new operator is determined by the number of parameters from the operator declaration. (See section 9).
9. It is legal to specify more than one version of the overloaded and new operators with the same symbol; in that case, the types of the parameters of the corresponding operator declarations must differ from any other operator declaration for the same symbol. (See section 6.3.1)
10. Overloaded operators can only be defined in a module where at least one of the operands is declared.

6.4 Operator Precedence

Four classes of operators with different levels of precedence (binding strengths) are syntactically distinguished when used in expressions. Operators of the same precedence associate from left to right. For example, $x - y - z$ stands for $(x - y) - z$. Operator precedence from highest to lowest is:

1. unary negation operator `~`
2. exponentiation operator `**`
3. multiplication operators `*` **div** **mod** `/`
4. addition operators `+` `-`
5. relations `<` `<=` `=` `#` `>=` `>` **in** **is**

The available operators are listed in the following tables. Some operators are applicable to operands of various types, denoting different operations. In these cases, the actual operation is effectively ‘overloaded’ and the appropriate one to use is identified by the type of the operands. The operands must be expression compatible with respect to the operator, see 12.6.

6.5 Numeric resolution within expressions

An expression consists of a series of evaluations of operators on their operands. For each operator the relationship between the resolution (width) of each of its operands and the result of the operation is defined as follows:

<i>operator</i>	<i>first operand</i>	<i>second operand</i>	<i>result</i>
+	integer{s}	integer{t}	integer{max(s, t)}
−	integer{s}	integer{t}	integer{max(s, t)}
*	integer{s}	integer{t}	integer{s + t}
div	integer{s}	integer{t}	integer{s}
mod	integer{s}	integer{t}	integer{t}
**	integer{s}	integer{t}	integer{s + t}
+	cardinal{s}	cardinal{t}	cardinal{max(s, t)}
−	cardinal{s}	cardinal{t}	cardinal{max(s, t)}
*	cardinal{s}	cardinal{t}	cardinal{s + t}
div	cardinal{s}	cardinal{t}	cardinal{s}
mod	cardinal{s}	cardinal{t}	cardinal{t}
+	real{s}	real{t}	real{max(s, t)}
−	real{s}	real{t}	real{max(s, t)}
*	real{s}	real{t}	real{s + t}
/	real{s}	real{t}	real{s + t}
**	real{s}	real{t}	real{s + t}
+	fixed	fixed	fixed
−	fixed	fixed	fixed
*	fixed	fixed	fixed
/	fixed	fixed	fixed
**	fixed	fixed	fixed

Note: max(s, t) = s, if s > t else t

The compiler implementation is responsible for conserving the integrity of intermediate values during the evaluation of an expression [Compiler].

7 Statements

Statements denote actions. There are elementary and structured statements. Elementary statements are not composed of any parts that are themselves statements. They are the assignment, the procedure call, *return* and *exit* statements. Structured statements are composed of parts that are themselves statements. They are used to express conditional, selective, and repetitive execution. A statement may also be empty, in which case it denotes no action. The empty statement is included in order to relax punctuation rules in statement sequences.

Statement = [Assignment | ProcedureCall | IfStatement | CaseStatement | WhileStatement | RepeatStatement | LoopStatement | ForStatement | **exit** | **return** [Expression { ", " Expression }]].

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

StatementSequence = Statement { "; " Statement }.

Example:

```
temp := a; a := b; b := temp (* swap values in a and b *)
```

7.1 The Assignment Statement

An assignment statement replaces the current value of a variable by a new value specified by an expression. The expression must be assignment compatible with the variable. (See 12.4). The assignment operator is written as ‘:=’ and pronounced as ‘becomes’.

Assignment = Designator { ", " Designator } ":=" Expression { ", " Expression }.

Note that multiple assignments may be made using a single statement. The effect when such statements are evaluated is as follows:

1. Each expression on the right-hand side is evaluated to produce a value.
2. Then the values are assigned in any order to their corresponding designated variables on the left-hand side.

The semantics are the same as for guarded commands [Dijkstra] and support the possibility of execution.

Examples:

```
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 := l;
w[i + 1].name := "John";
t := c;
x, y, z := a, b, c;
```

7.1.1 Indexer Assignments

It is convenient to access the fields within an object as if it were an array using an *indexer*, otherwise such variables are usually indirectly accessed via specific method calls. Indexer access is achieved using a built-in generic definition called “[]” which enables object types to implement this definition. A template for this is:

```
object X;
  operator “[ ]” (i, j: integer): real;
  begin ... (* implements x[i, j] *)
  end “[ ]”;

  operator “[ ]” (i, j: integer; y: real);
  begin ... (* implements x[i, j] := y *)
  end “[ ]”;
...
end X;
```

For indexed access of an object the following abbreviated syntax is allowed:

```
var x: X;

x[i, j] (* instead of x.[i, j] *)
x[i, j] := 3.14; (* instead of x.[i, j, 3.14] *)
```

For an indexer the number of index dimensions allowed depends on the compiler implementation [Compiler].

7.1.2 Abstract Assignments

The notion of an indexer is also used to achieve a so-called abstract assignment for direct access to a field of an object. Here the assignment operator “:=” is dropped and the abstract assignment is reinterpreted as a zero dimensional indexer.

A template for defining an abstract assignment is:

```
object A;
...
  operator “:=” (b: B);
  begin ... (* implements self [ ] := b *)
  end “:=”;
...
end A;
```

This can be then used in a full or an abbreviated form as follows:

```
var a: A; b: B;
...
a[ ] := b; (* full form *)
a := b; (* abbreviated form *)
```

7.2 The Procedure Call

Within a *module* a procedure call invokes a procedure. When it is declared within an *object* a procedure is referred to as a method. In either case it may contain a list of actual parameters which replace the corresponding formal parameters defined in the procedure declaration. (See section 1). The correspondence is established by the relative ordering of the parameters in the actual and formal parameter lists. There are two kinds of parameters: variable and value parameters.

If a formal parameter is a variable parameter, the corresponding actual parameter must be a designator denoting a variable. If it denotes an element of a structured variable, the component selectors are evaluated when the formal/actual parameter substitution takes place, i.e. before the execution of the procedure. If a formal parameter is a value parameter, the corresponding actual parameter must be an expression. This expression is evaluated before the procedure activation, and the resulting value is assigned to the formal parameter.

ProcedureCall = Designator.

Examples:

```
WriteInt(i * 2 + 1)
inc(w[k].count)
t.Insert("John")
```

A method call consists of the name of an object, followed by a period and then the name of a procedure declared within the object type declaration of the object. Within the method the reserved word *self* refers to the object on which the method was called.

A specific procedure call may also be ‘safeguarded’, by prefixing the object with a definition. For example:

```
object T implements I, D; ... end T; var t: T;
```

A client who wants to make specific use of *t*’s interpretation of the services specified by *D* (e.g. as a *supercall*) would then simply call *D*’s methods and fields safeguarded by *t*:

```
D(t).f(..); .. := D(t).x;
```

The order in which the parameters is evaluated during procedure/method invocation is defined in the *MiGOberon Programmers’ Manual* [Compiler].

7.3 The if Statement

```
IfStatement = if Expression then StatementSequence
              {elseif Expression then StatementSequence}
              [else StatementSequence] end.
```

Example:

```
if ("A" <= ch) & (ch <= "Z") then ReadIdentifier
elseif ("0" <= ch) & (ch <= "9") then ReadNumber
elseif (ch = "") or (ch = ' ') then ReadString
else SpecialCharacter
end
```

An *if* statement specifies the conditional execution of guarded statement sequences. The expression preceding a statement sequence is called its guard and its type must be *boolean*. The guards are evaluated in sequence of occurrence; if one evaluates to *true*, its associated statement sequence is executed. If no guard is satisfied, the statement sequence following the symbol *else* is executed, if there is one.

7.4 The case Statement

The *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 whose *case* label list contains the obtained value is executed. The *case* expression must either be of an integer or cardinal type that is expression compatible (see 12.6) with the types of all *case* labels, or both the *case* expression and the *case* labels must be of type *char* or an enumeration. *case* labels are constants, and no value must occur more than once. If the value of the expression does not occur as a label of any *case*, the statement sequence following the symbol *else* is selected, if there is one, otherwise the *UnmatchedCase* exception is

raised (see 7.10.1).

```
CaseStatement = case Expression of Case { "|" Case }
               [ else StatementSequence ] end.
Case = [ CaseLabel { "|" CaseLabel } ":" StatementSequence ].
CaseLabel = ConstExpression [ ".." ConstExpression ].
```

Example:

```
case ch of
  "A" .. "Z": ReadIdentifier (* assumes contiguous encoding of letters*)
| "0" .. "9": ReadNumber
| '"', "'": ReadString
else SpecialCharacter
end

case month of
  Month.Apr, Month.Jun, Month.Sep, Month.Nov: days := 30
| Month.Feb:
  if Leap(year) then days := 29
  else days := 28
  end
else days := 31
end
```

7.5 The while Statement

The *while* statement specifies the repeated execution of a statement sequence while the expression of type *boolean* (its guard) yields *true*. The guard is checked before every execution of the statement sequence and so the statement sequence will be executed zero or more times.

```
WhileStatement = while Expression do StatementSequence end.
```

Examples

```
var i, k, idNumber: integer;
...
while i # 3 do writeln('Hello'); i := i + 1 end
read(idNumber);

while ~Valid(idNumber) do write('Type ID number again '); read(idNumber)
end; (* Valid(idNumber) *)

while i > 0 do i := i div 2; k := k + 1 end

while (t # nil) & (t.key # i) do t := t.left end
```

7.6 The repeat Statement

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

```
RepeatStatement = repeat StatementSequence until Expression.
```

Examples:

```
var idNumber: integer;
repeat
  write ('Type ID number '); read(idNumber) until Valid(idNumber);
...

var i, x: integer; buffer: array 10 of integer;
...

(* convert non-negative value of x to decimal representation *)
i := 0; repeat buffer[i] := x mod 10; x := x div 10; inc(i) until x = 0;

(* write out digit characters in correct order *)
repeat dec(i); write(char(buffer[i] + integer("0"))) until i = 0
```

7.7 The for Statement

A *for* statement specifies the repeated execution of a statement sequence for a fixed number of times while a progression of values is assigned to a variable of integer or cardinal type called the control variable of the *for* statement.

ForStatement = **for** ident ":"=" Expression **to** Expression [**by** ConstExpression] **do** StatementSequence **end**.

The statement

for v := low **to** high **by** step **do** statements **end**

is equivalent to

```
v := low; temp := high;
if step > 0 then
  while v <= temp do statements; v := v + step end
else
  while v >= temp do statements; v := v + step end
end
```

The value of the expression *low* must be assignment compatible with *v* and that of *high* must be expression compatible with *v*. The value of *step* must be a non-zero constant expression of an integer or cardinal type. If *by step* is omitted, then *step* defaults to the value 1.

Example:

```
var i : integer;
...
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
```

7.8 The loop Statement

A *loop* statement specifies the repeated execution of a statement sequence. It is terminated upon execution of an *exit* statement within that sequence.

LoopStatement = **loop** StatementSequence **end**.

Example:

```
loop (* copy integers from input to output until 0 is typed *)
  read(i);
  if i = 0 then exit end; write(i)
end
```

loop statements are useful for expressing repetitions with several exit points or cases where the exit condition occurs naturally in the middle of the repeated statement sequence.

An exit statement is denoted by the symbol *exit*. It specifies termination of the enclosing *loop* statement and continuation with the statement following that *loop* statement. An *exit* statement is contextually, although not syntactically, associated with the *loop* statement which contains it.

7.9 The return Statement

A *return* statement is used within procedures and activities. In a procedure it is used to return a value from back its caller. It is denoted by the symbol *return*, followed by an expression if the procedure is a function procedure. The type of the expression must be assignment compatible (see 12.4) with the result type specified in the procedure. 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.

7.10 The Block Statement

The block statement allows the grouping together of logically related statements and the introduction of exception handlers. Block statements can be nested.

```
BlockStatement =  
  do [ Modifiers ]  
    [ StatementSequence ]  
    { ExceptionHandler }  
    [ CommonExceptionHandler ]  
    [ TerminationHandler ]  
  end.
```

The statement sequence within the block is carried out.

7.10.1 Exception Handling

If an exception occurs then the exception handlers are tried in the order in which they appear textually until one that matches the exception is found or the general exception is reached. The statement sequence corresponding to the exception name is then carried out.

```
ExceptionHandler = on ExceptionName { ", " ExceptionName } do StatementSequence.  
CommonExceptionHandler = on exception do StatementSequence.  
TerminationHandler = on termination do StatementSequence.
```

Exception names take the form of predefined identifiers and include:

- ZeroDivision: division by zero
- Overflow: value does not lie within $\min(type) \dots \max(type)$
- OutOfRange: array index out of bounds
- NilReference: uninitialized array/object/activity/protocol instance
- UnmatchedCase: control flow reached missing *else* in *case* statement
- Conversion: invalid type conversion (not guarded by '*t is type*')
- IncompatibleSizes: math arrays have incompatible sizes

Extra information about the exception can be accessed by calling the predefined function *reason*. This causes the runtime system to return a string which explains the reason for the exception, and possibly the system context, to aid program development. For example:

```
do  
  Statements  
on T1, T2 do  
  (* reason returns a string containing system defined info and the name T1 or T2*)  
  s := reason  
on exception do  
  (* reason returns a string with the name of the exception thrown and possible system-defined information*)  
  s := reason  
end
```

If *reason* is called outside the scope of an exception it returns the current error or warning status information from the runtime system. See also [Compiler].

The following form acts as a 'catch all':

```
do  
  Statements  
  ...  
on exception do  
  CatchAll  
end
```

means that *CatchAll* is only executed if an exception has occurred but no textually earlier exception clause in the block matched the exception.

Example:

```
var idNumber: integer; idValid: boolean;
begin
  do
    read(idNumber);
    if Valid(idNumber) then
      idValid := true; Process(idNumber)
    else
      idValid := false (* wrong number *)
    end
  on exception do
    idValid := false (* wrong sort of characters typed *)
  end
end
```

8 Procedure and Method Declarations and Formal Parameters

A procedure declaration consists of a procedure heading and a procedure body. The heading specifies the procedure's identifier and its formal parameters, if any. The body contains declarations and statements. The procedure identifier is repeated at the end of the procedure declaration. A procedure declared within an object is called a method.

There are two kinds of procedures: proper procedures and function procedures. The latter are activated by a function designator as a constituent of an expression and yield a result that is an operand of the expression. Proper procedures are activated by a procedure call. A procedure is a function procedure if its formal parameters specify a result type. The body of a function procedure must contain a return statement that defines its result.

All constants, variables, types, and procedures declared within a procedure body are local to the procedure. Since procedures may be declared as local items too, procedure declarations may be nested (subject to implementation restrictions). The call of a procedure within its declaration implies recursive activation.

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

```
ProcedureDeclaration = ProcedureHeading [ ProcImplementationClause ] ";" [ ProcedureBody ";" ].
ProcedureHeading = procedure [ Modifiers ] ProcedureName [ FormalParameters ].
ProcedureBody = Declarations UnitBody SimpleName.
FormalParameters = "(" [ FPSection { ";" FPSection } ] ")" [ ":" FormalType ].
FPSection = [ var ] ident { "," ident } ":" FormalType.
```

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 rules for the correspondence between formal and actual parameters are as follows. Let T_f be the type of a formal parameter f (not an open array) and T_a the type of the corresponding actual parameter a . For variable parameters, T_a must be the same as T_f , or T_f must be an *object* type and T_a must be derived from T_f . For value parameters, a must be assignment compatible with f . (See 12.4).

If T_f is an open array, then a must be array compatible with f . (See 12.5). The lengths of f are taken from a .

8.1 Procedure Modifiers

A modifier may optionally occur after the reserved word *procedure* to denote its nature. The following modifiers are defined:

- *private*: the procedure is only visible in the scope in which it is declared; this is the default.
- *public*: the procedure is visible in the scope in which it is declared and within any construct that imports the construct in which it is declared.
- *sealed*: the procedure may not be further redefined (overridden),

The inverse of being sealed is referred to as being *open*

Examples:

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

procedure {private} WriteHex(x: integer);
(* precondition: 0 <= x < 100000H *)
var i: integer; buf: array 5 of integer;
begin
  i := 0;
  repeat buf[i] := x mod 10H; x := x div 10H; inc(i) until x = 0;
  repeat dec(i);
    if buf[i] < 10 then write(char(buf[i] + integer("0")))
    else write(char(buf[i] - 10 + integer("A")))
    end
  until i = 0
end WriteHex;

procedure log2(x: integer): integer; (* precondition: x > 0 *)
var y: integer;
begin
  y := 0; while x > 1 do x := x div 2; inc(y) end;
  return y
end log2;
```

8.2 Method-level Protection

A finer granularity of protection is also available at the method level. Any method within a protected object can be ‘shared’ by marking its declaration with the `{shared}` modifier at the beginning of the method body. When running a block of shared statements within an object only a ‘share’ of the total lock can be owned by the accessing activity, and any number of activities may get their own share. However, statements within such a shared statement sequence must not modify (write to) the object’s data i.e. method-level shared accesses must be ‘read-only’. This provides for efficient access to read object data by many activities concurrently without the overhead of each on in turn having to own the object-level lock.

10 Mathematical extensions

Mathematical extensions of MiGOberon let arrays be used in a more convenient way for writing applications where multidimensional algebra is used.

10.1 Data structures

For general information about arrays see 5.3.5. Arrays to be used in mathematical extensions should be defined with special `{math}` modifier.

ArrayType = **array** "{" **math** "}" Length { "," Length } **of** Type.

10.1.1 Expression arrays

Expression array has the following structure:

ExpressionArray = "[" ArrayFactor "]"
ArrayFactor = ExpressionArray { "," ExpressionArray } | Expression { "," Expression }.

An expression array may be used for creating arrays or assigning values to it. An expression array is written as a comma-separated list of expressions, enclosed by braces "[" and "]". The rank of the constructed array will be equal to the number of enclosed braces. The n -th expression specifies the value of the $n-1$ -th array component.

Example:

```
var a : array {math} 2, 3 of integer;
begin
  a := [[1, 2, 3], [4, 5, 6]];
  (* it is equal to:
    a[0, 0] := 1;
    a[0, 1] := 2;
    a[0, 2] := 3;
    a[1, 0] := 4;
    a[1, 1] := 5;
    a[1, 2] := 6;
  *)
  ...
```

10.2 Indices

If a is a n -dimensional array, then a subset of its elements can be defined as $a[\text{index}_0, \dots, \text{index}_{n-1}]$.

In this context index_i (which is responsible for a subset in i -th dimension) can be either a simple index, a range, a numerical vector, or a boolean vector. All array accesses are checked at run-time. If array a is dynamic, it has to be initialized, otherwise a *NilReference* exception will be thrown at run-time.

10.2.1 Simple indices

A simple index is an expression of *integer* or *cardinal* type which value is out of set $\{0, \dots, \text{len}(a, i) - 1\}$. This kind of indices is acceptable for usual arrays too and provides a single element access. An attempt to use an index that doesn't lie in the set $\{0, \dots, \text{len}(a, i) - 1\}$ causes an *OutOfRange* exception to be thrown.

Example:

```
var
  b : array {math} 8,8,8 of real;
  k : real;
begin
  ...
  k := b[2, 3, 1];
```

11 Modules

A MiGOberon program may be textually partitioned into modules, each of which can be compiled separately.

CompilationUnit = Module ". " .

Within a program the module is an implementer of functionality.

11.1 The Module

A *module* has a dual nature, it declares a syntactic container for logically cohesive program declarations and it simultaneously declares an object which is managed by the system. So the module provides the mechanism for the textual partitioning of a source program and also the dynamic loading at execution time of a part of a program, in the form of an instantiated object.

Any number of dynamically created objects may have their life cycles managed by a program, however only a single instance of each module's object may be instantiated by the system at any given time. For this reason the module is also ideal for implementing abstract data types. Nesting of module declarations is not allowed.

```
Module = module [ Modifiers ] ModuleName [ ImplementationClause ] ";" [ ImportDeclaration ]
ModuleDeclarations ( UnitBody | end ) SimpleName.
Modifiers = "{" IdentList "}".
ModuleDeclarations = { SimpleDeclaration | ProcedureDeclaration | OperatorDeclaration }
ImportedName = ModuleName.
```

Each *module* has a unique name and constitutes a text that may be separately compiled as a unit. Optionally a *module* may *implement* one or more *definitions*. (See section 2). In this case the distinct facets of the object are defined separately in *definition* units which provide an abstract interface. A *module* may optionally *import* elements from one or more other

implementations, that is, gain access to their scope and make possible the aggregation of their content. By using the *as* clause it is also possible to rename all entities as they are imported. This can be used to avoid name clashes and/or to simplify long external names to promote program readability.

Example:

```
import System.Console as S; ... S.WriteLine('Hello'); (* equivalent to System.Console.WriteLine('Hello') *)
```

A module may optionally contain

• • • •

Other textual units i.e. *definitions*, *implementations* and objects Simple declarations of constants, types, variables, and procedures Operator declarations, for defining user defined operators Activity declarations, for defining activities within the *module* on instantiation

Examples:

```
module Small; begin  
write ('Hello World') end Small.
```

```
module BodyMassIndex; (* calculate body mass index *) var height, weight, bmi: real;  
begin  
write('weight in kg? '); read(weight); write('height in m? '); read(height); bmi := weight / (height * height); write(' body mass index is', bmi : 6: 2);  
if bmi < 19.0 then  
write('too thin') elsif bmi < 27.0 then  
write('OK')  
else  
write('too fat')  
end end BodyMassIndex.  
definition D; ...end D. definition E; ...end E.  
module M; import D, E;  
var a: object{D, E}; (* object is one that implements both D and E *)  
... end M.
```

11.2.1 Inheritance and Multiple Inheritance

There are two kinds of inheritance supported in MiGOberon: refinement and aggregation. Refinement is the inheritance of an interface definition whilst aggregation is the inheritance (reuse) of (fragments of) an existing implementation. All object declarations that do not explicitly refine some other object are deemed to refine *object*. Thus all objects (directly or indirectly) refine *object*. If an object *B* refines an object *A*, then *B* is said to be ‘derived from’ *A*; for further details see section 10.2.3.

Multiple inheritance is characterized by the possibility to refine from multiple definitions and/or to aggregate from multiple implementations. In MiGOberon there is no ambiguity associated with multiple inheritance, due to the use of qualified identifiers for naming (see 5.1).

11.2.2 Polymorphism

Polymorphism involves the selection of the appropriate method to invoke at execution time, depending on the type of the variable that it is to be acted upon. There are two concepts:

- 1) an object of type *T* is required here, and
- 2) an object is required here that implements an interface definition *D*

MiGOberon emphasizes the second more general concept (2 above) and goes further by allowing the specification of multiple definitions (so called ‘facets’ of the object’s overall interface) and so in this context polymorphism means ‘an object is required here that implements *D1* and *D2* and ...’.

13 Definition of Terminology

13.1 Numeric Types

The numeric types are:

- Integer types integer or integer{width}
- Cardinal types cardinal or cardinal{width}
- Real types real or real{width}

13.2 Same Types

Two variables a and b with types Ta and Tb are of the *same* type if

- Ta and Tb are both denoted by the same type identifier, or
- Ta is declared to equal Tb in a type declaration of the form $Ta = Tb$, or
- a and b appear in the same identifier list in a variable, object field, or formal parameter declaration and are not open arrays.

13.3 EqualTypes

Two types Ta and Tb are *equal* if

- Ta and Tb are the same type, or
- Ta and Tb are open array types with equal element types, or
- Ta and Tb are procedure types whose formal parameter lists match.

13.4 Assignment Compatible

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

- Te and Tv are the same type;
- Within each of the type families integer, cardinal, real, set, char an expression of type Te may be assigned to a variable v whose type Tv is large enough (defined by its width) to hold the set of values of type Te ;
- Tv is a procedure type and e is *nil*;
- Tv is a procedure type and e is the name of a procedure whose formal parameters match the signature of Tv

13.5 ArrayCompatible

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

- Tf and Ta are the same type, or
- Tf is an open array, Ta is any array, and their element types are array compatible

13.6 Compatible for Expressions and Operator Overloading

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), for example: $op1 > op2$. The table also implicitly defines the sets of operand combinations that are supported for operator overloading.

<i>Operator</i>	<i>First operand (op1)</i>	<i>Second operand (op2)</i>	<i>Result type</i>
$+ - ***$	integer{m}	integer{n}	max of integer{m} and integer{n}
$+ - ***$	cardinal{m}	cardinal{n}	max of cardinal{m} and cardinal{n}
$+ - ***$	real{m}	real{n}	max of real{m} and real{n}
$/$	real{m}	real{n} pre: op2 # 0	max of real{m} and real{n}
$+ - *$	set{m}	set{n}	max of set{m} and set{n}
div mod	integer{m}	integer{n} pre: op2 # 0	max of integer{m} and integer{n}
div mod	cardinal{m}	cardinal{n}	max of cardinal{m} and cardinal{n}
or & ~	boolean	boolean	boolean
$= \# < <= > >=$	integer{m}	integer{n}	boolean
$= \# < <= > >=$	cardinal{m}	cardinal{n}	boolean
$= \# < <= > >=$	real{m}	real{n}	boolean
$= \# < <= > >=$	enumeration T	enumeration T	boolean
$= \# < <= > >=$	char	char	boolean
$= \# < <= > >=$	character array	character array	boolean
$= \# < <= > >=$	string	string	boolean
$= \#$	boolean	boolean	boolean
$= \#$	set	set	boolean
$= \#$	procedure type T	procedure type T	boolean
$= \#$	nil	nil	boolean
in	integer	set	boolean

is object object type boolean

13.7 Matching Formal Parameter Lists

Two formal parameter lists match if

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

14 Predefined Procedures

The following table lists the predefined procedures. Some are generic procedures, i.e. they apply to several types of operands. Within the specifications v stands for a variable, x and n for expressions, and T for a type. The names of the predefined procedures can also be written entirely in upper-case letters.

Name	Argument(s) type(s)	Result type	Purpose
abs(x)	integer, cardinal or real	type of x	absolute value of x
assert(b)	b: boolean	none	if ~b terminate
assert(b, n)	b: boolean; n: integer or cardinal	none	if ~b terminate, report n to environment
cap(x)	x: char	char	corresponding capital letter precondition: x is a letter
copy(x, v)	x: string; v: character array	none	$v := x$
copy(v, x)	x: string; v: character array	none	$x := v$
copyvalue(v)	v: ref object	value object	dereference an object
dec(v)	v: integer, cardinal or enumeration type	none	$v := v - 1$
dec(v, n)	v: integer, cardinal or enumeration type, none n: integer or cardinal type	none	$v := v - n$
excl(v, x)	v: set; x: integer or cardinal type	none	$v := v - \{x\}$
halt(n)	n: integer or cardinal const	none	terminate program execution
inc(v)	v: integer, cardinal or enumeration	none	$v := v + 1$
inc(v, n)	v: integer, cardinal or enumeration, n: integer or cardinal type	none	$v := v + n$
incl(v, x)	v: set; x: integer or cardinal type	none	$v := v + \{x\}$
len(v, n)	v: array; n: integer or cardinal const	integer	length of v in dimension n (first dimension = 0)
len(v)	v: array	integer	equivalent to len(v, 0)
len(v)	v: string	integer	number of characters in string v (see 5.3.6)
low(x)	x: char	char	corresponding lower-case letter precondition: x is a letter
max(T)	integer	integer	maximum value of type integer{w}
max(T)	cardinal	cardinal	maximum value of type cardinal{w}
max(T)	enumeration	enumeration	maximum value of the enumeration
max(T)	char{w}	integer	maximum character
max(T)	real{w}	real	maximum value of type real{w}
max(T)	set{w}	integer	maximum element of a set{w}
min(T)	integer	integer	minimum value of type integer{w}
min(T)	enumeration	enumeration	minimum value of the enumeration
min(T)	char{w}	integer	minimum character
min(T)	real{w}	real	minimum value of type real{w}
min(T)	set{w}	integer	0
odd(x)	x: integer	boolean	$x \bmod 2 = 1$
pred(x)	x: integer	integer	$x - 1$, pre: $x \neq \min(\text{integer})$
pred(x)	x: enumeration	type of x	predecessor enumeration value, pre: $x \neq \min(\text{enumeration})$
pred(x)	x: char	char	predecessor char, pre: $x \neq \min(\text{char})$
reason	none	string	returns system information on current exception
size(T)	any type	integer	number of bytes required by T
succ(x)	x: integer or cardinal	integer	$x + 1$, pre: $x \neq \max(\text{integer})$
succ(x)	x: enumeration	type of x	successor enumeration value, pre: $x \neq \max(\text{enumeration})$
succ(x)	x: char	char	successor char, pre: $x \neq \max(\text{char})$

In *assert(x, n)* and *halt(n)*, the interpretation of n is implementation specific. (See [Compiler]).

15 Input and Output Procedures

The language includes built-in features for simple textual input and output. Conceptually, reading and writing corresponds to receiving and sending tokens from and to the predefined activities *standard input* and *standard output* respectively. For convenience, predefined procedures in a similar style to Pascal are provided for reading and writing text. The procedures for inputting text are *read* and *readln* and for outputting are *write* and *writeln*. All input and output is to texts which are implicitly assumed to be represented as lines of characters delimited by end of line markers.

15.1 Parameters and special syntax

The procedures are used with a non-standard syntax for their parameter lists. This allows for a variable number of parameters which may be of various data types. Parameters of type *char* require no data type conversion, however for other types such as integer, real, etc the data transfer includes an implicit data type conversion.

15.2 Input Procedures

15.2.1 The read procedure

The form of the *read* procedure is

```
read (v1, ..., vn)
```

It may have one or more parameters, each of which is a value of some basic data type. If *v* is a value of type *char* then *read(v)* transfers the next character from the input text to *v*. If *v* is a value of type *integer*, *cardinal* or *real* then *read(v)* implies the reading of a sequence of characters from the input text and assignment of that number to *v*. Preceding blanks and line markers are skipped and discarded.

15.2.2 The readln procedure

The form of the *readln* procedure is

```
readln(v1, ..., vn)
```

readln has the same functionality as *read* except that after reading *vn* all remaining characters on the line are skipped up to and including the next end of line marker.

15.3 Output Procedures

15.3.1 The write procedure

The form of the *write* procedure is

```
write (p1, ..., pn)
```

It may have one or more parameters, each of which has the form

```
e : e:m or e:m:n
```

Where *e* represents the value to be output and *m* and *n* are field-width specifiers. If the value of *e* requires less than *m* characters for its representation then blanks (spaces) are output to ensure that a total of exactly *m* characters are written. If *m* is omitted an implementation-defined default value will be assumed. The form *e:m:n* is only applicable to numbers of type *real*. (See below).

The *write* procedure parameters can be of type *char*, *string*, *boolean*, *integer*, *cardinal* and *real*.

- If *e* is of type *char* then *write (e : m)* writes out *m* – 1 spaces followed by the character contained in *e*. If *m* is omitted then only the character is written.
- If *e* is of type *string* then *write (e : m)* writes the characters of the string, preceded by blanks to ensure a total field width of *m*.
- If *e* is of type *boolean* then either the word *true* or *false* is written, preceded by blanks to ensure a total field width of *m*.
- If *e* is of type *integer* or *cardinal* then the decimal representation of the number *e* will be written, preceded by blanks to ensure a total width of *m*.

- If e is of type *real* then the decimal representation of the number e will be written, preceded by blanks to ensure a total width of m . If the parameter n is missing a floating point representation consisting of a coefficient and a scale factor will be written. If n is present then a fixed-point representation with n digits after the decimal point is provided.

15.3.2 The *writeln* procedure

The form of the *writeln* procedure is:

```
writeln (v1, ..., vn)
```

writeln has the same functionality as *write* except that after writing vn an end of line marker is written.

15.3.3 Default values of widths in *write* and *writeln*

The default field width for *write* and *writeln* procedure parameters depends on the type of the parameter, the default widths are:

- *char* default field width 1
- *string* default field width is the length of the string
- *boolean* default field width is 6
- *integer* default field width is 20
- *cardinal* default field width is 20
- *real* default field width is 20

16 Syntax

// MiGOberon Syntax in EBNF

// 1. Program and program units

CompilationUnit = { ProgramUnit "." }. ProgramUnit = (Module | Definition | Implementation | Object).

// 2. Modules

Module = **module** [Modifiers] ModuleName [ImplementationClause] ";"
 [ImportDeclaration]
 ModuleDeclarations
 (UnitBody | **end**) SimpleName.

Modifiers = "{" IdentList "}".

ModuleDeclarations = { SimpleDeclaration ";" | ProcedureDeclaration | OperatorDeclaration }.

ImportDeclaration = **import** Import { "," Import } ";".

Import = ImportedName [:= ident].

ImportedName = ModuleName.

UnitBody = **begin** [StatementSequence] **end**.

// 6. Declarations

SimpleDeclaration = (**const** [Modifiers] { ConstantDeclaration ";" }
 | **type** [Modifiers] { TypeDeclaration ";" }
 | **var** [Modifiers] { VariableDeclaration ";" }).

ConstantDeclaration = ident "=" ConstExpression.

ConstExpression = Expression.

TypeDeclaration = ident "=" Type.

VariableDeclaration = IdentList ":" Type.

// 7. Types

Type = (TypeName ["{" Width "}"] | EnumType | ArrayType | ProcedureType | RecordType).

Width = ConstExpression.

ArrayType = **array** Length { "," Length } **of** Type.

Length = (ConstExpression | "*").

EnumType = "(" IdentList ")".

ProcedureType = **procedure** [ProcedureTypeFormals].

ProcedureTypeFormals = "(" [PTFSection { ";" PTFSection }] ")" [":" FormalType].

PTFSection = [**var**] FormalType { "," FormalType }.

FormalType = { **array** "*" **of** } TypeName).

RecordType = record { VariableDeclaration ";" } end ident.

// 8. Procedures & operators

ProcedureDeclaration = ProcedureHeading [ProcImplementationClause] ";" [ProcedureBody ";"].

ProcedureHeading = **procedure** [Modifiers] ProcedureName [FormalParameters].

```

ProcedureBody = Declarations UnitBody SimpleName.
FormalParameters = "(" [ FPSection { ";" FPSection } ] ")" [ ":" FormalType ].
FPSection = [ var ] ident { "," ident } ":" FormalType.
OperatorDeclaration = operator [ Modifiers ] OpSymbol [ FormalParameters ] ";" OperatorBody ";".
OperatorBody = Declarations UnitBody OpSymbol.
OpSymbol = string. // A 1,2,3-character string; the set of possible symbols is restricted

```

```

// 9. Statements
StatementSequence = Statement { ";" Statement }.
Statement = [ Assignment | ProcedureCall | IfStatement | CaseStatement
             | WhileStatement | RepeatStatement | LoopStatement | ForStatement
             | exit | return [ Expression { "," Expression } ] | BlockStatement ].
Assignment = Designator { "," Designator } "!=" Expression { "," Expression }.
ProcedureCall = Designator.
IfStatement = if Expression then StatementSequence
             { elseif Expression then StatementSequence }
             [ else StatementSequence ] end.
CaseStatement = case Expression of Case { "|" Case } [ else StatementSequence ] end.
Case = [ CaseLabel { "," CaseLabel } ":" StatementSequence ].
CaseLabel = ConstExpression [ ".." ConstExpression ].
WhileStatement = while Expression do StatementSequence end.
RepeatStatement = repeat StatementSequence until Expression.
LoopStatement = loop StatementSequence end.
ForStatement = for ident ":" Expression to Expression [ by ConstExpression ] do StatementSequence end.
BlockStatement = do [ Modifiers ] [ StatementSequence
                       { ExceptionHandler } [ CommonExceptionHandler ] [ TerminationHandler ] end.
ExceptionHandler = on ExceptionName { "," ExceptionName } do StatementSequence.
CommonExceptionHandler = on exception do StatementSequence.
TerminationHandler = on termination do StatementSequence.

```

```

// 10. Expressions
Expression = SimpleExpression [ ( "=" | "#" | "<" | "<=" | ">" | ">=" | in ) SimpleExpression ] | Designator is TypeName.
SimpleExpression = [ "+" | "-" ] Term { ( "+" | "-" | or ) Term }.
Term = Factor { ( "*" | "/" | div | mod | "&" ) Factor }.
Factor = number | CharConstant | string | nil | Set | Designator | new TypeName [ "(" ActualParameters ")" ]
        | "(" Expression ")" | "~" Factor | "!" Factor | Factor "***" Factor.
Set = "{" [ SetElement { "," SetElement } ] "}".
SetElement = Expression [ ".." Expression ].
ExpressionArray = "[" ArrayFactor "]".
ArrayFactor = ExpressionArray { "," ExpressionArray } | Expression { "," Expression }.
ExpressionRange = Expression | Range.
Range = [Expressions ] ".." [Expression ] [ "by" Expression ] .
Designator = Instance | TypeName "(" Expression [ "," Size ] ")"
            | Designator "[" ExpressionRange { "," ExpressionRange } "]" | Designator "(" [ ActualParameters ] ")" |
            Designator "." MemberName

```

```

// Conversion // Dereference // Array element(s) // Function call // Member selector
Instance = ( InstanceName ).
Size = ConstantExpression.
ActualParameters = Actual { "," Actual }.
Actual = Expression [ "{" [ var ] FormalType }" ]. // Argument with type signature

```

```

// 11. Constants
number = (whole | real) [ "{" Width }" ].
whole = digit {digit} | digit {hexDigit} "H".
real = digit {digit} "." {digit} [ ScaleFactor ].
ScaleFactor = "E" [ "+" | "-" ] digit {digit}.
HexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
CharConstant = "'" character "'" | "'" character "'" | digit { HexDigit } "X".
string = '"' { character } '"' | '"' { character } '"'.
character = // Any character from the alphabet except the current delimiter character

```

```

// 12. Identifiers & names
ident = ( letter | "_" ) { letter | digit | "_" }.
letter = "A" | ... | "Z" | "a" | ... | "z" | // any other "culturally-defined" letter
IdentList = ident { "," ident }.
QualIdent = { ident "." } ident.
ModuleName = QualIdent.
NamespaceName = QualIdent.
ObjectName = QualIdent.
TypeName = QualIdent.

```

```
ExceptionName = QualIdent.  
InstanceName = QualIdent.  
ProcedureName = ident.  
MemberName = ( ident | OpSymbol ).  
SimpleName = ident.
```