GARDENS POINT MODULA
LANGUAGE REFERENCE MANUAL

Contents

About this manual

The material in this manual was prepared by John Gough, with contributions from John Hynd and Mike Roggenkamp. The manual was prepared camera-ready, using the LaTeX document processing system.

The material in this manual follows the order of presentation of topics in the *Report on the Modula-2 language* in Wirth's *Programming in Modula-2*, (Springer-Verlag).

The railroad syntax diagrams are adapted from an appendix in the book *Modula-2: a second course in programming* by Gough and Mohay (Prentice-Hall 1988), and are used by permission of the authors.

The date of this version of the manual is

16th July 1992

About gardens point modula

gardens point modula is a product of the programming language and systems group of the **Queensland University of Technology**. Implementations are currently available for 11 separate 32-bit machine types, and the demonstration version **gpm-pc** which emulates a 32-bit machine on any "industry standard pc".

Responsibility for design and implementation of the current versions is as follows:

John Gough (system architecture)

John Hynd (project manager)

Diane Corney, Christina Cifuentes and Peter Kolb (gpm-pc)

John Chalk (the **gpmake** utility)

Michael Roggenkamp (some libraries)

gpm is an entirely new implementation of the Modula-2 language. It inherits neither code nor data structures from any of the previous compilers associated with its authors. It has been designed for the computer architectures of the current generation, particularly those using the reduced instruction set philosophy.

Chapter 1

Introduction

This manual is a concise guide to the language Modula-2, or simply *Modula*. The manual does not set out to be a guide to learning to program in the language — for that you should see any one of a number of textbooks on the subject.

There is a slight bias in the presentation, in that there is frequent reference to the **gardens point modula** (henceforth **gpm**) version of Modula. Attention is drawn to the small number of places where **gpm** does not yet fully conform to the emerging ISO draft standard. Likewise, warnings are given regarding those features that are new in the language, and may hence cause problems of portability for users who are using old compilers as well as **gpm**. The final chapter of this manual collects in one place all of the changes that have taken place since the first edition of Wirth's *Programming in Modula-2*.

gpm is available for a large number of different machines, and behaves identically (almost) for all versions. The only differences are due to smaller table sizes in the PC-demonstration version, and the absence of some libraries in the PC version. Some versions have additional command line options, and there are some limitations in the sharing of data files between versions that use different byte packing conventions, or different file formats.

What this manual contains

This manual contains details of the language Modula as implemented by **gpm**. It contains a guide to the syntactic rules that govern the legality of programs, and an explanation of the semantics rules that programs must obey also. The syntax is given in both *EBNF* and *railroad diagram* forms.

This manual also contains examples of the use of various constructs, and points out some of the subtle consequences of the *fine print* details of the language semantics. There is some deliberate repetition in the details, so that important points on any particular topic may be found with a single lookup. For example, details regarding the use of anonymous types are given in the section on *variable declarations* and also in the section that deals with *type compatability*.

What this manual doesn't contain

This manual is not a textbook, and neither is it a technical reference manual for **gpm**. The separate *User Guide* contains the following important information —

- how to get started with **gpm**
- command line parameters
- helpful hints on using **gpm**
- error messages during compilation
- error messages at runtime
- how to do post-mortem debugging

The separate *Technical Reference Manual* contains the following further information —

- compiler limits
- representation issues
- detailed explanation of error messages
- how to interface to other languages
- interface definitions for the libraries

Finally, the *Release notes* that come with your particular version of **gpm** contain the following information —

- latest information on new libraries
- latest information on new features
- notification of any recent bug fixes
- a list of any known bugs (oops)
- known differences between versions

Chapter 2

Syntax and notation

Modula programs are sequences of symbols belonging to the alphabet of Modula. This alphabet consists of 41 keywords, a number of special symbols, some standard identifiers and user declared identifiers. To describe the syntax in a precise and compact way, a notation called Extended Backus-Naur Form (EBNF) will be used. This form describes the structure of the various phrases from which valid programs are constructed.

Please note from the outset that the syntactic rules of the language do not completely specify which programs are actually valid. As well as obeying the syntax rules that are detailed here, there are other *semantic* rules that must be obeyed also. These semantic rules specify such things as the fact that identifiers must be declared before they are used, and so on. It is just a matter of convenience that the rules are separated into two kinds.

EBNF consists of **replacement rules** that give the relationship between different kinds of **symbols**. There are two kinds of symbols — **terminal symbols**, which have no further structure at this level of description; and **syntactic categories** which may be further expanded. Because of the possibility of further expansion, syntactic categories are also called **non-terminal symbols**.

Terminal symbols consist of the special symbols such as := and the punctuation symbols, the keywords, and certain **lexical categories**. Lexical categories are symbols such as *number*, and *literalString*. These symbols have properties, such as a numeric value, or a string value which are important for the meaning of the program, but not important for syntactic correctness. For example, an assignment statement of the form

```
strVar := "some literal string";
```

is syntactically correct no matter what the identifier value is, or the literal string value. For *semantic* correctness the identifier must denote a variable that is declared as an array of characters and which is at least as long as the length of the literal string.

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

Examples:

syntactic categories - *Program, Statement*

lexical categories - *identfier, number, literalString.*

The EBNF notation specifies a set of *productions* or replacement rules, that show how each particular syntactic category may be built up from other syntactic categories and the symbols of the Modula alphabet. In any production, the phrase on the right-hand-side of the arrow symbol is a possible expansion of the syntactic category on the left-hand-side.

Braces, brackets and parentheses, the arrow sign, the solid vertical bar, and the fullstop are all special symbols of the notation. Square brackets indicate that the enclosed form is optional. Braces (curly brackets) indicate the enclosed form may be repeated zero, one or more times. Parentheses (round brackets) group terms that are alternatives. The solid vertical bar separates alternatives and the fullstop marks the end of the production. If there is a need to use one of the special symbols literally, it is enclosed in quote symbols. Consider the following example —

ValueList → "{" **Element** { , **Element** } "}".

This states that a *ValueList* may be expanded as a list of one or more comma-separated *Element*s, enclosed in literal braces (curly brackets). Note that the outer braces are literal, while the inner ones denote repetition.

2.1 How to read the syntax diagrams

Syntax diagrams provide an alternative to EBNF in expressing the syntax of the language. These diagrams, sometimes called *railroad diagrams*, indicate which sequences of symbols are valid. Any path with starts at the entry point on the left and finishes at the exit point on the right spells out a valid phrase, provided that it only travels in the direction of the arrows.

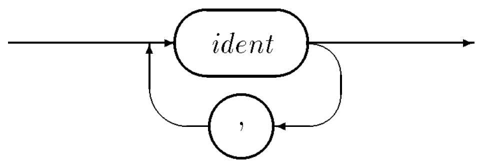
Rectangular boxes correspond to syntactic categories that appear elsewhere in the syntax diagrams. The oval and round boxes contain terminal symbols of the grammar such as **keywords**, special symbols, or the lexical categories *ident, number* and *litstring.*

Example:

The non-terminal symbol **IdentList** is used throughout the EBNF, denoting a comma-separated list of identifiers. The production rule is —

IdentList → *ident* { , *ident*}.

These identifier lists are always shown explicitly in the railroad diagrams, and look similar to this example —



2.2 Lexical categories and rules

The ASCII character set is assumed and the following lexical rules must be observed.

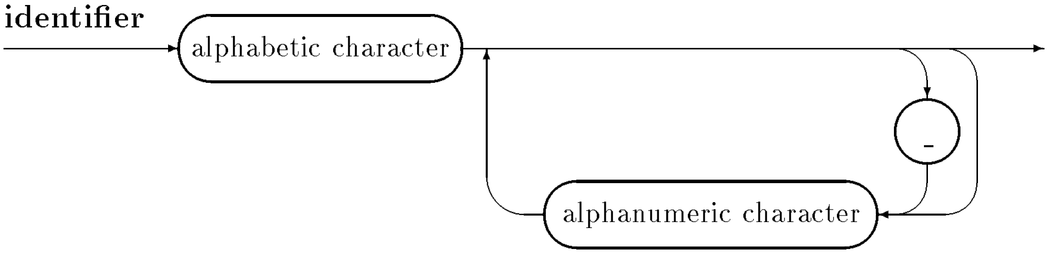
Blanks must not occur within symbols (except in strings). Blanks and line breaks are ignored unless they are essential to separate two consecutive symbols.

Identifiers are sequences of letters, digits and lowlines. The first character must be a letter. The *lowline* character "underscore" may be used in multi-word identifiers to separate words, to increase readability. However, in Modula, it is traditional to use upper and lower case rather than lowlines for this purpose, and this manual follows the Modula convention. **gpm** currently does not allow identifiers to start or end with a lowline, and does not permit lowlines to be adjacent.

ident → **letter**{**letter** | **digit**}.

Examples: count Modula get_Symbol

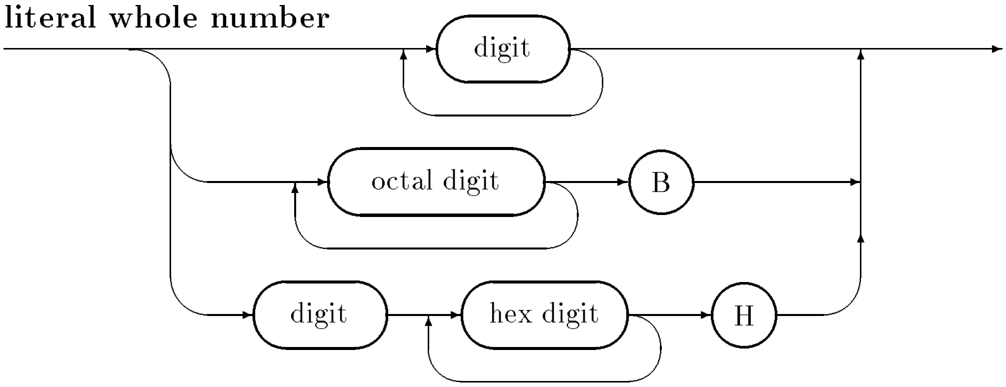
identifier



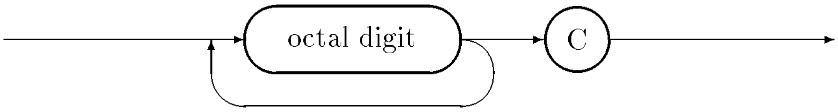
Numbers are whole numbers or real numbers. Whole numbers are sequences of digits. Whole numbers may be represented by decimal, octal or hexadecimal values. In each instance the first character of the number must be a digit; octal representations are terminated by a B or C and a hexadecimal representation is terminated by a H. The use of the C form indicates that the value is the octal ordinal representation of a character constant.

number	→	**WholeNumber** | **RealNumber.**
WholeNumber	→	**digit**{**digit**}
	|	**octalDigit**{**octalDigit**} `B` | `C`
	|	**digit**{**hexDigit**} `H`.
octalDigit	→	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`.
digit	→	**octalDigit** | `8` | `9`.
hexDigit	→	**digit** | `A` | `B` | `C` | `D` | `E` | `F`.

literal whole number



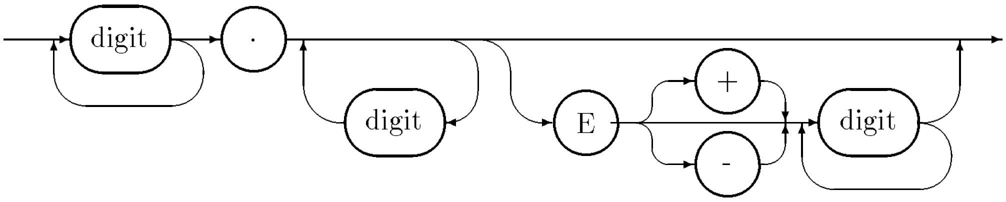
CHAR-valued number



A real number always contains a decimal point. Optionally it may contain a decimal scale factor. The letter E is pronounced as “ten to the power of”.

realNumber → **digit**{**digit**} ‘.’ {**digit**} [**scaleFactor**].
scaleFactor → `E` [+ | −] **digit**{**digit**}.

REAL-valued number

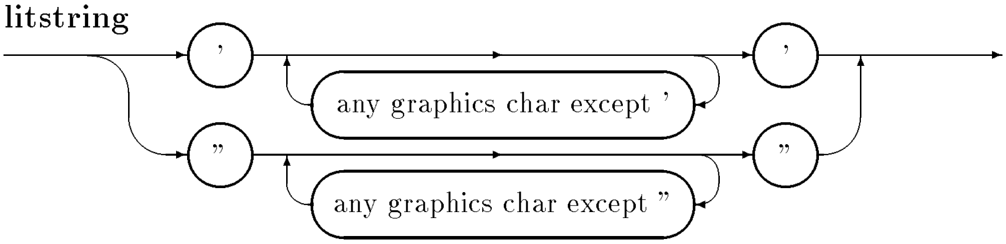


Strings are sequences of characters enclosed in quotes. Both double and single quotes (apostrophes) may be used. However, the opening and closing quotes must be the same character, and this character cannot occur within the string. A string must not extend over the end of a line.

litString → `'`{**character**}`'` | `"`{**character**}`"`.

Examples: `"Life's great"` `'Watch out!'`

litstring



The keywords and special symbols of Modula are listed below.

`AND`	`ARRAY`	`BEGIN`	`BY`	`CASE`
`CONST`	`DEFINITION`	`DIV`	`DO`	`ELSE`
`ELSIF`	`END`	`EXIT`	`EXPORT`	`FOR`
`FORWARD`	`FROM`	`IF`	`IMPLEMENTATION`	`IMPORT`
`IN`	`LOOP`	`MOD`	`MODULE`	`NOT`
`OF`	`OR`	`POINTER`	`PROCEDURE`	`QUALIFIED`
`RECORD`	`REPEAT`	`RETURN`	`SET`	`THEN`
`TO`	`TYPE`	`UNTIL`	`VAR`	`WHILE`
`WITH`				

`+` `-` `*` `/` `:=` `&` `.` `..`

```
;    (    )    [    ]    {    }    ,
^    ~    =    #    <    >    <>   <=
>=   "    '    |    :
```

The two symbols `<>` and # are lexical synonyms, with both denoting the inequality predicate (*not equal to*), the two symbols `~` and `NOT` are lexical synonyms, both denoting logical negation (*Boolean NOT*), finally, the two symbols & and `AND` are lexical synonyms, both denoting logical conjunction (*Boolean AND*).

Comments may be inserted between any two symbols in the program. Comments are enclosed in '(*' and '*)' markers. Material in comments does not affect the meaning of the program.

Comments may be properly nested, so that it is possible to comment out sections of code that include comments. Beware however of commenting out sections of code that contain literal strings that contain character sequences that will be interpreted as comment delimiters.

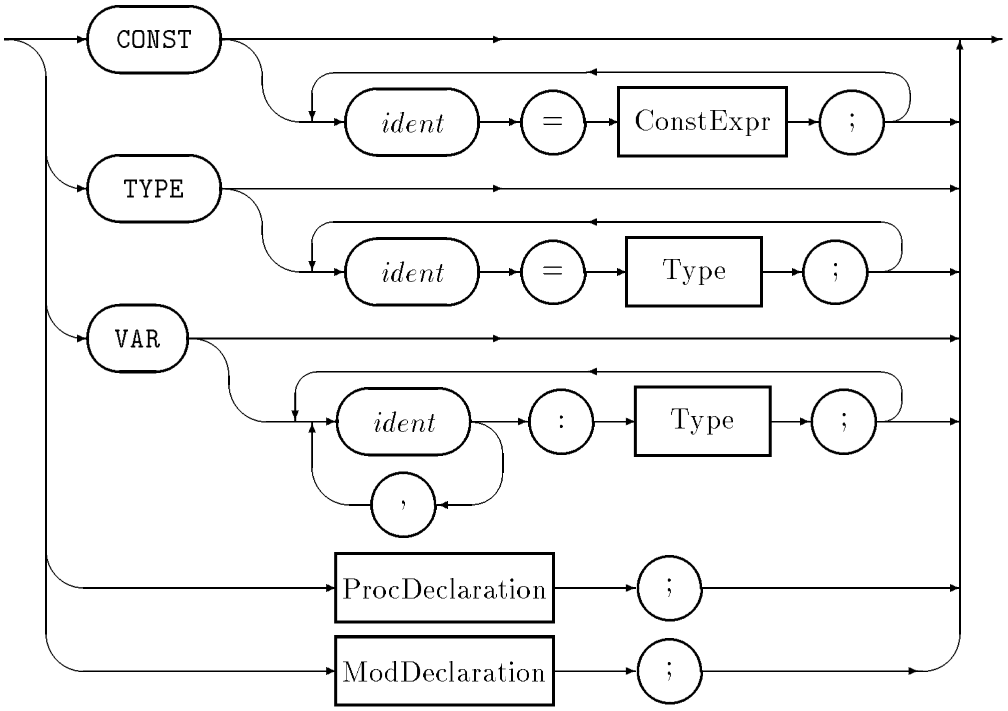
Chapter 3

Declarations and scope rules

Syntax

Declaration	$\rightarrow$	`CONST` {*ident* = **ConstExpr** ;}
	|	`TYPE` {*ident*= **Type** ;} | `VAR` {**IdentList** : **Type** ;}
	|	**ModDeclaration** ; | **ProcDeclaration** ;.
IdentList	$\rightarrow$	*ident* { , *ident*}.

Declaration



Every identifier occurring in a program must be introduced by a declaration, unless it is a standard identifier (that is, a *pervasive identifier*). Declarations also serve to specify certain permanent properties of an object, such as whether it is a constant, a type, a variable, a procedure, or a module.

The identifier is then used to refer to the associated object. This is possible in those parts of the program that are within the *scope* of the declaration. In general, the scope extends over the entire block (procedure or module declaration) to which the declaration belongs and to which the object is local.

The order of declaration of objects is not constrained as it is in language *Pascal*. This additional freedom may be used to place declarations together for objects that logically belong together. Thus the declarations of constants that are parameters in type declarations may be placed with the type declarations.

3.1 Declaration before use

In general, Modula demands that objects *used* in declarations should themselves have been declared (or imported) earlier in the program text. This rule is relaxed only in the declaration of pointer and procedure types, as detailed in a later chapter.

An object is not declared until its declaration is complete. This means, for example, that a type cannot refer to itself within its own declaration. An occurrence of an identifier is a *used occurrence in a declaration* if the identifier appears in the right-hand-part of the declaration of another object, is used as a type-name in a formal parameter list, or appears in an import list. The occurrence of an identifier in an *export* list does not constitute a used occurrence in a declaration.

The use of a procedure name in the body part of a procedure or module declaration does not constitute a used occurrence in a declaration, within the meaning of the declaration before use rule. Or at least **gpm** doesn't think so. The ability to call procedures that have not yet been declared is a traditional part of Modula. However, some simple compilers are unable to cope with this. Such compilers are said to have *single pass restrictions*. In order to legitimize such compilers, Modula now allows procedures to be explicitly declared `FORWARD`. Details are given in the chapter on procedure declarations. **gpm** allows such forward declarations, and checks their correctness, but does not need them for procedures. A special case of use is given in the chapter on module declarations.

3.2 Scope rules

In general, the scope of an object is the whole of the procedure or module in which it is declared. It is important to note some consequences of this rule. The *point of declaration* of an object may be somewhere in the middle of a module (say). The scope of the declaration is the whole of the module, but the name of the object may not be used in that part of the scope that precedes the point of declaration.

This rule prevents any other object from being associated with the identifier of a local object in that part of the procedure or module that precedes the declaration of the object. Consider the following situation —

```
TYPE Thing = POINTER TO CHAR;

PROCEDURE Foo;
  VAR  bar   : Thing; (* an error *)
  TYPE Thing = POINTER TO INTEGER;
```

```
BEGIN
  ...
```

The marked variable declaration might be either: an illegal reference to the relatively global type *Thing*, or is an illegal used occurrence of the identifier of the local type *Thing*. It is known that some early Modula compilers will not detect this error, but **gpm** will reject it firmly.

The exceptions to the *scope extends throughout the procedure or module* rule are as follows —

- the scope of an exported object expands to the whole of the block into which it is injected. In the case of compilation units the scope expands to include all of the modules that import that object
- if a procedure declaration is in the scope of some object, and the procedure contains a declaration of a local object with the same name as that object, inside the nested scope the identifier designates the local object. The more global object is rendered invisible by the more local declaration. It is said to be *occluded* or to be *incidentally invisible*. Such an occluded object becomes visible again at the end of the scope of the more deeply nested object

3.3 Pervasive identifiers

The following identifiers are said to be **pervasive**. They are visible in all scopes in which they are not made incidentally invisible by a local declaration. The purpose of each of these is explained in the relevant chapter, but they are collected here for reference.

Built-in constants

FALSE	NIL	TRUE

Built-in types

BITSET	BOOLEAN	CARDINAL	CHAR	INTEGER
LONGREAL	PROC	REAL	SHORTREAL[1]	

Built-in procedures

ABORT[1]	ABS	CAP	CHR	DEC
DISPOSE	EXCL	FLOAT	HALT	HIGH
INC	INCL	LENGTH	LFLOAT	MAX
MIN	NEW	ODD	ORD	SIZE
SFLOAT[1]	TRUNC	VAL		

[1]Not standard modula, **gpm** only

Chapter 4

Constant declarations

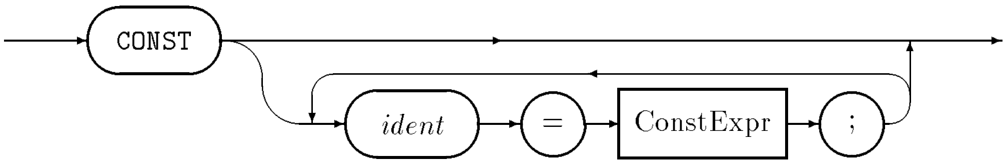
Syntax

Declaration → ... | CONST {*ident* = **ConstExpr** ; } | ...
Definition → ... | CONST {*ident* = **ConstExpr** ; } | ...
ConstExpr → **Expression**.

Constant declarations are used to introduce **manifest constants**, that is identifiers which denote particular constant values. Modula provides for the declarations of constants of all types for which expressions may be constructed.

Constant declarations may occur anywhere in the declaration part of blocks, and are used both in other declarations, and in the statement sequence part of blocks. Constant *definitions* share the same syntax, and occur in the definition parts of separately compiled modules. These are treated in the *Compilation Units* chapter of this manual.

Constant declarations



4.1 Literal constants

The basic literal constants were introduced in the chapter on syntax. These are *WholeNumbers*, *RealNumbers*, *litStrings*, and the various pervasive constants `TRUE`, `FALSE` and `NIL`. These are used as the building blocks from which other constant expressions are constructed. Other constant values are introduced by the declaration of enumeration types, and other constant declarations.

Once a constant has been declared, it may be used in other constant, variable or type declarations. Consider the following parameterized declarations —

```
CONST ElementNumber = 3743;
      MaxIndex      = ElementNumber - 1;
      StrTabMax     = ElementNumber * 10;
```

```
VAR   HashTable   : ARRAY [0 .. MaxIndex] OF Entry;
      StringTable : ARRAY [0 .. StrTabMax] OF CHAR;
```

Some literal constants are of special types, which obey less strict rules of type compatibility than do other objects. It is not possible to declare variables to have these types, but literal constants automatically are of the appropriate type.

Literal whole-valued numbers are of a special type *ZZ*. Constants of this type are compatible with both signed and unsigned numeric types.

Literal strings are of the special **abstract string type** *SS*. Constants of this type are assignment compatible with all sufficiently long arrays of `CHAR`. The special cases of literal strings of length zero and one are also compatible with the type `CHAR`. Length-zero strings, when treated as character constants, have a value which is equal to the implementation defined **string termination character**. In **gpm** this value may be also denoted as *Ascii.nul* (if module *Ascii* has been imported) or as 0C. However, the denotation `''` or `""` is simpler and preferred in some styles.

For number literals ending with character `C`, the type of the constant is referred to here as *S1* (length-1 string). This is a special case of string literal which is *expression compatible* with the type `CHAR`, but which is also *assignment compatible* with arrays of `CHAR` in the same way as other strings.

For all other whole number literals, the type of the literal is the special integer type *ZZ*, mentioned earlier.

Real valued literals are of the special *RR* type which is expression compatible with all named real types. The following assignments are valid because of this property, and not because of any kind of automatic conversion of the type that some other languages provide.

```
CONST pi = 3.14159265358979323855;
VAR   short : SHORTREAL; (* IEEE single *)
      long  : LONGREAL;  (* IEEE double *)
...
short := pi;
long  := pi;
```

Enumeration constants have type corresponding to the enumeration of which they denote a particular value.

The constant expressions which occur in constant declarations may use all operators which are applicable for the constant operand type. It is also permissible to use certain of the pervasive functions in such declarations. For example, it is legal (and perhaps even useful) to use constructs of the form —

```
CONST message = "dispute the dominant paradigm";
      mHigh   = LENGTH(message) - 1;
VAR   buffer  : ARRAY [0 .. mHigh] OF CHAR;
```

4.2 Procedure constants

Procedure constants provide a newly introduced method of declaring procedure synonyms. The name of a procedure, without parameters, denotes a procedure value of some particular procedure type. The use of such a name as the right hand side of a constant declaration introduces a procedure valued constant. The procedure constant is simply a manifest constant denoting the declared value.

This mechanism may be used to declare a synonym for a procedure. For example in the following program fragment, short synonyms are declared for two procedures.

```
IMPORT Terminal, StdError;

CONST  WrtT = Terminal.WriteString;
       WrtE = StdError.WriteString;
```

The mechanism is also useful in definition modules, for declaring specializations of existing modules, and thus sharing code —

```
DEFINITION MODULE Stacks;
  IMPORT Sequences;

  TYPE Stack = Sequences.Sequence;

  CONST MakeEmpty = Sequences.MakeEmpty(* *);
        IsEmpty   = Sequences.IsEmpty (* s : Stack *)
                                 (* returns BOOLEAN *);
        Push      = Sequences.LinkRight (*   VAR s : Stack;
                                            elem : Element *);
        Pop       = Sequences.UnLinkRight (* VAR s : Stack;
                                        VAR elem : Element *);
```

Note the use of comments in this example to remind the human reader of the actual parameter list and return types of the procedure constants.

4.3 Constant constructors

The syntax of constructors is given in detail in the chapter on expressions. In this chapter examples are given of constructors of various types.

Set constructors

Here is a simple example of constant set constructors, and an expression involving constant sets.

```
TYPE  CharSet = SET OF CHAR;
CONST alphas  = CharSet{'A' .. 'Z','a' .. 'z'};
      vowels  = CharSet{'a','e','i','o','u',
                        'A','E','I','O','U'};
      consonants = alphas - vowels;
```

Record constructors

Here are two examples of use of constant record constructors.

```
TYPE  Tree = POINTER TO Glue;
      Glue = RECORD
               key : KeyType;
               left, right : Tree;
             END;
CONST mtTree = Tree{"empty",NIL,NIL};
```

In this case the initialization of a tree node can be performed by the assignment of *mtTree*.

```
  t := mtTree;
```

The same effect might be obtained by the procedure call —

```
PROCEDURE MakeEmptyTree(VAR t : Tree);
BEGIN
  t := Tree{"empty",NIL,NIL};
END MakeEmptyTree;
```

Array constructors

Here are two different constant array constructors.

```
TYPE  ShortStr = ARRAY [0 .. 15] OF CHAR;
CONST str1 = ShortStr{'e','x','a','m','p','l','e',0C BY 9};
      str2 = "example";
```

In this example the two constants have different types and different possible operations, although they would look identical if printed. The first, *str1*, is of type *ShortStr*. This constant may have its components selected by indexing, but is only compatible with other object of this same array type.

The second constant is of the special abstract string type *SS*. This constant may not have its components selected by indexing, but may be assigned to character array variables of any sufficiently large index range.

```
TYPE  BytePosition = [0 .. 3];
      PositionMap  = ARRAY BytePosition OF BytePosition;
CONST bigEndian    = PositionMap {3,2,1,0};
      littleEndian = PositionMap {0,1,2,3};
      map = littleEndian; (* edit this if necessary *)
...
FOR index := 0 TO 3 DO
  word[index] := value[map[index]];
  ...
```

In this second example, a constant mapping array is defined, so that a single edit operation may change the mapping function while leaving the text of the code unaltered. In this case, since the map is a constant some compilers will *fold* many mapping operations at compile time, so that the mapping function will not consume runtime resources.

4.4 Function applications in constant expressions

Normally the use of a function application in an expression signals that the expression is not constant. Usually it is necessary to *execute* the code of a function procedure to evaluate the result of the function application. However, for some built-in procedures it is possible for the compiler to evaluate the function at compile time, provided the actual parameters are constant expressions.

The following *pseudo-functions* take type names as parameter, and may be used freely in constant declarations —

function	result type	meaning
`MAX`(*typeName*)	*type*, or *ZZ*	maximum value of type
`MIN`(*typeName*)	*type*, or *ZZ*	minimum value of type
`SIZE`(*typeName*)	`CARDINAL`	storage size of type

The first two of these may only be applied to the *totally ordered types*, such as the numeric, enumeration and subrange types. The last of these may also be called using an expression as a parameter. In that case the function returns the storage size of the expression type, but the expression must itself be constant.

The following functions may be applied in constant expressions, but require that the parameters are compile time constants.

function	result type	meaning
`ABS`(*ordExp*)	ExprType	absolute value of *ordExp*
`CAP`(*charExp*)	`CHAR`	capitalized value of *charExp*
`CHR`(*ordExp*)	`CHAR`	char of same ordinal value as *ordExp*
`FLOAT`(*numExp*)	`REAL`	real of same value as *numExp*
`INT`[1](*numExp*)	`INTEGER`	integer of same value
`LENGTH`(*strExp*)	`CARDINAL`	length of string constant
`LFLOAT`(*numExp*)	`REAL`	real of same value as *numExp*
`ODD`(*ordExp*)	`BOOLEAN`	predicate *is-odd?*
`ORD`(*ordExp*)	`CARDINAL`	ordinal value of *ordExp*
`SFLOAT`(*numExp*)	`SHORTREAL`	short of same value as *numExp*
`TRUNC`(*realExp*)	`CARDINAL`	largest cardinal `<=` *realExp*
`VAL`(*typeName,ordExp*)	*type*	value of type with same ordinal value

[1] `INT` is not implemented in **gpm** at this time

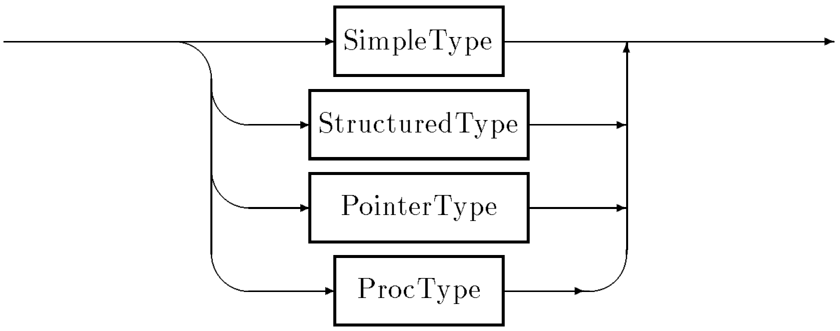
Chapter 5

Type declarations

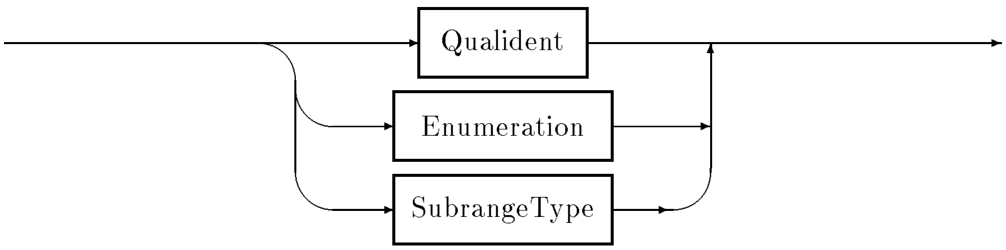
Syntax

Declaration	$\rightarrow$	... | `TYPE` = **Type** ; | ...
Definition	$\rightarrow$	... | `TYPE` [= **Type**] ; | ...
Type	$\rightarrow$	**SimpleType** | **StructuredType** | **PointerType** | **ProcType**.
SimpleType	$\rightarrow$	**Qualident** | **Enumeration** | **SubrangeType**.
StructuredType	$\rightarrow$	**SetType** | **ArrayType** | **RecordType**.

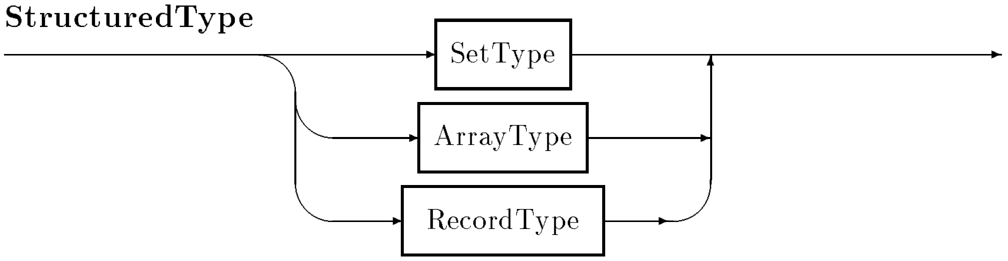
Type



SimpleType



StructuredType



The declaration of a type introduces a set of values that a variable of the type may assume. Every type is characterized by its *structuring method* and its *set of values*.

5.1 The built-in types

All user-defined types are constructed from certain basic types that are built-in to the language. These are `BITSET, BOOLEAN, CARDINAL, CHAR, INTEGER, LONGREAL, REAL`, and `SHORTREAL`[1]. The properties of each of these is as follows –

Type Name	Type Value-Set
`BITSET`	subsets of an implementation defined set in **gpm** `SET OF [0 .. 31]`
`BOOLEAN`	may be thought of as the enumeration `FALSE, TRUE`
`CARDINAL`	the set of integers from 0 to `MAX(CARDINAL)` in **gpm** 0 to $2^{32}-1$, [0 .. 4 294 967 295]
`CHAR`	an implementation defined set of characters in **gpm** the ascii set 0C to 377C
`INTEGER`	some set of integers in **gpm** -2^{31} to $2^{31}-1$, [-2 147 483 648 .. 2 147 483 647]
`LONGREAL`	some implementation defined set of model numbers in **gpm** the IEEE double precision type
`PROC`	parameterless proper procedures
`REAL`	some implementation defined set of model numbers in **gpm** the IEEE double precision type
`SHORTREAL`	some implementation defined set of model numbers in **gpm** the IEEE single precision type

The types `BOOLEAN, CARDINAL, CHAR, INTEGER`, are **ordinal types**, that is, they have a total order defined on their values, and may be placed in one-to-one correspondence with some subrange of the integers.

The built-in function `ORD` extracts the ordinal number associated with a value of any ordinal type. For example, for the Boolean type `ORD(FALSE)` = 0 and `ORD(TRUE)` = 1. The built-in procedures `INC, DEC` also apply to all ordinal types.

[1]Because the types `REAL` and `LONGREAL` are identical in **gpm** the non-standard type `SHORTREAL` is included as a language extension

The types `CARDINAL, INTEGER, LONGREAL, REAL, SHORTREAL` are **number types**. They are not only ordered, but permit arithmetic operations to be performed on expressions of these types.

5.2 Type synonyms

A type may be declared as a synonym for an already declared type. Such a declaration does not introduce a new type, but just another name for the existing type.

As a particular instance of this, inside an implementation module a type declared opaquely in the definition may be elaborated as a synonym of another type.

```
(* type MyOpaque is an opaque type *)
TYPE MyOpaque = SYSTEM.ADDRESS;
(* importing modules need not know this *)
```

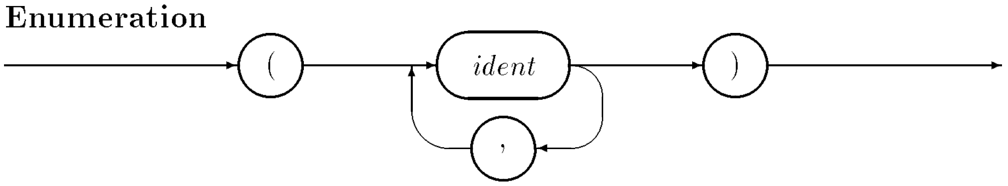
In this particular example the opaque has been elaborated as the special type `ADDRESS`, which has special type compatability rules. The *importing module* will treat this type strictly, but within the implementation the special properties may be used freely.

5.3 Enumeration types

Syntax

Enumeration → "("**IdentList**")".

Enumeration



Enumerations introduce ordinal types with named values. The values have ordinal numbers starting from 0. Thus in the example —

```
TYPE DaysOfWeek = (sunday, monday, tuesday, wednesday,
                   thursday, friday, saturday);
```

the expression `ORD(friday)` evaluates to 5, and `MAX(DaysOfWeek)` evaluates to *saturday*.

The identifiers in the type declaration are called **enumeration constants**. **gpm** limits the number of enumeration constants in any type to 256, as an implementation limit.

Enumerations have special properties as far as import and export are concerned. When an enumeration type is imported or exported, the names of the enumeration constants are imported or exported in the same mode (qualified or transparent) as is the type name. This is a matter of convenience, but sometimes causes subtle errors if two enumeration types are imported, but share an enumeration type name. In this case the compiler must give information as to which identifier is the cause of the error.

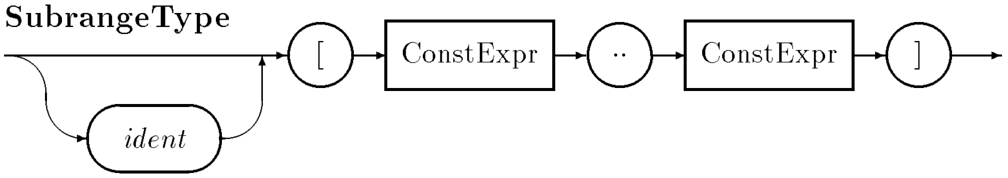
Because enumeration types are ordinal types all the usual equality, inequality and ordering tests apply to expressions of these types. However, arithmetic is not permitted on values of these types, except for the special case of the built-in procedures `INC, DEC` which respectively increment and decrement variables of any ordinal type.

5.4 Subrange types

Syntax

SubrangeType $\rightarrow$ [*ident*] "[" **ConstExpr** .. **ConstExpr** "]".

SubrangeType



Subrange types are ordinal types derived from some other ordinal type by restriction of the range of values. The declaration contains two constant expressions that give the lower and upper bounds to the range of values for the new type. The types of the two bounding expressions must be (expression) compatible. The next section defines this property. It is an error if the value of the first bounding expression is greater than the value of the second bounding expression. It follows that every subrange has a least a single value.

In the example —

```
TYPE  SubType = [min .. max];
```

the types of *min* and *max* must be compatible and *min* must be less than or equal to *max*.

The type from which the new type is derived is called the **host type**. Usually the host type is inferred from the types of the bounding expressions.

It is possible to control the host type by including a type name as the optional *ident* in the declaration. This type name is the *candidate host type*. In such a case each of the bounding expressions must be compatible with the candidate host type. The actual host type is the candidate type or, if the candidate type is itself a subrange, the host type is the host type of the candidate type. The following examples illustrate various possibilities —

```
TYPE  SubType1 = [0 .. 7];         (* host type is CARDINAL *)
      SubType2 = INTEGER[0 .. 7];  (* host type is INTEGER  *)
      SubType3 = SubType1[0 .. 4]; (* host type is CARDINAL *)
      SubType4 = SubType2[0 .. 7]; (* host type is INTEGER  *)
      WeekDays = [mon .. fri];   (* host type is DaysOfWeek *)
```

In the first example the type of the two bounding expressions is the internal type *ZZ*, which is compatible with either `CARDINAL` or `INTEGER`. However, since the two values lie entirely in the `CARDINAL` range, this will be the default host type.

The purpose of forcing the host type to be `INTEGER`, as in the second example, might be to ensure that variables of the subrange type are compatible with other signed types.

The thoughtful use of subranges is a mark of a skilled and careful programmer. By using subranges, the runtime error checks in the program can be made more effective. In many cases the use of subranges actually leads to faster program execution, since some runtime checks may be recognized as redundant during code optimization.

In **gpm** subranges are allocated only enough space in computer memory to store the range of values that are represented.

5.5 Type compatability

Two types are said to be **expression compatible** or simply **compatible** if any of the following conditions are met —

- the types are the same type, or
- one type is a subrange of the other, or they are both subranges of the same host type, or
- one type is *ZZ* and the other is *ZZ*, `INTEGER`, `CARDINAL`, or a subrange of one of these.

In the following example, the types *SubType1* and *SubType2* are not compatible, since they have different host types. However, types *SubType1* and *SubType3* are compatible, since the two types have the same host type.

```
TYPE  SubType1 = [0 .. 7];          (* host type is CARDINAL *)
      SubType2 = INTEGER[0 .. 7];   (* host type is INTEGER  *)
      SubType3 = SubType1[0 .. 4];  (* host type is CARDINAL *)
```

It is important to remember that in general an anonymous type is different to *every* other named or unnamed type. This hardly has any effect on anonymous subranges, since they are automatically compatible with all other subranges of the same host type. However, for the structured types, anonymous types are incompatible with all other types.

5.6 Pointer Types

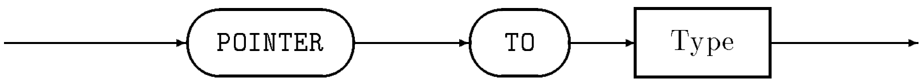
Syntax

PointerType $\rightarrow$ `POINTER` `TO` **Type**.

Pointers are used to designate dynamically allocated objects. Every pointer type is only permitted to designate objects of one particular type, the pointer is said to be **bound** to its **target type**.

A pointer type declaration simply declares the target type to which the pointer is to be bound.

PointerType



It is commonplace for the target types of pointer types to contain components of the pointer type. Such a structure is called a **recursive type**. Because of the possibility of recursive types, the declaration before use rule is relaxed in this case to allow pointer types to be declared before the target type to which they are bound. It is an error if the **forward target type** of a pointer type is not declared in the same block as the pointer declaration.

Here is a typical recursive type declaration involving a forward target type —

```
TYPE  TreePtr  = POINTER TO TreeGlue;
      TreeGlue = RECORD
                   key   : KeyType;
                   info  : InfoType;
                   left  : TreePtr;  (* left child  *)
                   right : TreePtr;  (* right child *)
                 END;
```

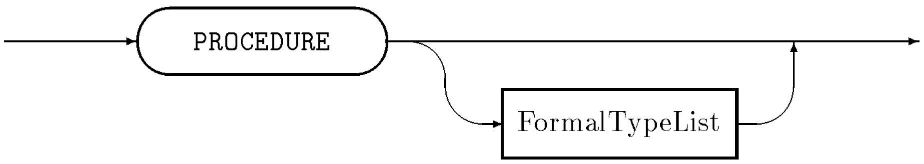
5.7 Procedure types

Syntax

ProcType	$\rightarrow$	`PROCEDURE` [**FormalTypeList**] .
FormalTypeList	$\rightarrow$	"("[**FTSection** { , **FTSection** }] ")" [: **Qualident**] .
FTSection	$\rightarrow$	[`VAR`] **FormalType**.

Modula provides for the declaration of procedure types, and the declaration of variables of those types. A procedure type consists of the keyword `PROCEDURE` followed by an optional formal parameter list. The declared or imported procedures with conforming parameter lists and result types are the possible values that an object of procedure type may have. In principle there are an unbounded number of possible values of every procedure type.

ProcType



The formal parameter list is a list of the formal types of the parameters and, in the case of a function procedure type, a return type as well. Each formal type consists of a possibly qualified type name, together with an indication whether the formal parameter is of value or variable mode, and whether is an open array or not. The formal type does not give a dummy name to the formal parameters that every procedure of the defined type will have.

Examples

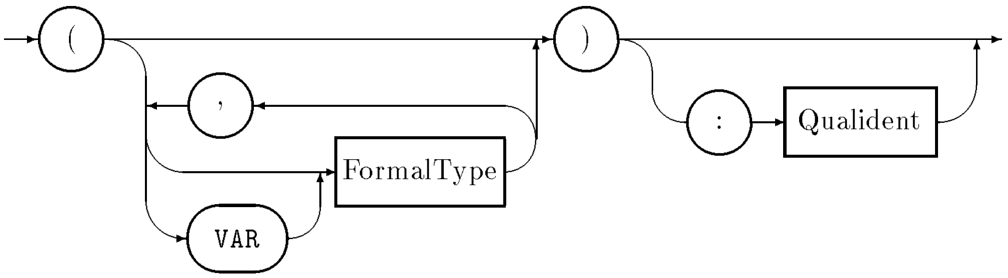
```
TYPE StringWriteProc = PROCEDURE(ARRAY OF CHAR);
     CharReadProc    = PROCEDURE(VAR CHAR);
     StringPredicate =
       PROCEDURE(ARRAY OF CHAR,ARRAY OF CHAR) : BOOLEAN;
```

In these examples *StringWriteProc* is a proper procedure type which has a value open array of characters as its formal parameter. The familiar procedures *Terminal.WriteString* and *InOut.WriteString* are values of this type.

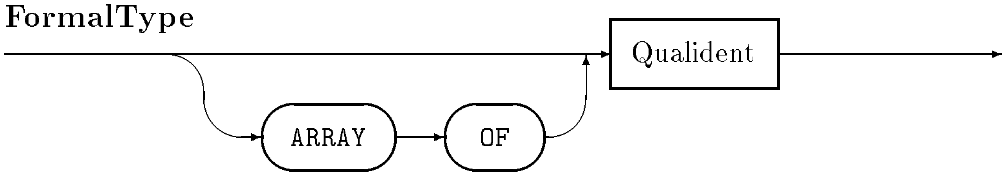
CharReadProc is a proper procedure type which has a variable character as its formal parameter. The procedures *Terminal.Read* and *InOut.Read* are values of this type.

The example *StringPredicate* is a function procedure type which takes two value open array of character formals, and returns a Boolean value. A string equality-comparison procedure would be a value of this type.

FormalTypeList



FormalType



The formal type lists for procedure types may refer to types that have not yet been declared. As is the case with pointer declarations, that share this property, the type must be declared within the same block. The possibility of using forward declarations in procedure type declarations allows recursive procedure types to be created. For example, here is a procedure type that contains a variable parameter of its own type.

```
TYPE ScannerProc = PROCEDURE(SymbolType, VAR ScannerProc);
```

Such a procedure type might be used to construct lexical scanners that are used in the following manner —

```
VAR current : ScannerProc;
    symbol  : SymbolType;

BEGIN
  current := InitialProc;
  GetSymbol(symbol);
  WHILE symbol <> endSymbol DO
    current(symbol,current);
    GetSymbol(symbol);
  END;
```

In this example, each call of *current* performs one step of the scanning process and replaces the current procedure value with another corresponding to the new state of the scanner.

Type compatibility for procedure value assignments

The comparison of the types of objects that are of procedure types is different to all other objects in Modula. Elsewhere in Modula *name equivalence* is the rule. That is, two objects have the same type if they have the same type name (or the names are synonyms) or if they have the same anonymous type as a result of having been declared together. For procedures the rule is **structural equivalence**. That is —

> Two procedure objects are compatible if they have the same number, type and mode of formal parameters, and have the same return type.

This rule is necessary since procedure *values*, that is, actual procedures, do not have a declared type. They only have a type structure that may be inferred from the formal parameters of the procedure heading.

As an example, if a procedure type *VoidProc* was declared by the following —

```
TYPE VoidProc = PROCEDURE; (* no formals *)
```

then any parameterless proper procedure is compatible with variables and parameters of this type.

5.8 Set types

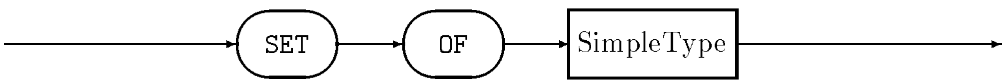
Syntax

SetType → `SET OF` **SimpleType**.

Set types are structured types which take values that are subsets of some ordinal **base type**. Although the syntax allows any simple type as base type, the semantic rules only allow ordinal types.

Modula implementations usually enforce some sort of limit on the cardinality of the base type that is permitted. In the case of **gpm** the allowed base types must have ordinal numbers lying in the subrange [0 .. 256][2]. Thus sets of `CHAR` and any enumeration type are allowed, but sets of the whole number types are quite restricted.

SetType



Examples:

```
TYPE CharSet      = SET OF CHAR;
     PrintCharSet = SET OF [" " .. "~"];
```

[2]Later releases are expected to relax this constraint, allowing arbitrary ordinal subranges of some maximum cardinality

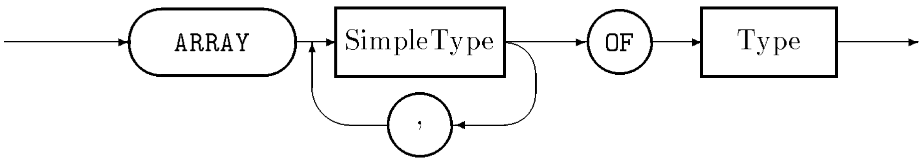
5.9 Array types

Syntax

ArrayType → `ARRAY` **SimpleType** {, **SimpleType**} `OF` **Type**.

Array type declarations introduce positional aggregates, that is, structures formed by aggregating a number of components, and which allow selection of components based on position (indexing). In this case, all of the components must be of the same type — the **element type**, while the type of the selector expression is the **index type**. Index types must necessarily be ordinal types, but element types may be any type.

ArrayType



In particular, it is possible to have an array in which the element type is another array. This construction occurs so often that the syntax allows an abbreviated form for specifying such types. The use of a comma separated list of index types, as in —

```
TYPE Matrix = ARRAY Index1, Index2 OF REAL;
```

is a short notation that is identical in meaning to the full form

```
TYPE Matrix = ARRAY Index1 OF
                ARRAY Index2 OF REAL;
```

The same effect might have been obtained by having a named inner type as in this final example —

```
TYPE Vector = ARRAY Index2 OF REAL;
     Matrix = ARRAY Index1 OF Vector;
```

Note carefully that if square brackets occur in the definition of a type, they appear only as part of the declaration of an anonymous index type. This is different to language *Pascal*, and a potential cause of beginning errors. The following example contrasts correct and incorrect declarations —

```
TYPE TwoDims = ARRAY [0 .. 2], [0 .. 2] OF REAL; (* correct *)
TYPE TwoDims = ARRAY [0 .. 2, 0 .. 2] OF REAL; (* incorrect *)
```

The same abbreviation is allowed in the selection of components as in the declarations. Thus if the (0,3) element of a variable *foo* of *Matrix* type is required it may be selected equivalently by either of the designators

```
foo[0][3]        foo[0,3]
```

Some compilers may place a limit on the depth of nesting for such arrays. In **gpm** this limit is 16.

Anonymous element types

When a type declaration declares a multidimensional array, the type is actually a one-dimensional array of some element type that is another array type. If the multi-dimensional array type is declared directly, the element type is an **anonymous type**, that is, the type does not have a type identifier associated with it.

This style has no particular disadvantage if objects of the type will always be manipulated as entire variables. However, if the element type needs to be, for example, used as the formal parameter type for a procedure, then the type must have a name.

In the following example, unless the type *Vector* is explictly declared, it could not be declared as the formal parameter type of the procedure *VecLength*.

```
TYPE  Vector = ARRAY [0 .. 2] OF REAL;
      Matrix = ARRAY [0 .. 2] OF Vector;
VAR   matrix1 : Matrix;
...
x := VecLength(matrix1[1]);
...
```

5.10 Record types

Syntax

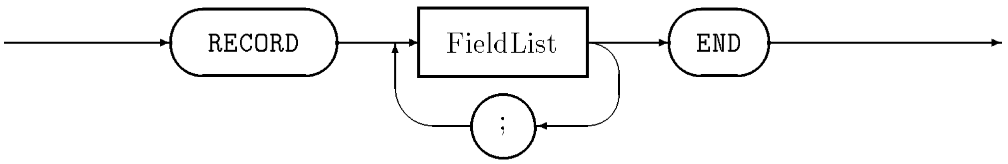
RecordType	→	`RECORD` **FieldListSequence** `END`.
FieldListSequence	→	**FieldList** {; **FieldList**} .
FieldList	→	[**IdentList** : **Type**]
	|	`CASE` [*ident*] : **Qualident** `OF`
		Variant {"|" **Variant**}
		[`ELSE` **FieldListSequence**] `END`.
Variant	→	[**CaseLabelList** : **FieldListSequence**] .

Record type declarations introduce named aggregates, that is, structures formed by aggregating a number of components, and which allow selection of components by name. The components may be of any type, and there may be any number of them. The values that record types may have are thus **product types**, since they are the (mathematical) direct product of the component type values.

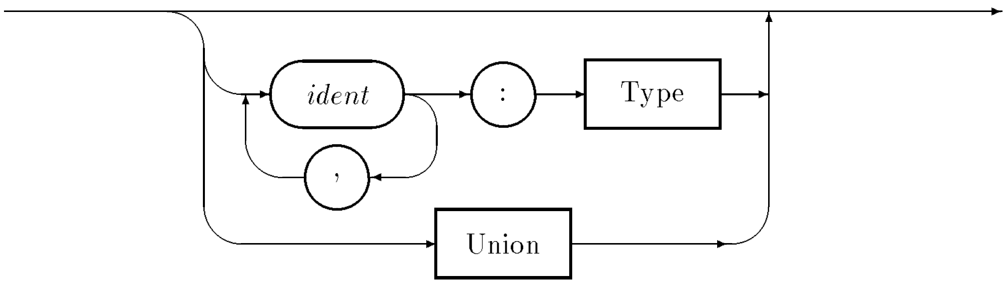
Modula also allows a variation of the same syntax to specify **union types**, that is types that allow one of several forms. In Modula, types containing unions are called **variant records**.

Here is the syntax diagram for the overall structure of the record types. The field lists in the declarations may be specified in either of two ways. The fields that form part of the product part of the type are specified using the *ident* : **Type** form, while unions are specified by means of **Union** phrases.

RecordType



FieldList



All the identifiers used as field names for a particular record type must be distinct, but need not be distinct from other names used in the same scope. This is because field names are used qualified by the name of the object to which they refer, thus providing additional context.

Note that the semicolon is a separator between field lists, but that a *FieldList* may be empty, allowing the use of redundant semicolons. In particular, there is a empty field list in every record that has a semicolon immediately before an `END`. In all the examples in this manual the redundant semicolon is deliberately left in, as a matter of style.

Here are some examples of ordinary record types declared using the first form of field list —

```
TYPE PersonInfo = RECORD
                    name  : StringIndex;
                    class : ClassType;
                    prNum : PayRollNumber;
                  END;
```

In this case, the fields of the record would be selected using the variable name qualified by the field identifier.

```
IF person.name = chosenName THEN
  rank := person.class;
  num  := person.prNum;
  ...
```

Anonymous component types

When a type declaration declares a record where one of the components contains a nested declaration of some structured type, the nested type is an **anonymous type**, that is, the type does not have a type identifier associated with it.

This style has no particular disadvantage if objects of the type will always be manipulated as entire variables. However, if the component type needs to be, for example, used as the formal parameter type for a procedure, then the type must have a name.

In the following example, if the type *NameType* was anonymous, it could not be declared as the formal parameter type of the procedure *PrintName*.

```
TYPE  NameType = RECORD
                   length : CARDINAL;
                   chars  : CharPtr;
                 END;

      PersonId = RECORD
                   serial : CARDINAL;
                   name   : NameType;
                   ...
...
PrintName(thisPerson.name);
...
```

Note that if *NameType* was actually an anonymous type, the procedure *PrintName* could still have been declared with multiple parameters, and the components *thisPerson.name.length* and *thisPerson.name.chars* passed to it individually. Alternatively, if *NameType* were an anonymous *array* type, and if *PrintName* had an open array formal parameter of appropriate type, then the *name* component could still have been passed to the procedure.

Types for dynamic data structures

Records are widely used for creating and manipulating *dynamic data structures*, that is, pointer accessed structures. In this case it is common to have declarations in the following form —

```
TYPE  ElemPtr   = POINTER TO GlueBlock;
      Sequence  = RECORD
                    first, last : ElemPtr;
                  END;
      GlueBlock = RECORD
                    next : ElemPtr;
                    info : PersonInfo;
                  END;
```

Structures using these types may be accessed by code of which the following is typical —

```
PROCEDURE InitSequ(VAR seq : Sequence);
BEGIN
  seq.first := NIL; seq.last := NIL;
END InitSequ;

PROCEDURE LinkLeft(VAR seq : Sequence;
```

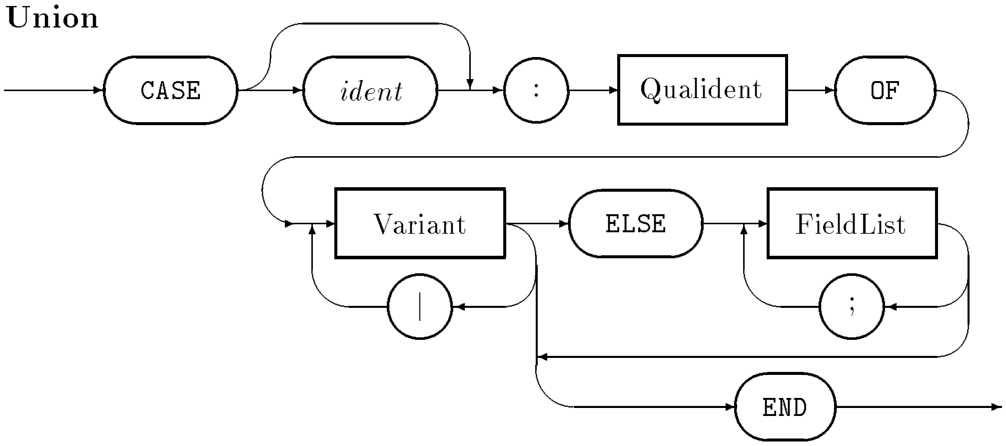
```
                           nxt : PersonInfo);
    VAR ptr : ElemPtr;
  BEGIN
    NEW(ptr);                 (* get new glue *)
    ptr^.next := seq.first;   (* hook up glue *)
    seq.first := ptr;
    ptr^.info := nxt;         (* load up info *)
    IF seq.last = NIL THEN seq.last := ptr END;
  END LinkLeft;
```

Variant records

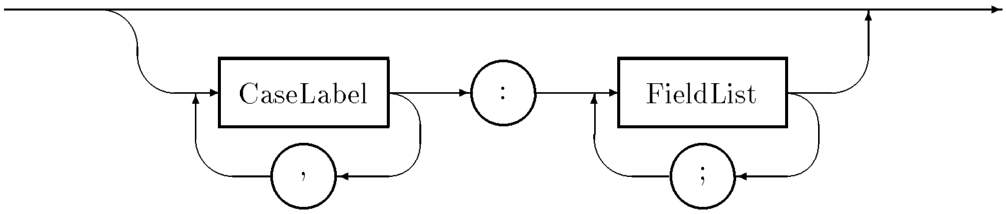
Variant records are used when an information structure has one of several forms depending on the value of some ordinal selector component. They are also necessary for certain kinds of dynamic data structures, the so called **heterogeneous linked structures** where a pointer must be bound to a union type, rather than to a simple product type.

Components of records that are union types are declared using syntax that is similar, in general terms, to that used for `CASE` statements. Notionally, every union embedded in a record has a **tag type** that selects the **active variant**. This tag type must be an ordinal type. A variant must be specified for every value of this type, either explicitly in a case selector list, or by means of the `ELSE` part of the declaration.

Union



Variant



It is permissible to omit the tag field in the declaration of a union part of record declaration. In this case the colon and the tag type name must still appear. It is good practice to place a comment noting the deliberate omission of the tag, since this is a potentially insecure mechanism that should not pass without notice.

Note that the vertical bar is a separator between variants, but that a *Variant* may be empty, allowing the use of redundant bar symbols. In particular, there is a empty variant in every union that has a vertical bar immediately after the OF. In all the examples in this manual the redundant bar is deliberately left in, as a matter of formatting style.

Here is a simple example of a variant record declaration.

```
TYPE VehicleType = (coupe, sedan, truck, omnibus);
     VehicleInfo =
        RECORD
          make   : NameString;
          model  : NameString;
          engine : EngineDetailType;
          CASE type : VehicleType OF
          | truck   : loadCapacity  : REAL;
          | omnibus : numberOfSeats : CARDINAL;
          ELSE (* nothing extra *)
          END;
        END;
```

In the case of heterogeneous linked data structures, it is necessary to allow a pointer to point to any one of several different block types. Since pointers are bound to unique target type, that type must be a variant record. Here is an example that demonstrates the principle in the case of a representation of expression syntax —

```
TYPE Expression = POINTER TO Block;
     BlockType  = (add,sub,mul,div,neg,val);
     Block = RECORD
               CASE op : BlockType OF
               | add, sub, mul, div :
                   leftEx, rightEx : Expression;
               | neg : operand : Expression;
               | val : number  : REAL;
               END;
             END;
```

Note that four different tag values share the same field types, reflecting the fact that all binary operators have a left and right operand. Here now is the code for evaluating such expressions.

```
PROCEDURE ValueOf(exp : Expression) : REAL;
BEGIN
  WITH exp^ DO
    CASE op OF
    | val : RETURN number;
    | neg : RETURN - ValueOf(operand);
    | add : RETURN ValueOf(leftEx) + ValueOf(rightEx);
    | sub : RETURN ValueOf(leftEx) - ValueOf(rightEx);
    | mul : RETURN ValueOf(leftEx) * ValueOf(rightEx);
```

```
      | div : RETURN ValueOf(leftEx) / ValueOf(rightEx);
      END; (* case *)
    END; (* with *)
  END ValueOf;
```

Note how the code of the procedure mimics the syntax of the type declaration. This is a typical phenomenon.

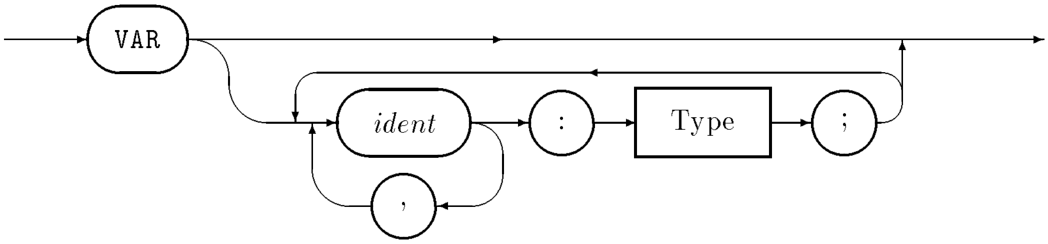
Chapter 6

Variable declarations

Syntax

Declaration → ... | `VAR` {**IdentList** : **Type** ; } | ...
Definition → ... | `VAR` {**IdentList** : **Type** ; } | ...

Variable declarations



Variable declarations introduce local objects into the current scope. The type of the variable is the type that appears in the declaration.

6.1 Extent of variables

Variables that are declared in scopes that are nested only within modules but not procedures, are said to be of **static extent**. This means that such a variable is allocated storage space at the time of the start of the program, and remains bound to that location throughout the entire running of the program.

Variables that are declared in scopes that are nested within one or more procedures are said to have **automatic extent**. This means that the variable is bound to some location at the time of invocation of the most closely enclosing procedure, and ceases to exist when the most closely enclosing procedure terminates. Thus repeated calls to the same procedure cannot rely on finding an initial value that is the same as the final value in the previous invocation.

Variables that are declared to be of pointer types, have bound variables that are of **dynamic extent**. This means that the bound variable does not exist until it is explictly allocated by `NEW` (or equivalently `ALLOCATE`). Such a variable continues to exist until it is explicitly deallocated by `DISPOSE` (or equivalently `DEALLOCATE`). Note that the bound variable continues to exist, even if all the pointers that were bound to the variable cease to exist. However such a variable is inaccessible — it is said to be **garbage**. Programs should, in general, try to avoid the creation of garbage.

6.2 Anonymous types

If a variable declaration has an explicit type rather than a type name on its right hand side, the variable is said to be of an **anonymous type**. In the case of variables of anonymous subrange types, very little harm is done, except to the aesthetics of the program. This lack of harm arises because subrange types are automatically compatible with their host type, and with all other subranges of the same base type, whether anonymous or not.

The practice of declaring variables of anonymous structured types is rather more dangerous. Variables of anonymous structured types are not compatible with any other type, and every anonymous type is distinct. In the following example some of the potential problems are indicated.

```
TYPE  Thing = ARRAY [0 .. 7] OF CHAR;
VAR   str1  : Thing;
      str2  : ARRAY [0 .. 7] OF CHAR;
      str3  : ARRAY [0 .. 7] OF CHAR;
...
str1 := str2;  (* illegal, types are incompatible *)
str2 := str3;  (* illegal, types are incompatible *)
```

If the variables of anonymous type are declared together, one of the assignments then becomes legal.

```
TYPE  Thing = ARRAY [0 .. 7] OF CHAR;
VAR   str1  : Thing;
      str2, str3  : ARRAY [0 .. 7] OF CHAR;
...
str1 := str2;  (* illegal, types are incompatible *)
str2 := str3;  (* legal, variables same anon type *)
```

Such an anonymous type cannot be the declared type of a formal parameter in a procedure, so that variables declared in this way cannot be passed to procedures as parameters (except maybe as open array parameters).

Note however, that in any case, an assignment of a string literal to any one of the string variables in the above examples would be legal. It would also be legal to pass any one of these variables as an actual parameter to a procedure with a formal parameter that is of `ARRAY OF CHAR` type. Thus it would be legal to write —

```
InOut.WriteString(str1);
InOut.WriteString(str2);
InOut.WriteString(str3);
```

no matter what the style of the variable declarations.

Chapter 7

Expressions

Expressions are used to specify **values**. Every expression in Modula has a value, and a **type**. The type of every expression in Modula may be determined by a mere textual scan of the program. However, it is sometimes impossible to find the value of an expression except by executing the program.

Expressions consist of two kinds of things — the **operands** and the **operators**.

7.1 Designators

Operands of expressions consist of various constants literals — *WholeNumbers*, *RealNumbers*, *litStrings* — together with **constructors** and **designators**. Constructors are dealt with in a later section of this chapter. Designators are used to denote objects, such as variables, constants and procedures.

In expressions, that is, when used as operands, designators denote the *value* of the object that they name. Designators consist of an identifier (possibly qualified by module name), and optional **selectors** in the case of objects of structured type.

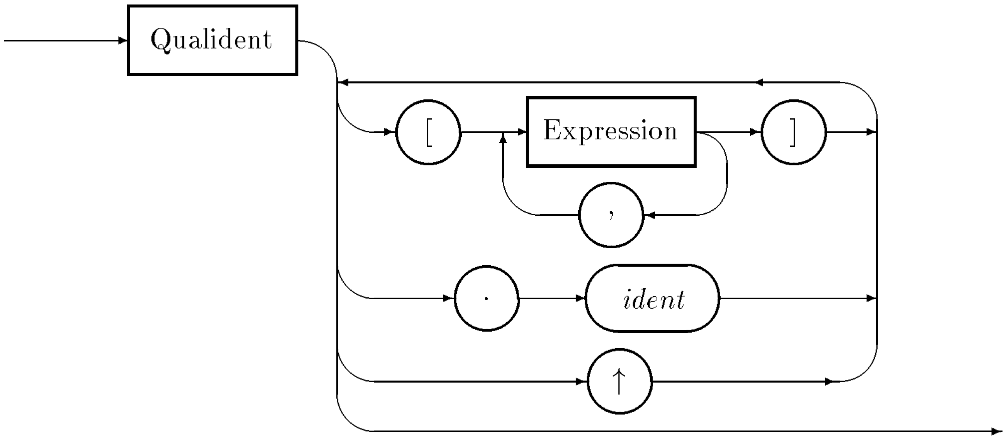
Syntax

Designator	$\rightarrow$	**Qualident** {**Selector**} .
Selector	$\rightarrow$	"`[`" **ExpressionList** "`]`" | $\uparrow$ | "." *ident* .
Qualident	$\rightarrow$	*ident* {"." *ident*} .

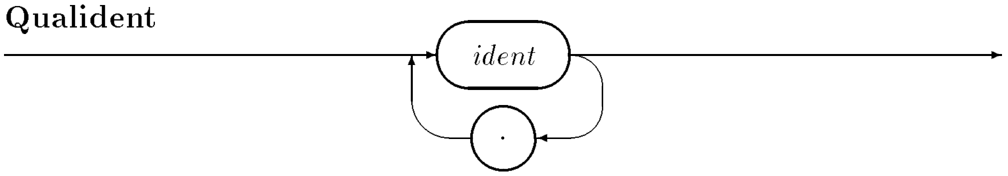
Note that the $\uparrow$ symbol is shown as the carat character '`^`' on most terminals, and is shown in this way in the program fragments in this manual.

The **qualident** (qualified identifier) part of a designator denotes an entire object. This object might be the name of variable, a constant or a procedure. In the case of an object of a structured type a particular component of the object may be selected by means of the optional selectors that follow the identifier.

Designator



Qualident



Selection of the named fields of record types is achieved by use of the *dot* notation. The identifier following the dot is the name of the field to be selected[1]. Here is an assignment statement using field selection

```
range := recVar.last - recVar.first;
```

It is an error if the identifier following the dot is not the name of a field belonging to the type of the object denoted by the (partial) designator that precedes the dot.

Selection of positional components (array elements) is achieved by **indexing**. The index of a designator is the value of the expression enclosed in square brackets. In the case of multiple selection in multi-dimensional arrays, the array indices may be given in a comma-separated expression-list, rather than in separate selectors. Thus the two following designators are equivalent

```
arrayVar[i][j][k]
arrayVar[i,j,k]
```

It is an error if the value of an array index does not lie strictly within the declared bounds of the array index type. Note that since index values are expressions, the values cannot always be determined at compilation time, and hence *runtime checks* are required.

Dereference of pointers is denoted by the use of the 'uparrow' selector ↑ or in typewriter fonts '^'. Such a dereference selects the object that the pointer denotes — it selects the *object that the pointer points to*. It is very common, in pointer linked data structures, for pointer dereference and field selection to be used together. This is because the most common target types of pointers are the record types. Here is a typical example

[1]note that the use of the same dot notation for both module-qualification and field selection is syntactically ambiguous. However, the usage is justified by the fact that in each case the identifier that precedes the dot qualifies the context in which the identifier after the dot is to be interpreted.

```
isLast := (current^.next = NIL);
```

It is an error if the value of the pointer does not denote a dynamically allocated object of the bound type. In particular, it is an error if the value of the pointer is uninitialized or is `NIL`. Such errors can, in general, only be detected during program execution.

7.2 Expressions

Syntax

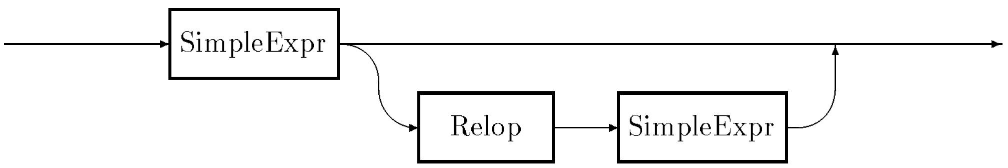
ConstExpr	→	**Expression**.
Expression	→	**SimpleExpression** [**Relop SimpleExpression**] .
SimpleExpression	→	[**Sign**] **Term** {**Addop Term**} .
Term	→	**Factor** {**Mulop Factor**} .
Factor	→	*WholeNumber* | *RealNumber* | *litString*
	|	"(" **Expression** ")"| **Notop Factor** | **Constructor**
	|	**Designator** | **Designator ActualParams** .
Relop	→	`=` | `<>` | # | `<` | `<=` | `>` | `>=` | `IN` .
Addop	→	+ | − | `OR` .
Mulop	→	`*` | / | `REM` | `DIV` | `MOD` | `AND` | &.
Notop	→	`NOT` | `~` .
Sign	→	+ | − .

Expressions denote the rules by which values are to computed. They consist of operators and operands, that are understood in terms of conventional **precedence**. There is further discussion of operator precedence in the *Operators* section of this chapter.

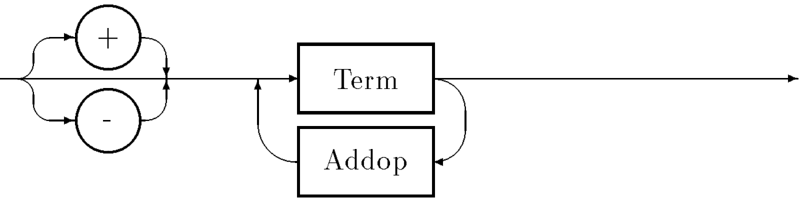
- Expressions are either a single *simple expression* or consist of two simple expressions separated by a relational operator
- Simple expressions consist of one or more *terms* connected by *adding operators*. The complete simple expression may be optionally signed, if it is of numeric type
- Terms consist of one or more *factors* connected by *multiplying operators*
- Factors are either the logical negation of another factor, or consist of a literal, a designator or a constructor

ConstExpr

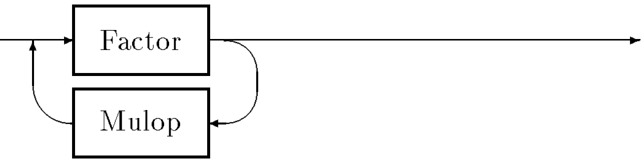
Expression



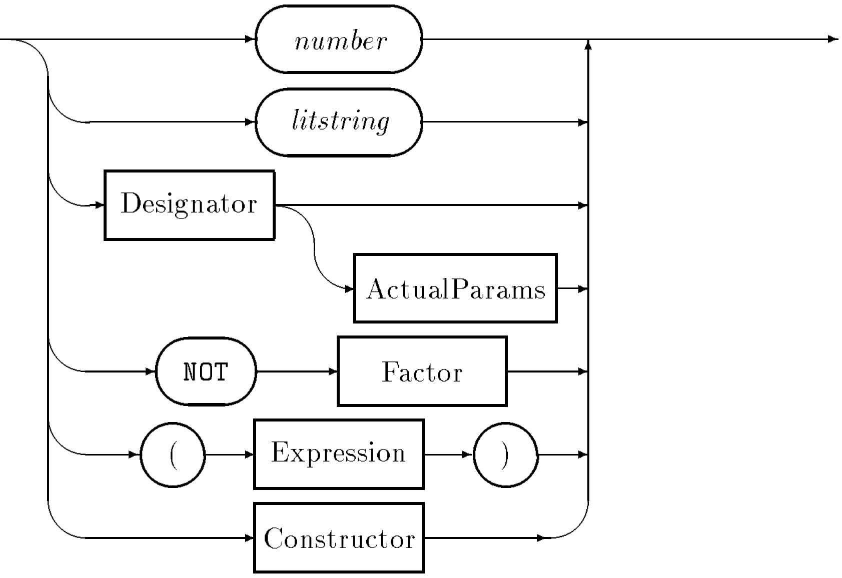
SimpleExpr



Term



Factor



7.3 Operators

Expressions denote the rules by which values are to computed. They consist of operators and operands, that are understood in terms of conventional **precedence**. Not all operators apply to operands of all possible types.

The negation operator `NOT` (or its lexical alternative ~) has the highest precedence, followed by the muliplying operators, **Mulop**s, then the addition operators, **Addop**s, and finally the relational operators, **Relop**s with the lowest precedence. The following interpretations follow from these rules —

Expression	**Equivalent expression**
`NOT` a `OR` b	(`NOT` a) `OR` b
a + b * c	a + (b * c)
a < b + c	a < (b + c)

Within an operator group, evaluation is from left to right. Thus the following interpretations follow —

Expression	**Equivalent expression**
a / b * c	(a / b) * c
a − b + c	(a − b) + c

Parentheses may always be used to ensure a particular order of grouping of operands. Redundant parentheses do not add any execution time overhead and may be used freely. As an example, the following expression would have the same meaning if fully parenthesized as shown.

```
  aVar * 3 + bVar < 5
((aVar * 3) + bVar) < 5
```

Expression compatiblity

In general, the operands of binary operators must be **expression compatible** or more briefly just **compatible**. Compatability may be determined by the application of one or more of the following rules —

- values of all types are compatible with other values of the same type
- subrange types are compatible with their host type
- the constant numeric type *ZZ* is compatible with both `INTEGER` or `CARDINAL`
- the constant real numeric type *RR* is compatible with all the real types
- character literals are compatible with type `CHAR`
- strings of length 0 and 1 may be considered character literals
- the `NIL` value is compatible with all pointer types

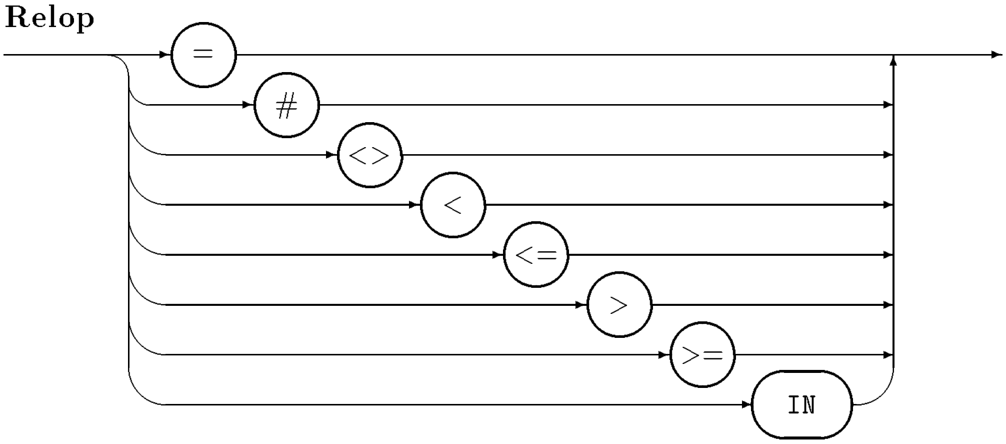
Logical negation operator

The logical negation operator is *unary*, that is, it takes a single operand. It is denoted by `NOT` (or its lexical alternative ˜), and performs the logical negation of its operand. It thus applies only to the Boolean type.

Relational operators

The relational operators perform various computations that produce Boolean results. These operators are said to be *overloaded*, since they have different meaning when applied to operands of different type.

Relop



Ordinal types

For the ordinal types, the *ordering* operators may be used. Thus `=, <>, >, >=, <, <=` apply to all such types, even those such as Boolean and enumeration types that do not have numeric values. All such types are said to be strictly ordered. The meaning of such comparisons is the same as an arithmetic comparison between the numbers to which the ordinal values correspond. The two operands of the ordering operator must be of *expression compatible* type.

Real types

For the real types `REAL, LONGREAL, SHORTREAL`, all of the ordering operators may be used, and return values of the Boolean type. The two operands must be *expression compatible*[2]. Note that the precise meaning of comparisons between real values where one or more are *not-a-number-symbols* may depend on mode settings of the hardware.

Set types

For set types only four of the six ordering operators apply. These are `=, <>, >=, <=`. Equality and its negation have the obvious meaning, while the non strict inequality operations correspond to non-strict set inclusion. Thus if a and b are sets, `a <= b` will have the same truth value as the predicate $a \subseteq b$. The two set expressions must be of *expression compatible* types.

Note particularly the absence of strict inequality operators. When it is absolutely necessary, these may be synthesized by compound expressions such as

```
(set1 <= set2) AND (set1 <> set2)
```

The relational operator `IN` applies to sets, with the meaning of a membership test. In other words `lhs IN rhs` is true if and only if the value of the expression *lhs* is a member of the set denoted by the expression *rhs*. In this case, the left operand must be of ordinal type, and the right operand must be of some set type. The left operand must be *expression compatible* with the *base type* of the set on the right.

[2]In **gpm** `REAL` and `LONGREAL` are type synonyms and are thus compatible. This compatability is not portable to other systems where `REAL` is often the short type

Pointer types

Equality and non-equality apply to pointer and to the *opaque* types that are introduced later, but no meaning can be given to other ordering comparisons on such types. As usual, the two values must be of *expression compatible* types.

There are no built-in relational operators for structured types. If these are required they must be created by programming value returning procedures. In particular there are no built-in relational operators for the character string types, however, there are functions for this purpose in the libraries.

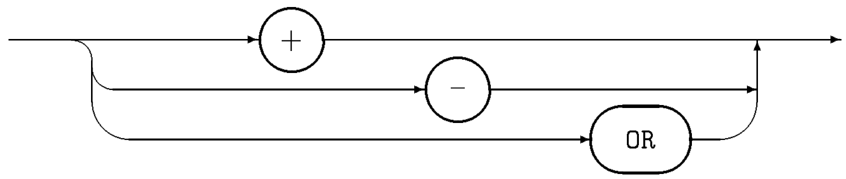
Procedure types

Equality and non-equality apply to procedure types, but no meaning can be given to other ordering comparisons on such types. As usual, the two values must be of *expression compatible* types. In the case of procedure types this is a structural equivalence test. As a matter of fact, these two tests (together with assignment, and invocation) are the *only* operations that can be performed on objects of procedure type.

Adding operators

The adding operators are binary operators with lower precedence than the multiplying operators. As usual, not all operators apply to all types.

Addop



Numeric types

For all numeric types, the valid adding operators are + and −. These denote addition and subtraction respectively. The two operands must be *expression compatible*. These two operators may also be used as sign operators on the first **term** in a simple expression. In such a case, the sign operates on the value of the first term only. Thus $-a + b$ is equal to $(-a) + b$ and not $-(a + b)$.

Boolean type

The final adding operator `OR` denotes logical disjunction. The expression a `OR` b evaluates to `TRUE` if either or both of the operands are true. In this case both operands must be of Boolean type. Note however that expressions involving `OR` are evaluated in a special way. Each term is evaluated from left to right in the expression, with evaluation stopping as soon as any true-valued term is encountered. This is referred to as *short-circuit evaluation.* In particular, suppose that a has the value `TRUE` while the value of b is undefined. In such a case, the expression a `OR` b has the value `TRUE`, while b `OR` a would be undefined.

Short circuit evaluation is often used to allow safe evaluation of expressions where one or more components may be undefined. The idea is that an earlier term in the expression (that is, one to the left of the unsafe term) acts as a **guard expression**. It the guard expression is `TRUE`, the unsafe expression is not evaluated. An example is the expression `(ptr = NIL) OR ptr^.flag`. If the pointer has the `NIL` value, the pointer is not dereferenced.

The programming convenience of this particular construction is believed to outweigh the disadvantage of invalidation of the normal commutative property that would be expected in a Boolean algebra.

Set types

For set types, the operators + and − apply. The meaning of the operations is set union and set difference respectively.

$$S_1 + S_2 = \{e : e \in S_1 \texttt{ OR } e \in S_2\}$$

$$S_1 - S_2 = \{e : e \in S_1 \texttt{ AND NOT } e \in S_2\}$$

The two operands must be of the same set type.

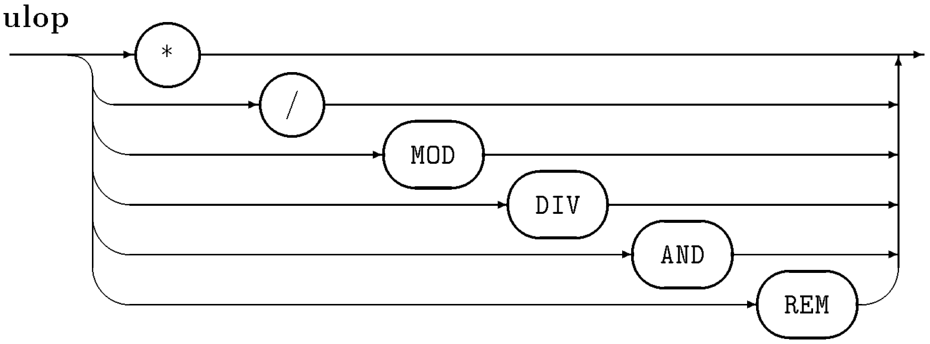
String literals

The only operator defined for string types is +, denoting string concatenation. It applies only to literals. Other operations, and operations including variables, are available via procedures imported from the standard strings library.

Multiplying operators

The multiplying operators are binary operators with higher precedence than the adding operators. As usual, not all operators apply to all types.

Mulop



Whole number types

The multiplying operators that apply to the whole number types are `*`, `DIV MOD` and two recent additions `/, and REM`. The asterisk character denotes multiplication. The pair of operators `DIV` and `/`, each denote whole number division. They produce the same result for positive operands, but differ in their treatment of signed operands if one or both of the operands is negative.

Similarly the pair of operands `MOD` and `REM` produce the same result for positive operands, but differ in their treatment of signed operands if one or both of the operands is negative.

The following table summarizes the values produced by these four operators for operands of every possible combination of signs —

op	31 **op** 10	31 **op** (-10)	(-31) **op** 10	(-31) **op** (-10)
/	3	-3	-3	3
`REM`	1	1	-1	-1
`DIV`	3	Exception	-4	Exception
`MOD`	1	Exception	9	Exception

Note that it is an error if the right operand of either `DIV` or `MOD` have a negative value. Also, it is an error if the right operand of any of these operators is the value zero. In all cases the two operands must be *expression compatible*.

Real types

For real operands, the applicable operands are `+`, `-`, `*`, and `/`. These denote a implementation defined approximation to the addition subtraction, multiplication and division operations. As usual, the two operands must be *expression compatible*. For the division operation, the right hand operand must not be zero.

Boolean type

For the Boolean type, the only multiplying operator is `AND`. It denotes the logical conjunction of its two operands. That is to say, the expression a `AND` b has the value `TRUE` if and only if both operands are true. Both operands must be of Boolean type. Note however that expressions involving `AND` are evaluated in a special way. Each term is evaluated from left to right in the expression, with evaluation stopping as soon as any true-valued term is encountered. This is referred to as *short-circuit evaluation*. In particular, suppose that a has the value `FALSE` while the value of b is undefined. In such a case, the expression a `AND` b has the value `FALSE`, while b `AND` a would be undefined.

Short circuit evaluation is often used to allow safe evaluation of expressions where one or more components may be undefined. The idea is that an earlier factor in the expression (that is, one to the left of the unsafe factor) acts as a **guard expression**. It the guard expression is `FALSE`, the unsafe expression is not evaluated. An example is the expression `(ptr <> NIL) AND ptr^.flag`. If the pointer has the `NIL` value, the pointer is not dereferenced.

The programming convenience of this particular construction is believed to outweigh the disadvantage of invalidation of the normal commutative property that would be expected in a Boolean algebra.

Set types

For set operands, the applicable operands are `*` and `/`. The asterisk denotes set intersection, while the slash denotes set symmetric difference.

$$S_1 * S_2 = \{e : e \in S_1 \texttt{ AND } e \in S_2\}$$

$$S_1/S_2 = \{e : (e \in S_1 \texttt{ OR } e \in S_2)\texttt{AND NOT } e \in (S_1 * S_2)\}$$

The two operands must be of the same set type.

The effect of the four set operations may be summarized as follows —

op	{1, 3, 5, 7} **op** {5, 7, 9}
+	{1, 3, 5, 7, 9}
−	{1, 3}
*	{5, 7}
/	{1, 3, 9}

7.4 Function applications

In an expression, a function application is denoted by the identifier of the function, together with a (possibly empty) parameter list. The meaning of a function application is the value of the function, evaluated with the actual parameters substituted for the formal parameters that were declared in the function declaration. The parameter list is not optional — the name of a function without a parameter list denotes the *function itself* and not the result of applying the function. Consider the following program fragment, which contains two assignments, both of which are correct —

```
TYPE SeqBoolProc = PROCEDURE(Sequence) : BOOLEAN;
VAR  bProc : SeqBoolProc;
     ended : BOOLEAN;
     seq   : Sequence;

BEGIN
  ...
  bProc := CardSequences.IsEmpty;
  ended := CardSequences.IsEmpty(seq);
  ...
```

The first of these assignments assigns the function procedure *IsEmpty* to the procedure variable *bProc*. The second assignment *applies* the function procedure to the actual parameter *seq*. In the first case the type of the expression on the right hand side of the assignment is *SeqBoolProc*, while in the second case the type is *Boolean*.

Except for a few of the built in procedures that obey special rules, the type of the value returned by a function application is the return value type of the function declaration.

As explained in the chapter on procedure declarations, the actual parameters must be *parameter compatible* with the declared formal parameter types and modes.

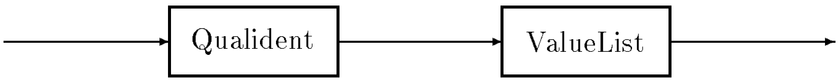
7.5 Constructors

Syntax

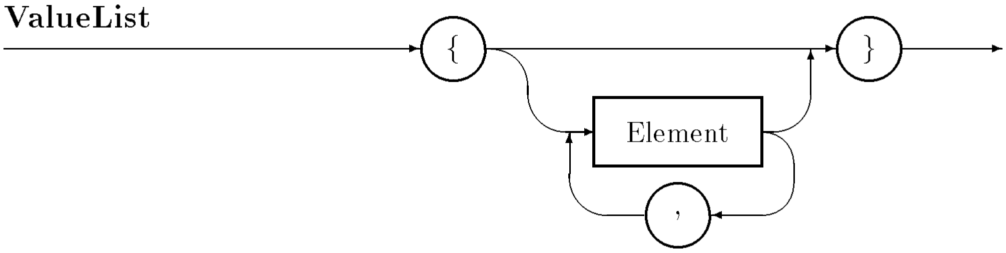
Constructor	→	**Qualident ValueList**.
ValueList	→	"{" **Element** {, **Element** } "}".
Element	→	**SetElement** | **Component** .
SetElement	→	**Expression** [.. **Expression**].
Component	→	(**Expression** | **ValueList**) [BY **ConstExpr**].

Original Modula possessed only set constructors, but current standards allow constructors for record and array types as well[3].

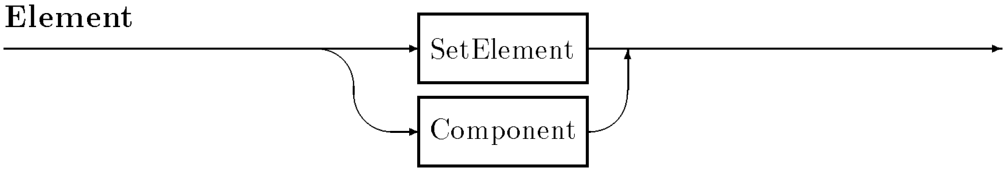
Constructor



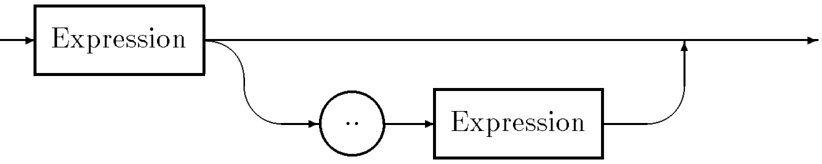
ValueList



Element



SetElement



Constructors consist of a possibly empty list of elements from which the value is to be constructed. In the case of set constructors, the order of the elements in the constructor is not important. In the case of records and arrays, the values must be listed in an order that corresponds to the order of the components of the type.

The replicator BY **ConstExpr** clause only applies to array constructors, even though it is sometimes meaningful for records that have repeated fields of the same type. Semantic restrictions include the fact that any repeat counts must be non-negative, whole-number constants, and the total number of elements (taking into account any repetition of elements) must exactly match the number of elements in the structure. Values must be assignment

[3]At the time of this edition of this manual **gpm** versions other than **gpm-pc** only permit **constant constructors** for record and array types

compatible with the component type to which the equivalent assignment is being made. Thus a valid record initialization might be –

```
TYPE  NamType = ARRAY [0 .. 15] OF CHAR;
      RecType = RECORD
                  name  : NamType;
                  b,c,d : CARDINAL;
                END;

CONST initial = RecType{"anon", 0, 0, 0};
```

It should be noted that the elements of a constructor may themselves contain lists of elements, and that such nested constructs do not need to specify a typename, although they are free to do so. This relaxation is necessary for multidimensional arrays, where the types of the inner components may be anonymous. Consider the array –

```
TYPE  Matrix  = ARRAY [0 .. 2], [0 .. 2] OF REAL;
```

This is an array of three elements each of which is an array of three reals *which has no type name*. In this case a constructor may be specified as follows –

```
CONST initial = Matrix {{1.0, 0.0, 0.0},
                        {0.0, 1.0, 0.0},
                        {0.0, 0.0, 1.0}};
```

Of course it is possible to name the inner type as follows –

```
TYPE  Vector  = ARRAY [0 .. 2] OF REAL;
      Matrix  = ARRAY [0 .. 2] OF Vector;
CONST initial = Matrix {Vector{1.0, 0.0, 0.0},
                        Vector{0.0, 1.0, 0.0},
                        Vector{0.0, 0.0, 1.0}};
```

As a final example, here is an alternative initialization of the same record type that was given earlier —

```
TYPE  NamType = ARRAY [0 .. 15] OF CHAR;
      RecType = RECORD
                  name  : NamType;
                  b,c,d : CARDINAL;
                END;

CONST initial = RecType{{" " BY 16}, 0, 0, 0};
```

The use of constant value constructors facilitates the sensible initialization of structures. In some speed critical programs it also can lead to faster code, by moving array index and offset computations from runtime to compile time. Against this must be weighed the fact that, as a new language construct, the feature is not present in *any* previous Modula compilers. The use of the feature may thus make programs difficult to port to machines for which **gpm** is not yet available.

Chapter 8

Statements

The executable statements of Modula-2 comprise:

- the assignment statement, for storing values in variables,
- the procedure call statement, which invokes a procedure, passing parameters,
- the conditional statements `IF` (two-way) and `CASE` (multi-way),
- the iterative statements `WHILE`, `REPEAT`, `FOR` and `LOOP`,
- the `WITH` statement, which applies record field qualification to a statement.
- the `EXIT` statement, which terminates a `LOOP`
- the procedure termination and function result `RETURN` statement,

Syntax

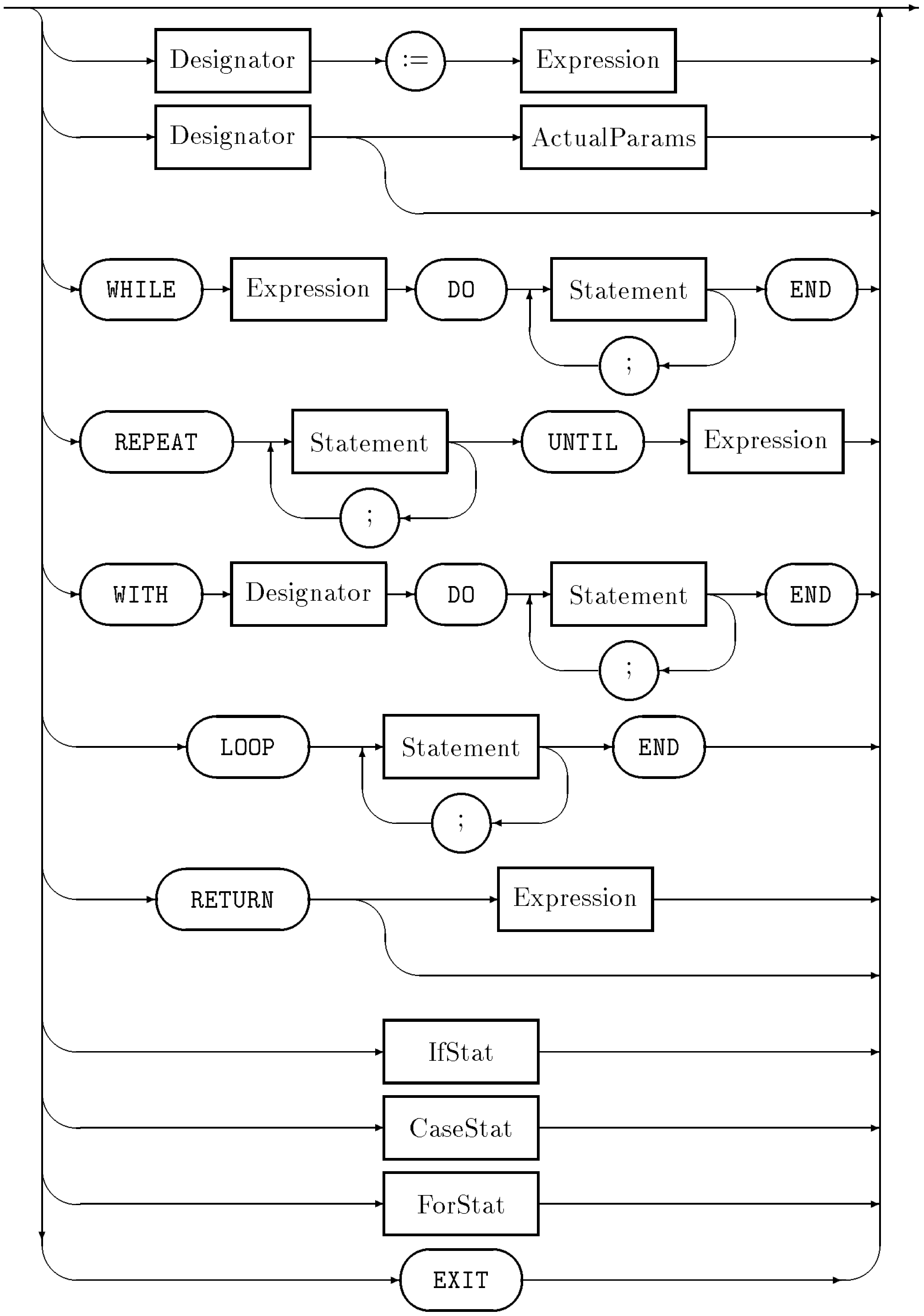
Statement	→	**Assignment** | **ProcCall** | **CaseStat** | **IfStat**
	|	**WhileStat** | **RepeatStat** | **LoopStat** | **ForStat**
	|	**WithStat** | **ExitStat** | **ReturnStat** | .

Assignments, procedure calls, `RETURN`, and `EXIT` are unstructured statements – that is, they do not contain parts that are themselves statements; the other statement types all contain statement (sequences) and are classed as structured statements.

8.1 The empty statement

Anywhere in the Modula syntax where a statement can go, it is perfectly legal to to have an empty statement. The empty statement does not perform any action, and usually exists only as a formatting convenience.

Statement


Designator
:=
Expression
Designator
ActualParams
WHILE
Expression
DO
Statement
;
END
REPEAT
Statement
;
UNTIL
Expression
WITH
Designator
DO
Statement
;
END
LOOP
Statement
;
END
RETURN
Expression
IfStat
CaseStat
ForStat
EXIT

8.2 Assignments

Syntax

Assignment $\rightarrow$ **Designator := Expression**.

The assignment statement evaluates an expression, and assigns the result to a variable (stores the result in the variable). The assignment operator ":=" separates the variable and the assigned expression.

The variable is denoted by a designator, that comprises a (possibly qualified) identifier, and optional selectors. The qualified identifier identifies a variable object of some module; in the case of the current module or unqualifying import, no qualification may be needed. The selectors are used to specify a component of a structured variable, using the appropriate combination of record field selection with '.', array element selection with '[...]', and pointer dereferencing with '^'.

The expression must be assignment compatible with the variable. Assignment compatibility is an extension of expression compatibility — an expression and a variable are assignment compatible if[1]

- they are *expression* compatible, or
- they are `CARDINAL` and `INTEGER` (in either order), or
- the variable is of pointer type and the expression is `NIL`, or
- the variable is a procedure variable and the expression is a procedure with the same structure (proper procedure or function, number and types of parameters and result), or
- the variable is `CHAR` and the expression is a string of length 0 or 1, or
- the variable is a fixed-size array of characters and the expression is a string constant whose length is less than or equal to the number of elements in the array.

Entire arrays or records may be assigned; the effect is the same as element-by-element or field-by-field assignment, except that undefined values may be assigned.

When a string constant is assigned to an array of characters and the string does not fill the array, an end-of-string character *Ascii.nul* is appended and the rest of the array becomes undefined.

When assignment to the tag field of a record variant-part causes a new variant to be selected, all the fields of the old variant become undefined. The standard also requires that it is an exception if a value from a non-active component of a variant record is accessed. Together, these strictures would prevent even the infrequent but entrenched use of undiscriminated variant records to circumvent type checking. Like most compilers, **gpm** does not make fields undefined or check which variant is active, so that assignment to a tag field has no effect on the values of other fields, and access to non-active variants is allowed.

The order of evaluation of the expression and the variable designator in an assignment is not defined by the language. Thus programs will not be portable if they rely on evaluation

[1] Some additional special cases which involve the low-level types BYTE, WORD and ADDRESS are added later

being in a particular order. This can happen if the evaluations have side-effects. As an example if a function procedure *Pop* removes and returns the top element from a stack, the effect of the statement —

```
array[Pop()] := array[Pop()]; (* very bad ! *)
```

will be quite unpredictable. Even on the same compiler the effect might change with the optimization level of the compilation. For clarity as well as portability, good programming practice would avoid such side effects.

Examples:

```
(*1*)   length := minLength;
        nodePtr^.keys[index] := key;
        tree.CompareProc := CompareComplex;
        result := (item[index] # undefined) AND
                  (tree.CompareProc(item[index],key) = equal);
(*2*)   TYPE
          Row = ARRAY [1..8] OF Contents;
          Board = ARRAY [1..8] OF Row;
        VAR
          row : Row;
          board : Board;
        ...
        board[1] := row;
```

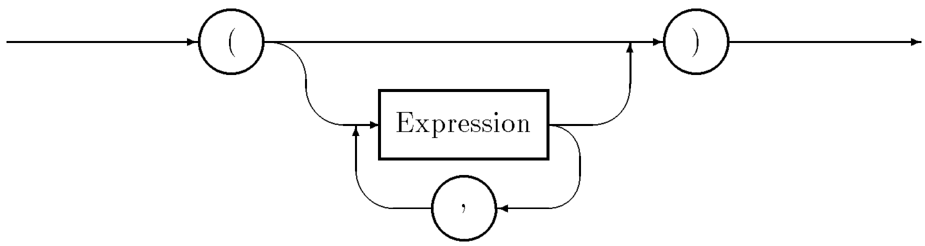
8.3 Procedure calls

Syntax

ProcCall	→	**Designator** [**ActualParams**].
ActualParams	→	"(" **ExpressionList** ")".
ExpressionList	→	**Expression**{, **Expression**}.

A procedure call invokes a proper procedure, passing parameters which supply input values and recipients for output values. It comprises the name of the called procedure, or the name of a procedure variable, followed by a list of actual parameters enclosed in parentheses. The number of parameters must be the same as the number of formal parameters given in the procedure declaration, and they must be parameter compatible with the corresponding formal parameters.

ActualParams



Value parameters, which may be arbitrary expressions, are evaluated and the result passed to the procedure as an initialised local object of the procedure, with the name of the corresponding formal parameter; they thus serve as inputs to the procedure and local variables. Variable (reference or `VAR`) parameters, which must be variables, are directly accessed by the procedure wherever the corresponding formal parameter is designated; they thus serve as both inputs to and outputs from the procedure.

The rules for parameter compatibility reflect the two parameter-passing mechanisms. For **value** formal parameters, an actual parameter is *parameter compatible* if it is —

- assignment compatible (see section 8.2) with the corresponding formal parameter, or
- the formal parameter is an open array of some type, and the actual parameter is an array of elements of an assignment compatible type, or
- the formal parameter is of type `BYTE` or `WORD` and the actual parameter is of byte or word size, or
- the formal parameter is an open array of type `BYTE` or `WORD`, and the actual parameter is *any* type[2]

For **variable** formal parameters, an actual parameter is *parameter compatible* if it is —

- of identical type to the corresponding formal parameter, or
- the formal parameter is an open array of some type, and the actual parameter is an array of elements of an identical type, or
- the formal parameter is of type `BYTE` or `WORD` and the actual parameter is of byte or word size, or
- the formal parameter is an open array of type `BYTE` or `WORD`, and the actual parameter is *any* type[3]

The order of evaluation of procedure designator and actual parameters is implementation-dependent, but all are evaluated before the procedure block is executed. A program that depends on a particular evaluation order is incorrect, and will be non portable.

Examples:

```
Lookup (dictionary, key, success);
homeBucket := Hash(key);
```

[2]**gpm** actually insists that the actual parameter is at least as strictly aligned in the underlying hardware as is the formal parameter

[3]as for value parameters, **gpm** insists that the actual parameter is at least as strictly aligned as is the formal parameter

8.4 Statement sequences

Syntax

StatSequence → **Statement**{; **Statement**}.

A statement sequence expresses the concept of a high-level algorithmic step that has been translated into a sequence of Modula-2 statements. The sequence comprises any number of statements, separated by ";". Statement sequences form the 'bodies' of module or procedure blocks, and of looping and other structured statements. Since each of the statements in a sequence may itself be a structured statement, arbitrarily complex statement structures may be built.

Though the syntax requires at least one statement in a sequence, and a ';' only between statements, an empty statement is valid. Thus a statement sequence may be completely empty, and a ';' may precede the first or follow the last statement in a sequence.

Examples: see the examples for LOOP, RETURN and WITH.

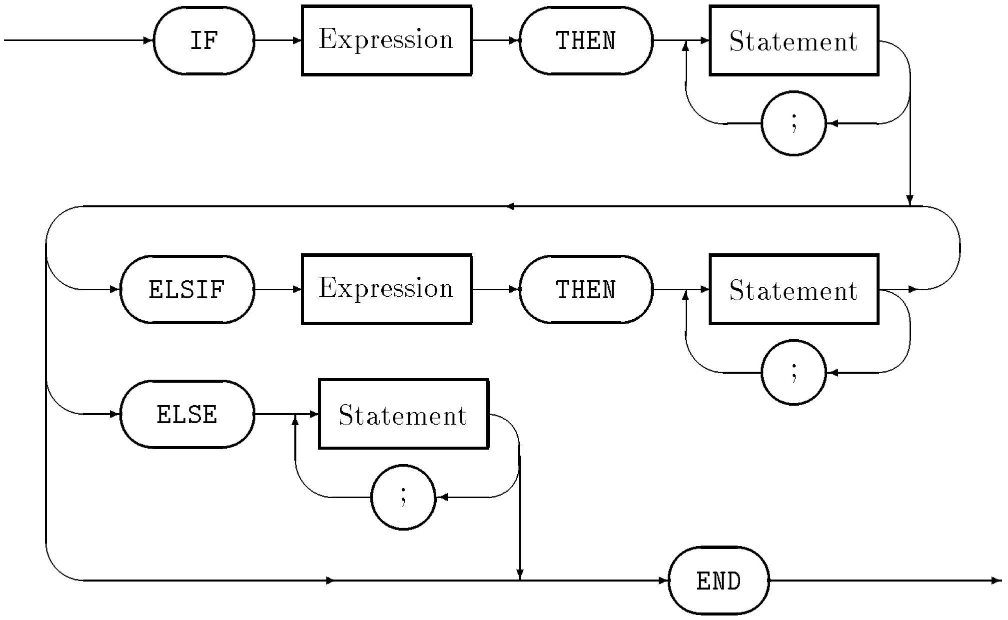
8.5 IF statements

Syntax

IfStat → `IF` **Expression** `THEN` **StatSequence**
{`ELSIF` **Expression** `THEN` **StatSequence**}
[`ELSE` **StatSequence**] `END`.

The `IF` statement expresses the conditional construct of algorithms, where one of a number of alternative statement sequences is chosen on the basis of Boolean expressions.

IfStat



The syntax specifies a single `IF` statement, but options and repetitions in the syntax lead to a number of common forms. The simplest form chooses between one statement sequence, executed if the Boolean expression (the 'guard') evaluates to `TRUE`, and no action at all (if the guard evaluates to `FALSE`):

```
IF BooleanExpression THEN statementSequence END
```

Another form also specifies a statement sequence to be executed if the guard evaluates to `FALSE`:

```
IF BooleanExpression THEN
  statementSequence
ELSE
  statementSequence
END
```

Others allow for any number of guards, before a final sequence to be executed if all fail:

```
IF BooleanExpression THEN
  statementSequence
ELSIF BooleanExpression THEN
  statementSequence
ELSIF BooleanExpression THEN
  statementSequence
ELSE
  statementSequence
END
```

Thus each of the simpler forms is a special case of this final general form. Note that any sequence may be empty, specifying no action if that guard succeeds.

The evaluation of the Boolean expressions occurs in the order stated, until one succeeds; each is evaluated short-circuit. Thus a suitable ordering of expressions and subexpressions can avoid any errors due to dependencies such as array indexes out of range, pointers `NIL`, etc.

Examples:

```
IF pointer = NIL THEN
  Error ("Invalid pointer in structure");
ELSIF pointer^.count = 0 THEN
  Error ("Empty occurrence list");
ELSE
  PrintList (pointer^.occurrences);
END;
```

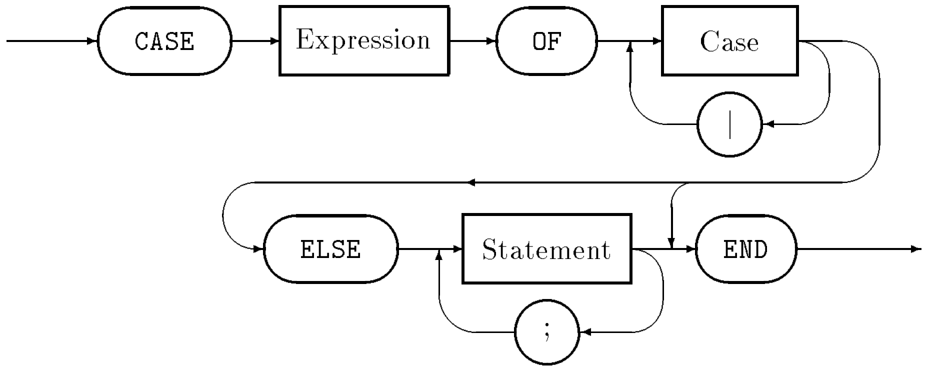
8.6 CASE statements

Syntax

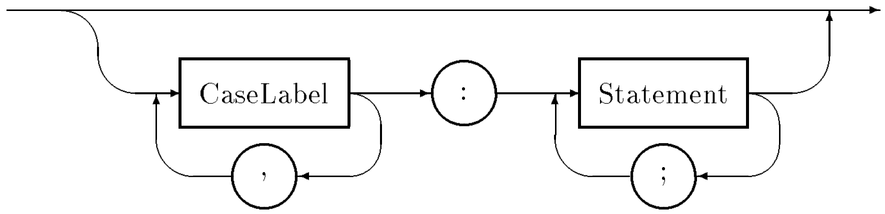
CaseStat	→	`CASE` **Expression** `OF`
		Case{“|” **Case**} [`ELSE` **StatSequence**] `END`.
Case	→	[**CaseLabelList** : **StatSequence**].
CaseLabelList	→	**CaseLabel**{, **CaseLabel**}.
CaseLabel	→	**ConstExpr**[. . **ConstExpr**].

The CASE statement also expresses the conditional construct of algorithms, but selects on the basis of the value of a single expression matching one of a number of alternatives.

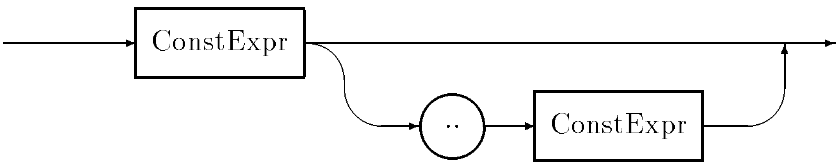
CaseStat



Case



CaseLabel



The overall form of the CASE statement is

```
CASE selector OF
| case
| case
  ...
END
```

where each case comprises a list of the values that select it for execution, and a statement sequence to be executed; each case is separated from the next by ‘|’. An optional final case labelled with `ELSE` specifies a default statement sequence to be executed if the selector expression does not match any case label; if the default is not supplied, failure to match a case label is an error.

The selector expression must be of ordinal type, and expression-compatible with each of the case labels. Each label must be a constant expression, or a constant range expressed as *lowerLimit .. upperLimit*. All labels must be distinct, and non-overlapping in the case of ranges.

Examples:

```
CASE nodeType OF
  internal : IF ...
               ...
               ...;
| leaf     : INC (referenced);
END;

CASE markEntered OF
|  0..24 : grade := lowFail;
| 25..46 : grade := fail;
...
| 85..100 : grade := highDistinction;
ELSE WriteString ("Invalid mark");
     WriteLn;
END;
```

Note that an empty case is valid, allowing the common style of prefixing the first case label list with '`|`' so that each case is clearly signalled.

8.7 WHILE statements

Syntax

WhileStat $\rightarrow$ `WHILE` **Expression** `DO` **StatSequence** `END`.

The `WHILE` statement is the pre-tested, condition-controlled loop structure. A Boolean expression is (short-circuit) evaluated, and if the result is TRUE the statement sequence which forms the loop body is executed; control then returns to the re-evaluation of the Boolean expression, and so on, until the test fails and the entire `WHILE` terminates.

Due to the pre-test, the loop body may never be executed.

Examples:

```
index := first;
WHILE (index <= limit) and (values[index] <= sought) DO
  INC (index);
END;
```

```
(* Assert: index in [1..limit] and values[index] = sought
           or
           index = limit+1 *)
```

8.8 REPEAT statements

Syntax

RepeatStat $\rightarrow$ `REPEAT` **StatSequence** `UNTIL` **Expression**.

The `REPEAT` statement is the post-tested, condition-controlled loop; it is thus similar to the `WHILE` loop, but with the test for loop continuation applied after each execution of the loop body. Thus the loop body executes at least once; the precondition of the loop must ensure that this is valid.

Examples:

```
index := first-1;
(* Assert: sought is in array values, at position beyond index
           (e.g. by appending as sentinel) *)
REPEAT
  INC (index);
UNTIL values[index] = sought;
(* Assert: values[index] = sought *)
```

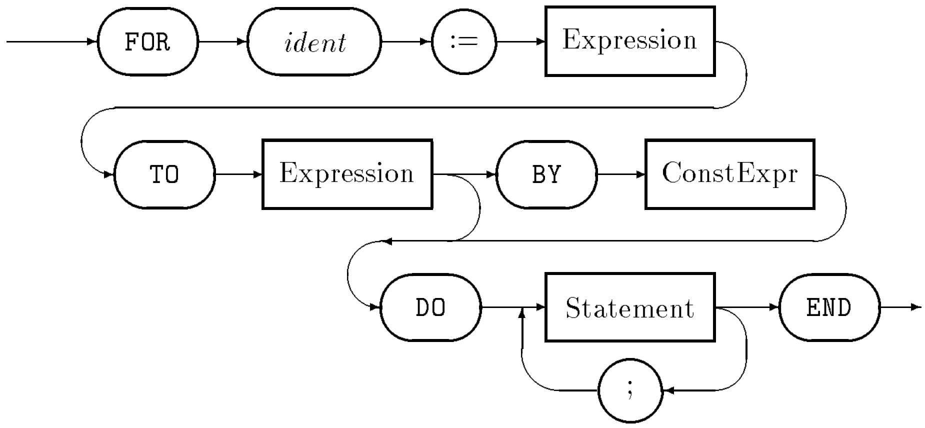
8.9 FOR statements

The `FOR` statement is the pre-tested, count-controlled loop structure. It is appropriate where it is desired to iterate over each of a known range of values. The range is established by evaluating initial and final limit expressions once only before the loop body is executed; a control variable is then iterated over each value in the range, and the loop body — a statement sequence — is executed for each value. Due to the pre-test, the loop body may not be executed at all. The single evaluation of the limit expressions, and protection of the control variable from modification, ensure that the expected number of iterations is performed.

Syntax

ForStat $\rightarrow$ `FOR` *ident* `:=` **Expression** `TO` **Expression**
[`BY` **ConstExpr**] `DO` **StatSequence** `END`.

ForStat



The control variable must be of *ordinal type*. The initial limit expression must be assignment-compatible with it, and the final limit expression must be expression-compatible with it. In the simpler form of `FOR` statement, the iteration is from the initial limit value to the final limit value inclusive, incrementing the control variable after each execution of the loop body.

If an optional step is specified by the `BY` clause, it gives the difference in ordinal values between successive values of the control variable; thus it may be used to step the control variable through, say, every third value, or downward from a higher initial value to a lower final value. If the step is not 1 or −1, the iteration terminates when the control variable would have passed the final value in the appropriate direction.

The protection of the control variable from alteration (except by the steps of the `FOR` control structure) is strong: the control variable may not be 'threatened' by assignment, passing as a variable parameter, uplevel access by nested procedures, or export. In recognition of its local purpose, it must also be a simple variable declared in the block containing the FOR loop.

After completion of the `FOR` loop, the value of the control variable is undefined.

Examples:

```
FOR index := 1 TO limit DO
  sum := sum + reading[index];
END;

FOR month := dec TO jan BY -2 DO
          (* dec, oct, aug, jun, apr, feb *)
 IF rainfall[state, month] > 100.0 THEN
   WriteCard (rainfall[state, month],6);
 END;
END;
```

8.10 LOOP and EXIT statements

Syntax

LoopStat → `LOOP` **StatSequence** `END`.
ExitStat → `EXIT`.

The `LOOP` statement is a general iterative structure, with the loop termination specified by one or more `EXIT` statements within its statement sequence body. Thus it can be used to simulate each of the other loop structures, but in less clear and robust ways. Good programming practice dictates that it should be used only where none of the other loop structures is appropriate, and that the number of `EXIT`s be small.

An `EXIT` statement terminates the execution of the immediately enclosing `LOOP` statement; execution resumes at the statement following the `LOOP END`. If it is desired to exit from further enclosing `LOOP` structures, a further `EXIT` in each is needed. If an `EXIT` occurs within a `WHILE`, `REPEAT` or `FOR` loop nested within a `LOOP`, both the inner loop and the enclosing `LOOP` are terminated. (Again, good practice suggests that such a violation of the apparent intent of the nested `WHILE`, `REPEAT` or `FOR` should be used with discretion.)

Examples:

```
LOOP (* The classic "n&1/2-times" iteration *)
  Read (ch);
  IF ch = terminator THEN EXIT END;
  Process (ch);
END;
```

8.11 RETURN statements

Syntax

ReturnStat → `RETURN` [**Expression**].

The `RETURN` statement is used to return from a module, proper procedure or function procedure, in the latter case supplying the function result.

A `RETURN` in a module or procedure block simply terminates execution of the block at that point, rather than the more common termination by reaching the final `END`; clearly, it must be a simple return, with no result expression.

A `RETURN` from a function procedure also terminates the execution at that point, and provides the function result. The result expression must be assignment-compatible with the declared result type of the function. There may be any number of `RETURN` statements in a function, though good practice suggests few. It is an error to reach the `END` of a function procedure (i.e. without encountering a `RETURN`).

Examples:

```
PROCEDURE Factorial (number : CARDINAL) : CARDINAL;
(* Return number!,
   or the maximum cardinal if overflow would occur. *)
```

```
CONST
  limit = 12;        (* 13! > 2**31 *)
BEGIN
  IF number = 0 THEN
    RETURN 1;
  ELSIF number > limit THEN
    RETURN MAX(CARDINAL);
  ELSE
    RETURN number * Factorial(number-1);
  END;
END Factorial;
```

8.12 WITH statements

Syntax

WithStat $\rightarrow$ `WITH` **Designator** `DO` **StatSequence** `END`.

A `WITH` statement provides automatic qualification of the field names of a record object within its statement sequence, avoiding the need for frequent repetition of the qualification in a statement sequence that repeatedly refers to fields of the record. The designator between '`WITH`' and '`DO`' must designate a record variable or constant. Reference to a field of that record object is allowed without qualification by the record name.

The qualifying designator is evaluated once, before executing the statement sequence. Thus any change to variables that might alter the value of the qualifying designator has no effect on the qualification (see example 2).

Examples:

```
(*1*)   WITH date DO
          day := 12;    (* equivalent to date.day *)
          month := Dec;
          year := 1990;
        END;

(*2*)   TYPE
          ReadingIndex = [1..maxReadings];
          Element = RECORD
                      name : NameString;
                      readings : ARRAY ReadingIndex OF INTEGER;
                    END;
          Table   = ARRAY ReadingIndex OF Element;
        VAR
          table : Table;
          i : CARDINAL;

        ...
```

```
i := 5;
WITH table[i] DO
  name := inputString;
  FOR i := 1 TO maxReadings DO (*  Not good form, but valid -
                                *  WITH still qualifies table[5]
                                *)
    ReadInt (readings[i]);
  END (* FOR *) ;
  (* i is now undefined *)
END (* WITH *) ;
```

Chapter 9

Procedure declarations

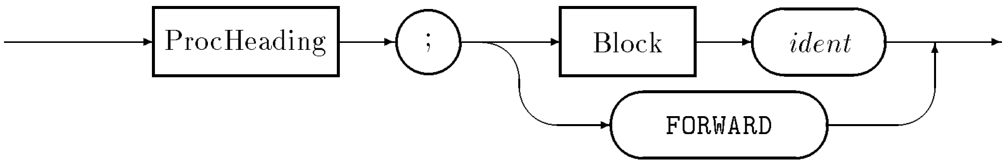
Syntax

ProcDeclaration	→	**ProcHeading** ; (**Block** *ident* | `FORWARD`) .
ProcHeading	→	`PROCEDURE` *ident* [**FormalParams**] .
FormalParams	→	"("[**FPSection** {; **FPSection** }] ")"[: **Qualident**] .
FPSection	→	[`VAR`] **IdentList** : **FormalType** .
FormalType	→	[`ARRAY OF`] **Qualident** .

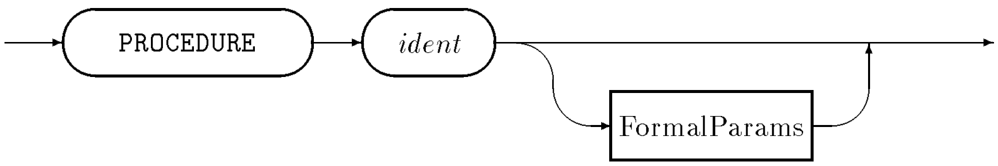
Procedure declarations consist of a **procedure heading** and a **procedure body**. The procedure heading names the procedure, and lists the types and modes of the **formal parameters**. The procedure body consists of a **Block** followed by a further occurrence of the procedure name. The block contains declarations of any local objects of the procedure, and declares the statement sequence which is to be executed when the procedure is invoked.

In **forward declarations**, the body is replaced by the keyword `FORWARD`. **gpm** does not require the use of forward declarations but supports their use in the interests of portability.

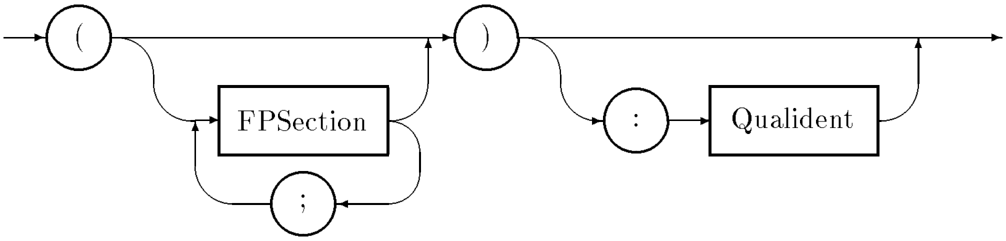
ProcDeclaration



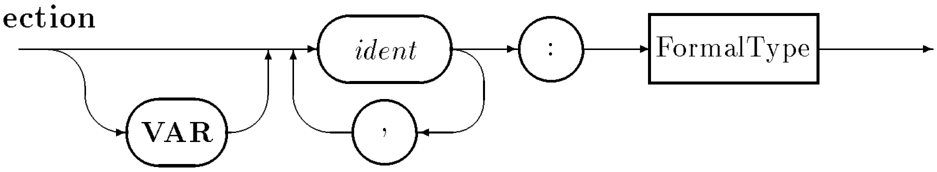
ProcHeading



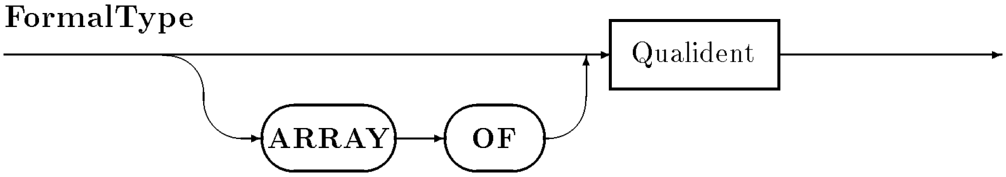
FormalParams



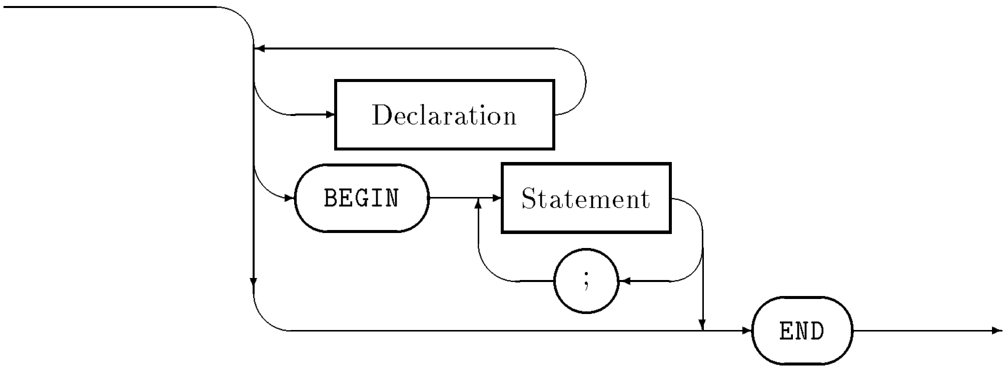
FPSection



FormalType



Block



There are two kinds of procedures — **function procedures**, that return values of some prescribed type, and **proper procedures** that do not have a return value. The value returned by a function procedure is of the type denoted by the (possibly qualified) identifier that optionally ends the formal parameter list. Function procedures are invoked by using the function identifier, together with an *actual parameter list* within an expression. Proper procedures are invoked by using the procedure identifier, together with an *actual parameter list* in a *procedure call* statement.

9.1 Formal parameters

Formal parameters are dummy identifiers that denote data objects, the values of which may be referenced within the procedure block. The actual data objects that are used are specified in the **actual parameters** of each procedure call. The correspondence between actual and formal parameters is determined at the time of the call, so that the same procedure may process different data objects on each occasion that it is called.

Formal parameters are specified as being of some particular **formal type**, and either of **value** or **variable** mode. Variable mode parameters are distinguished by the presence of the keyword `VAR` preceding the dummy name of the parameter.

Variable mode parameters are *references* to the actual variable that is specified at the procedure call. Because of this, any statement within the body of the procedure that changes the value of the formal parameter changes the value of the actual variable that the formal parameter denotes. Variable parameters are sometimes called *inout* parameters. It is an error if an actual parameter corresponding to a variable formal parameter does not denote a variable.

Value mode parameters denote local variables of the procedure. The initial values of these variables are determined by the evaluation of actual parameter expressions at the time of the procedure call. The values of these variables may be changed within the procedure body without affecting any variables that are involved in the actual parameter expressions. Value parameters are sometimes called *in* parameters since they carry values into the procedure but do not carry values out.

The formal type of every formal parameter is specified either as a (possibly qualified) type name, or as an **open array type**. Open arrays are arrays of some particular type, specified by the syntax `ARRAY OF` *typename*, where as usual the type name is an optionally qualified identifier. Open arrays permit the procedure to accept as actual parameter any array with the specified element type. Procedures may thus be devised that process arrays of arbitrary size. In such a case, the size of the array is determined by the size of the actual array that the formal denotes on any particular procedure call.

It is possible to declare several formal parameters all of the same mode and formal type by using the *identList* construct in the procedure heading syntax. The meaning of such a heading is not changed by such a grouping, which may thus be used as a matter of formatting convenience. For example, the two following procedure headings are identical in meaning —

```
PROCEDURE StringCompare(str1, str2 : ARRAY OF CHAR) : Order;
PROCEDURE StringCompare(str1 : ARRAY OF CHAR;
                        str2 : ARRAY OF CHAR) : Order;
```

Note that the return type specified in the heading of a function procedure syntactically forms part of the formal parameters. Thus a proper procedure without any parameters may omit the parameter list entirely, as in

```
PROCEDURE WriteLn;
```

or include an empty parameter list, as in the equivalent example

```
PROCEDURE WriteLn();
```

For function procedures however, the result type cannot be declared without a formal parameter list. Thus a function procedure without parameters must be declared with an explicitly empty parameter list —

```
PROCEDURE IsEnded() : BOOLEAN;
```

9.2 The procedure body

The procedure body consists of declarations, and a statement sequence. The declarations may be of any Modula object. There may be declarations of modules, procedures, constants, types, or variables. All such declared objects are said to be *local* to the procedure, and are visible throughout the procedure body, but not outside that scope.

If a procedure is declared local to another procedure, then within the more deeply nested procedure the local objects of the outer procedure are within scope, and are therefore visible unless *occluded* by the declaration of a more local object with the same name.

On the other hand, if a module is declared local to a procedure, the local objects of the procedure are only visible within the module if they are explicitly imported.

Any variables declared local to a procedure are said to have **automatic extent**. That is to say, such variables are created at the time of invocation of the procedure, and are destroyed at the time of procedure termination. If a procedure is invoked recursively, then in each invocation the variable name will denote a different data object.

The statement sequence of the procedure body is executed whenever the procedure is invoked. The procedure terminates when control reaches the end of the statement sequence, or whenever a `RETURN` statement is executed. It is an error for a function procedure to reach the end of the statement sequence without an explicit return.

9.3 Open array parameters

Formal open arrays are declared by the `ARRAY OF` *ElementType* construct. Invocations of procedures may substitute any array of the correct element type as the actual parameter corresponding to a formal open array parameter. (There are some special cases which are detailed in the chapter on *low-level facilities*).

Within a procedure, an open array appears as an array with an index type of some zero-based subrange of `CARDINAL`. The index ranges from zero to some upper limit determined by the cardinality of the index type of the actual parameter. This upper limit may be accessed by the built-in function `HIGH`. The array index may thus be considered to be the subrange [0 .. `HIGH`(*ident*)], where *ident* is the identifier of the formal open array.

The function procedure `HIGH` may not be used in declarations. This restriction is necessary since the high value corresponds to a the value of a hidden *size* attribute which is passed to the procedure, and is only evaluated at runtime.

Here are two examples that illustrate this correspondence.

An actual parameter of `ARRAY CHAR OF INTEGER` will appear as an open array formal of `ARRAY [0 .. 255] OF INTEGER`.

If an `ARRAY [-5 .. +5] OF INTEGER` is passed to the same open array formal, it will appear as an `ARRAY [0 .. 10] OF INTEGER`.

Within the body of a procedure, an open array formal parameter may be manipulated in almost exactly the same way as a fixed size array parameter of the same mode. In particular, such an array may be accessed element by element, and the entire variable may be passed as actual parameter to another procedure. However, the array may not otherwise be accessed as an entire variable. In particular, the following program fragment contains an error —

```
PROCEDURE Foo(VAR s : ARRAY OF CHAR);
```

```
BEGIN
  GetString(s); (* this IS legal code *)
  s := message; (* this is NOT legal  *)
```

Literal character strings

If a literal string is passed as actual parameter to a (value) open array of `CHAR` formal parameter, the `HIGH` value is one less than the length of the abstract string. Thus if the string `"example"` is passed to an open array `str`, then `HIGH(str)` will have the value 6. In this same example `str[1]` will have the value `'x'` and so on.

The character constants, and constant strings of length-1 appear with a `HIGH` value equal to zero, so that the formal parameter is of the perfectly legal type `ARRAY [0 .. 0] OF CHAR`. Note however that a *character variable* cannot be passed to an open array of characters — a character variable is not an array. The ability to send a literal character constant arises because such a constant is a special kind of literal string, the *S1* subtype of the abstract string type *SS*.

Literal strings of length equal to 0 have even stranger properties. An empty string, as an open array parameter, appears with a `HIGH` value of 0, and with first character equal to the string terminator value. This is necessary, since a `HIGH` value of -1 is impossible. Apart from this necessity, there is logic to the rule, as the first character would be the same if the empty string were assigned to a fixed size array of characters, and *that* was passed as the actual parameter.

9.4 Forward declarations

Some Modula compilers require that procedures be declared prior to any used occurence. This is called a *single pass restriction* since it commonly occurs in simple compilers that use this technology. **gpm** does not require forward declarations, but thoroughly checks the syntax and semantics of any that it encounters.

A forward declaration consists of a procedure heading, followed by the single symbol `FORWARD`. The procedure heading must have a complete formal parameter list, and the declaration must be elaborated by a complete procedure declaration within the same scope.

The elaboration of a forward declaration consists of a perfectly normal procedure declaration. The formal parameters of the procedure heading in the elaboration must match the formal parameter list in the the forward declaration in number, order, type and mode. The actual dummy names for the parameters need not match, and the grouping of parameters into identifier lists also does not affect the meaning. The following shows a forward declaration and the header of a legal elaboration.

```
PROCEDURE Centroid(ein  : CoordPair;
                   zwei : CoordPair;
                   drei : CoordPair) : CoordPair;
FORWARD;

...
```

```
PROCEDURE Centroid(pt1, pt2, pt3 : CoordPair) : CoordPair;
BEGIN
  ...
```

9.5 Procedure variables

Modula allows the declaration of **procedure types**, and the declaration of variables and formal parameters of such types. As explained in the chapter on type declarations, the rules of type compatability are somewhat different to those for other type. Assignment compatability for procedure types is based on *structural equivalence* rather than *name equivalence*.

The values that an object of procedure type may take are procedures that have matching parameter and result types. Thus the ordinary procedures that are declared in a program, or are imported into a program are the *constant values* of the structurally equivalent type. Thus *Terminal.WriteLn* is a value of the pervasive type `PROC`. However, only procedures that are declared at level one are able to be used as procedure value for either procedure variables or actual parameters passed to formal parameters of procedure types. This is called the **level-one rule**.

The level-one rule

Procedures are said to be declared at level one (or more simply, at the outer level) if the declaration is not nested within any other procedure. Note here particularly that nesting within *modules* does not count, no matter how deep any such nesting is.

The restriction is necessary, since nested procedures are able to access the local variables and parameters of their enclosing procedure(s). If such a procedure were assigned to a statically declared procedure variable, it might be activated at a time when the variables of the enclosing procedure did not exist. This would be a serious error.

9.6 Pervasive procedures

Proper procedure ABORT

The procedure `ABORT`[1] takes no parameters, and causes the immediate abnormal termination of the program. On most systems, this implies the production of a **core dump** for post-mortem analysis.

Function procedure ABS

The procedure `ABS` takes an expression of any signed numeric type as a parameter, and returns the absolute value of the parameter. The type of the return value is the same as that of the parameter.

[1]Not a standard procedure, **gpm** only

Function procedure CAP

The procedure `CAP` takes an expression of character type as a parameter and returns a character. If the parameter is a lower case character, the returned value is the corresponding upper case character, otherwise the parameter value is returned unchanged.

Function procedure CHR

The procedure `CHR` takes a value of any ordinal type as a parameter, and returns the character with the same ordinal value. It is an error if the value of the parameter is outside the character ordinal range.

Proper procedure DEC

There are two different versions of the procedure `DEC`. The simpler takes a single variable parameter of ordinal type. The procedure decrements the variable by one. The second version has a second parameter that specifies the number by which the variable is to be decremented. As before, the first parameter must be a variable of ordinal type, while the second must be an expression of unsigned numeric type, or a positive literal number.

It is an error if the result of decrementing the variable is outside the range of the type.

Proper procedure DISPOSE

The procedure `DISPOSE` deallocates dynamically allocated storage. The parameter is a variable of any pointer type. It is an error if the pointer does not point to a valid variable of the bound type of the pointer. After the execution of the procedure, the pointer will have the value *NIL*, and the bound variable will be deallocated.

This procedure is actually a denotation for the visible procedure with name `DEALLOCATE`, usually from module *Storage*. A call of `DISPOSE(x)` corresponds exactly to the expanded call `DEALLOCATE(x,SIZE(T))` where T is the bound type of the pointer designated by x.

Proper procedure EXCL

The procedure `EXCL` removes (excludes) an element in a set. The first parameter is a variable of some set type, while the second parameter must evaluate to a value of the base type of the set. If, at the call, the selected element is not in the set, the procedure has no effect.

Function procedure FLOAT

The procedure `FLOAT` returns a value of type `REAL` with value approximately equal to the value of the single actual parameter. The actual parameter may be of any numeric type, including `SHORTREAL, REAL, CARDINAL, INTEGER`, *ZZ*, or *RR*.

In **gpm** the function converts numbers to *IEEE double precision* format, and is identical in effect to `LFLOAT`.

Proper procedure HALT

The procedure `HALT` takes no parameters, and causes the immediate termination of the program. This is a *normal termination*.

Function procedure HIGH

The procedure `HIGH` takes the name of an open array formal parameter as parameter, and returns the highest index value of the corresponding actual parameter. The return type is the unsigned type. Note that this procedure is only applicable to open arrays, and not to fixed arrays.

Proper procedure INC

There are two different versions of the procedure `INC`. The simpler takes a single variable parameter of ordinal type. The procedure increments the variable by one. The second version has a second parameter that specifies the number by which the variable is to be incremented. As before, the first parameter must be a variable of ordinal type, while the second must be an expression of unsigned numeric type, or a positive literal number.

It is an error if the result of incrementing the variable is outside the range of the type.

Proper procedure INCL

The procedure `INCL` inserts (includes) an element in a set. The first parameter is a variable of some set type, while the second parameter must evaluate to a value of the base type of the set. If, at the call, the selected element is already in the set, the procedure has no effect.

Function procedure INT

This function takes a numerical expression as its parameter, and returns as integer with an implementation defined approximation to that value. **gpm** does not implement this function in this release.

Function procedure LENGTH

The procedure `LENGTH` takes an array of characters, or a literal string as its single parameter, and returns the string length. In the case of character arrays, the length is the number of characters before the first *string terminator* character, or the number of characters in the array. Thus in the following example, the function will return length 3 —

```
strVar := "abcdefgh";
strVar[3] := ""; (* string terminator *)
length := LENGTH(strVar);
```

where it is assumed that the array indexes from 0.

`LENGTH` is computed at compile time where possible, and may thus be used in declarations.

Function procedure LFLOAT

The procedure `LFLOAT` returns a value of type `REAL` with value equal to the value of the single actual parameter. The actual parameter may be of any numeric type, including `SHORTREAL, REAL, CARDINAL, INTEGER`, *ZZ*, or *RR*.

In **gpm** the function converts numbers to *IEEE double precision* format, and is identical in effect to `FLOAT`.

Function procedure MAX

The procedure `MAX` takes the name of a type as "actual parameter" and returns the greatest value of that type. In the case of subranges of the whole number types the return type is the special type *ZZ*, which is compatible with both signed and unsigned types.

Function procedure MIN

The procedure `MIN` takes the name of a type as "actual parameter" and returns the least value of that type. In the case of subranges of the whole number types the return type is the special type *ZZ*, which is compatible with both signed and unsigned types.

Proper procedure NEW

The procedure `NEW` allocates dynamically allocated storage. The parameter is a variable of any pointer type. After the execution of the procedure, the pointer will denote a newly allocated variable of the bound type of the pointer. The bound variable will be disjoint in memory from all other allocated objects.

This procedure is actually a denotation for the visible procedure with name `ALLOCATE`, usually from module *Storage*. A call of `NEW(x)` corresponds exactly to the expanded call `ALLOCATE(x,SIZE(T))` where T is the bound type of the pointer designated by x.

Function procedure ODD

The procedure `ODD` takes an actual parameter of any whole number type (including *ZZ*) and returns a Boolean value as result. The returned value is `TRUE` if and only if the value of the actual parameter is odd.

Function procedure ORD

The procedure `ORD` takes an actual parameter of any ordinal type, and returns a value of type `CARDINAL` of the same ordinal value.

Function procedure SIZE

The procedure `SIZE` exists in two forms. The first form takes a type name as an "actual parameter", and returns the storage size of objects of that type. The second form takes the designator of a (non open-array) variable, and returns the storage size of that variable[2].

In both cases, the storage size is expressed in units of `LOC`s, which in **gpm** are always `BYTE`s. The type of the returned value is always *ZZ*.

Function procedure SFLOAT

The procedure `SFLOAT` returns a value of type `SHORTREAL` with value approximately equal to the value of the single actual parameter. The actual parameter may be of any numeric type, including `SHORTREAL, REAL, CARDINAL, INTEGER`, *ZZ*, or *RR*.

[2] In the standard, the variable must be *entire*, but in **gpm** components may be selected. In this extension, the returned value is the size of the variable *if it existed*. Thus the bounds of index expressions, or the allocation status of pointer targets is not investigated

In **gpm** the function converts numbers to *IEEE single precision* format.

Function procedure TRUNC

The procedure `TRUNC` takes a parameter of any real number type, and returns a value of unsigned type. The returned value is the largest whole number that is less than or equal to the value of the parameter. It is an error if the parameter has a value that is outside the range of `CARDINAL` values.

Function procedure VAL

The procedure `VAL` provides a generic type conversion facility.

If the first "actual parameter" of the procedure is the name of an ordinal type, then the second parameter may be an expression of any ordinal type.

If the first "actual parameter" of the procedure is the name of a numeric type, then the second parameter may be an expression of any numeric type.

In either case, the returned value is of the type specified by the first parameter, and is an approximation to the value of the expression in the second. In the case of ordinal types the value is exact, but in the case of real expressions converted to whole number types the result is the integer part of the expression value. It is an error if the value of the expression may not be expressed in the range of the specified type.

Several of the other built in functions are special cases of `VAL`. In particular —

```
CHR(x)     == VAL(CHAR,x)

FLOAT(x)   == VAL(REAL,x)

INT(x)     == VAL(INTEGER,x)

LFLOAT(x)  == VAL(LONGREAL,x)

SFLOAT(x)  == VAL(SHORTREAL,x)
```

`TRUNC(x)   == VAL(CARDINAL,x)`   provided x is of some real type

`ORD(x)     == VAL(CARDINAL,x)`   provided x is of ordinal type

Chapter 10

Nested modules

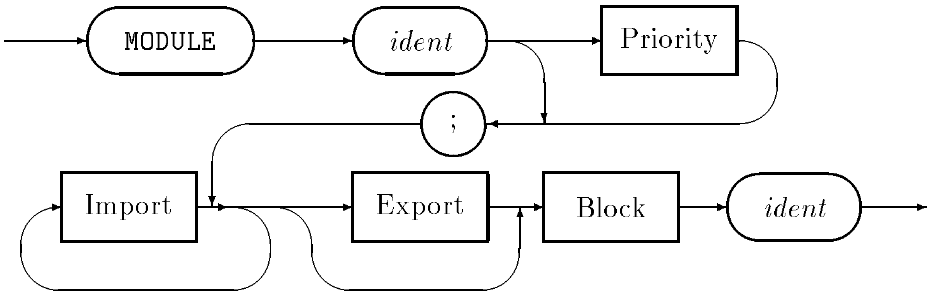
There are two mechanisms in Modula that provide for data-hiding and the modularity from which the language gets its name. Modularity may be obtained by use of the safe separate compilation features described in the section *compilation units*, or by the use of *nested modules* that are described in this section.

Modules are named and declared objects that may occur within Modula programs. Roughly speaking a nested module may be declared in any position in which a procedure declaration would be legal. Such declarations may be nested to any depth.

Syntax

ModDeclaration	$\rightarrow$	`MODULE` *ident* [**priority**] ; {**Import**} [**Export**] **Block** *ident*.
priority	$\rightarrow$	**ConstExpr**.
Import	$\rightarrow$	[`FROM` *ident*] `IMPORT` **IdentList** ;.
Export	$\rightarrow$	`EXPORT` [`QUALIFIED`] **IdentList** ;.

ModDeclaration



Just as is the case for procedures, a module declaration finishes with an `END` *identifier* sequence. As for procedure declarations, the identifier at the end must exactly match the module name.[1]

[1]Those implementations of Modula that provide interrupt priority control include an optional *priority expression* in the module heading. **gpm** does not implement these mechanisms for *UNIX* based systems, and will ignore any such priority expressions after giving a warning.

10.1 Import and export lists

A module declaration begins with a header, followed by zero or more *import lists*, followed by an optional *export list*.

Each import list is of one of the forms –

> `IMPORT` *identList*;
> `FROM` *moduleName* `IMPORT` *identList*;

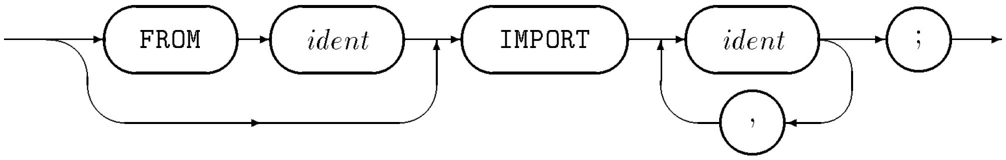
As usual, the *identList*s are just comma-separated lists of identifiers.

The export list, if present, is in one of the forms –

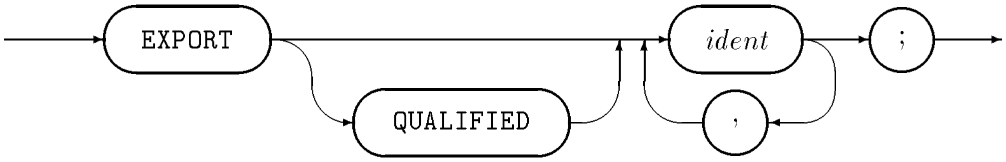
> `EXPORT` *identList*;
> `EXPORT QUALIFIED` *identList*;

The first form of export list is called *direct export*, while the second is called *qualified export*.

Import



Export



10.2 Visibility and scope rules

The declaration of a module creates a new lexical scope in which constants, types, variables, procedures and more deeply nested modules may be declared. The visibility of the objects declared in such a nested module differs from those declared in procedures, and gives the module the special properties that motivate its use. The introduction of a module acts as a boundary or frontier through which identifiers pass only by explicit authorization. An exception applies to the so called *pervasive identifiers* such as `CHAR` and `ORD`, that may be thought of as being inherited by every module scope.

Exactly the following objects are visible inside a module –

- objects declared within the module
- objects imported into the module from the immediately surrounding scope by an `IMPORT` statement
- objects inherited from the pervasive scope and not redeclared in the module

- objects injected into the scope by an export statement in a more deeply nested module

In order for an object to be imported into a module the object must be *visible* in the immediately surrounding scope. The object need not be actually declared in the surrounding scope.

The set of objects imported as a result of an import statement is called the *closure* of the identifier list in the import statement. The closure consists of exactly the identifiers named in the list, together with the enumeration constants of any enumeration types that occur on the list.

The export statement explicitly makes one or more objects that are visible inside a module also visible in the lexical scope immediately surrounding the module. If the object is exported using the `EXPORT QUALIFIED` form, the object is only visible in the surrounding scope using the qualified name *moduleName.objectName*. If the object is directly exported, then in the surrounding scope it is known by its unqualified name. (Note however that the Draft ISO standard for Modula-2 clarifies that such an exported object is *also* visible in qualified form).

The set of objects exported as a result of the export statement is called the *closure* of the identifier list in the export statement. The closure consists of exactly the identifiers named in the list, together with the enumeration constants of any enumeration types that occur on the list.

It is an error to declare a local object within a scope that has the same name as an imported object. Conversely, it is an error to export an object into a surrounding scope in which a different object with the same name is visible. Potential name clashes of this kind must be resolved by the use of qualified names.

Examples

If an object is visible unqualified outside the module then it may be imported and is visible unqualified within the module.

```
MODULE CompUnit;

  MODULE Nest1;
    EXPORT EnumType;          (* exports closure of type   *)
    TYPE EnumType = (zero, one, two);
  END Nest1;

  MODULE Nested;
    IMPORT EnumType;
    VAR thing : EnumType;     (* EnumType is visible here  *)
  BEGIN (* Nested *)
    thing := two;             (* so are the enum constants *)
  END Nested;

END CompUnit.
```

If an object is visible qualified outside the module then the qualifying module name may be imported, in which case the object is visible qualified within the module.

If an enumeration type is visible qualified, the enumeration constants are visible in qualified form also.

```
MODULE CompUnit;

  MODULE Nest1;
    EXPORT QUALIFIED EnumType;
    TYPE EnumType = (zero, one, two);
  END Nest1;

  MODULE Nest2;
    IMPORT Nest1;
    VAR thing : Nest1.EnumType;   (* visible qualified *)
  BEGIN (* Nest2 *)
    thing := Nest1.two;       (* also visible qualified *)
  END Nest2;

END CompUnit.
```

If an object is visible qualified outside the module then the object may be imported using the `FROM` *moduleName* `IMPORT` **identList** form of import, in which case the object is visible unqualified within the module.

```
MODULE CompUnit;

  MODULE Nest1;
    EXPORT QUALIFIED EnumType;
    TYPE EnumType = (zero, one, two);
  END Nest1;

  MODULE Nest2;
    FROM Nest1 IMPORT EnumType;
    VAR thing : EnumType;   (* EnumType is visible here  *)
  BEGIN (* Nest2 *)
    thing := two;           (* so are the enum constants *)
  END Nest2;

END CompUnit.
```

10.3 Module bodies

Modules, as well as containing declarations, may also contain a module body, that begins with the keyword `BEGIN` and is ended by the module end. The module body may contain initialization statements. The syntax is identical to the *Block* that appears in the syntax of procedures.

The statements of the module body are executed immediately prior to the execution of the body of the surrounding procedure or module. If the module is surrounded by another module, the body of the nested module is executed prior to the body of the surrounding module. If the module is surrounded by a procedure, the body of the module is executed as part of every invocation of the surrounding procedure. This rule implies that if modules are deeply nested, then the bodies are executed in depth first order.

Example

Consider the following program that contains nested modules.

```
MODULE Deep;
  FROM Terminal IMPORT WriteString,WriteLn;
    MODULE One; (* ---------------------------+   *)
      IMPORT WriteString; (*                  |   *)
        MODULE Two; (* ---------------------+ |   *)
          IMPORT WriteString; (*            | |   *)
            MODULE Three; (* -------------+ | |   *)
              IMPORT WriteString; (*      | | |   *)
            BEGIN (* Three *) (*          | | |   *)
              WriteString("three "); (*   | | |   *)
            END Three; (* ----------------+ | |   *)
        BEGIN (* Two *) (*                  | |   *)
          WriteString("two "); (*           | |   *)
        END Two; (* -----------------------+ |   *)
    BEGIN (* one *) (*                        |   *)
      WriteString("one "); (*                 |   *)
    END One; (* ------------------------------+   *)
BEGIN
  WriteString("zero"); WriteLn;
END Deep.
```

The output obtained by running this program is –

```
three two one zero
```

If several modules are contained within the same module or procedure, then each body is executed in order of textual appearance, followed by the body of the surrounding module or procedure.

Example

Consider the following program that contains nested modules.

```
MODULE Shallow;
  FROM Terminal IMPORT WriteString,WriteLn;
    MODULE One; (* ---------------------------+   *)
      IMPORT WriteString; (*                  |   *)
    BEGIN (* One *) (*                        |   *)
      WriteString("one "); (*                 |   *)
    END One; (* ------------------------------+   *)
```

```
    MODULE Two; (* ---------------------------+   *)
      IMPORT WriteString; (*                  |   *)
    BEGIN (* Two *) (*                        |   *)
      WriteString("two "); (*                 |   *)
    END Two; (* ------------------------------+   *)
    MODULE Three; (* -------------------------+   *)
      IMPORT WriteString; (*                  |   *)
    BEGIN (* Three *) (*                      |   *)
      WriteString("three "); (*               |   *)
    END Three; (* ----------------------------+   *)
BEGIN
  WriteString("four");
END Shallow.
```

The output obtained by running this program is –

```
one two three four
```

10.4 Dynamic modules

The nesting of modules inside procedures is unusual but interesting. Such modules are called *dynamic modules*, since the variables of such modules have automatic scope.

Any variables that are declared within the module have the same extent, and are at the same lexical level as the variables of the surrounding procedure. In the usual implementation, these variables are part of the same stack frame as the surrounding procedure. In any case, every invocation of the procedure causes storage space to be allocated to variables of the module, as well as those of the procedure. If the procedure is called recursively, then each invocation has separate instances of the module variables.

Whenever the surrounding procedure is called, memory is allocated for the data structures of the module body, and the module body is executed. This conveniently allows for the automatic initialization of the data structures of the module by the statements of the module body.

Example

Here is an example of the use of a dynamic module inside a procedure that requires a stack data structure. The module imports the element type for the stack, and exports the *push* and *pop* operations.

```
PROCEDURE UseStack();

  MODULE Stack;
    IMPORT TypeT;
    EXPORT Push, Pop;

    TYPE Index  = [0 .. 15];
    VAR  stackP : Index;
```

```
        stackD : ARRAY Index OF TypeT;

      PROCEDURE Pop(VAR e : TypeT);
      BEGIN
        DEC(stackP); e := stackD[stackP];
      END Pop;

      PROCEDURE Push(e : TypeT);
      BEGIN
        stackD[stackP] := e; INC(stackP);
      END Push;

    BEGIN
      stackP := 0; (* automatic initialization *)
    END Stack;

  BEGIN
    ...
  END UseStack;
```

Whenever the procedure *UseStack* is called, a new stack is allocated automatically, and the module body initializes the empty stack. On the completion of the procedure, the stack space is reclaimed.

Note that this example, for simplicity of exposition, has had the normal stack overflow and underflow tests removed. In the example as given, such an exception would be detected as a range check violation. Because the stack index is declared as belonging to a subrange type, the increment and decrement operations will be protected by range checking code.

Chapter 11

Compilation units

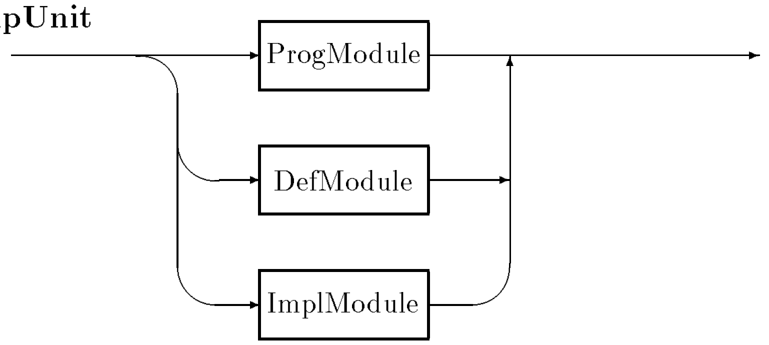
Syntax

CompUnit	→	**ProgModule** | **DefModule** | **ImplModule**.
ProgModule	→	`MODULE` *ident* [**priority**] ; {**Import**} **Block** *ident* ".".
DefModule	→	`DEFINITION MODULE` *ident* ; {**Import**} {**Definition**} `END` *ident* ".".
ImplModule	→	`IMPLEMENTATION MODULE` *ident* [**priority**] ; {**Import**} **Block** *ident* ".".

A Modula program comprises one or more compilation units. A compilation unit is the smallest unit that may be separately compiled, and should correspond to the decomposition of the program into a number of components of manageable size and complexity, separating loosely connected concepts, hiding detail, and utilising existing modules where possible.

There are three types of compilation unit: **program modules, definition modules** and **implementation modules**.

CompUnit



The simplest program consists of a single compilation unit, which must be a program module. Nevertheless, such a simple program will use facilities in standard compilation units such as the input-output library module *InOut* in order to make a useful application. The program module will be compiled and then linked with the library modules it has used to make an executable program.

More complex programs will be decomposed into a number of compilation units, of which one only must be the program module, and the others are definition module – implementation module pairs. Each definition module defines some logically-related group of facilities (constants, types, variables, procedures), making them available for use by other modules; it does not, however, supply the code that implements those procedures, and may define types in an *opaque* form. Thus the definition module provides the mechansim for separating specification from implementation, leaving the implementation hidden and protected. The corresponding implementation module supplies that hidden implementation. It completes any opaque type definitions, supplies the bodies of procedures defined in the definition module, and also contains other constants, types, variables and procedures that are relevant only to the implementation.

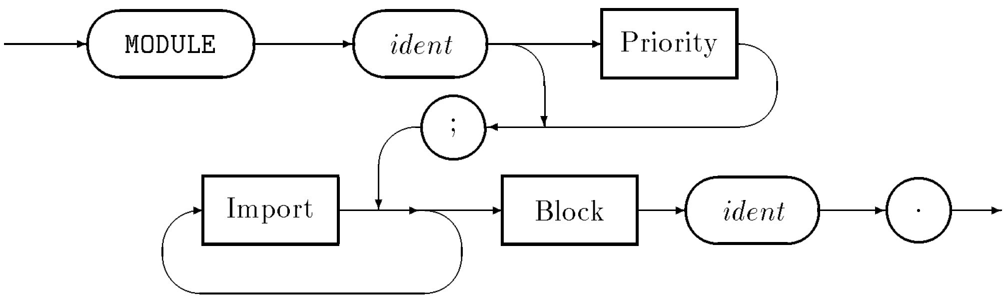
Each compilation unit is compiled separately. A definition module must be compiled before any other module that refers to objects defined in it. Subject to this constraint, program and implementation modules may be compiled in any order. The link phase then combines the object code from the program module and the implementation modules of any facilities it uses (directly or indirectly), making consistency checks, and produces an executable program.

Local (nested) modules may occur anywhere within the code bodies of program or implementation modules — they serve the same purpose of encapsulation, but for objects that presumably have no expected use outside the enclosing module. If such a nested module is, later found to have wider use, it can be extracted and split into two separate compilation units — the usual definition module – implementation module pair.

11.1 Program modules

Program modules are the ordinary modules that form the base module of every program. They may and almost always do import objects from other modules, but do not export objects. Unless deliberate renaming is done, the module name in the program module of a program becomes the name of the program.

ProgModule



11.1.1 A note on module priority

The syntax of Modula allows for program and implementation modules to have a declared **priority**. This permits mechanisms for controlling the interruptability of code to be controlled. Versions of **gpm** on the *UNIX* system do not support this feature.

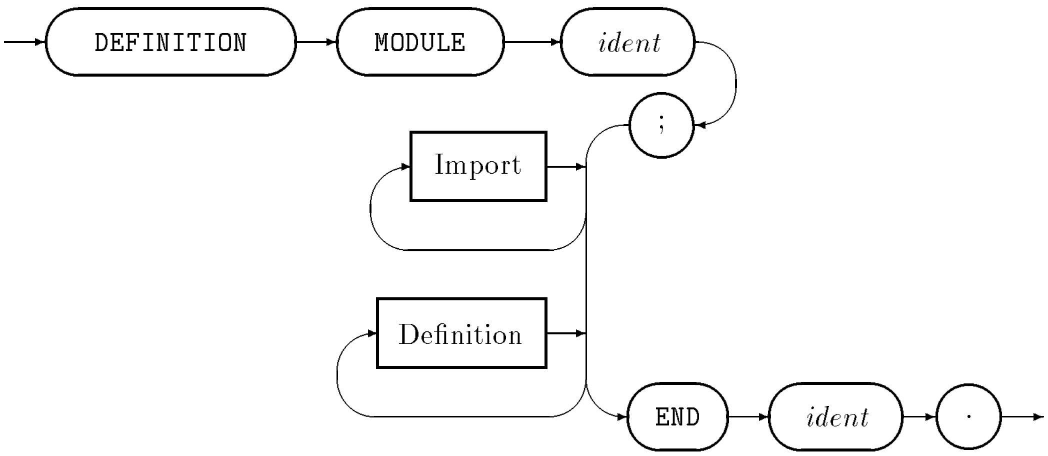
The priority is simply a constant expression of a type defined by the particular implmentation.

11.2 Definition modules

The purpose of a definition module is to define objects that may be imported and used in other modules. The definition makes available all of the imformation that the compiler needs in order to use the facilities of the module, and *hides away* all the other details.

A definition module is introduced by the reserved word `DEFINITION`, and has the syntax —

DefModule



As for any other module, the identifier following `MODULE` and that following the final `END` must be identical.

The import lists are identical to the import lists of program modules — thus a definition module may use objects defined in other definition modules. Instead of **declarations**, definition modules have **definitions**.

11.3 Definitions

Syntax

Definition	→	`CONST` { *ident* `=` **ConstExpr** ;}
	|	`TYPE` {*ident* [`=` **Type**] ;}
	|	`VAR` {**IdentList** : **Type** ;}
	|	**ProcHeading** ;.

Definitions are similar to the declarations of program modules, and of procedures, comprising constant, type, variable and procedure definitions. They differ in that

- type definitions may be *opaque*, and
- only procedure *headers* are given — that is, name, parameters and result type of function procedures

Note that no export list appears in a definition module — any identifier defined in the definition module is automatically exported in qualified mode.

If a type is defined by an opaque type declaration

```
TYPE OpaqueType;
```

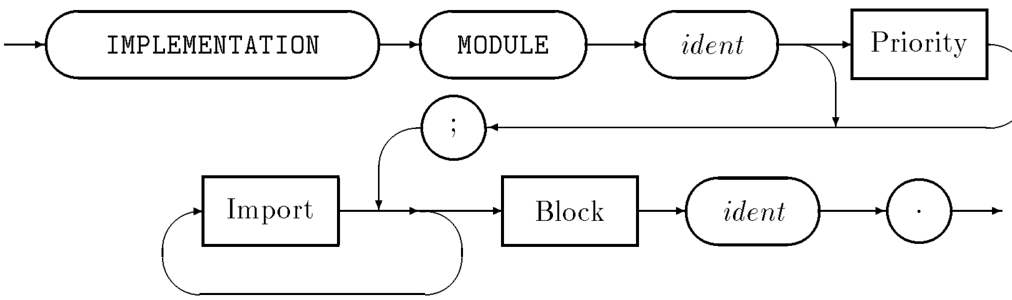
then the type must be *elaborated* in the the corresponding implementation module. Opaque types may only be elaborated as pointers or as type `ADDRESS`, but may be declared as synonyms to other opaque or pointer types.

Similarly, all procedures defined in the definition module must be elaborated in the corresponding implementation module.

11.4 Implementation modules

An implementation module is introduced by the reserved word `IMPLEMENTATION`, and has the syntax —

ImplModule



The implementation module supplies the details of objects whose definitions (interfaces) were given in the definition module of the same name — elaborations of opaque types, and the code of procedures. It may also import and/or declare other objects useful to the implementation. Such objects that are not included in the definition module are not exported, and thus are not visible to modules that import the facilities specified by the definition module. Finally, the body of the implementation module supplies initialization code that will be executed before any call to the procedures of the module.

The implementation module's own definition module is automatically imported unqualified, so that identifiers completely defined in the definition module may simply be used. Incomplete definitions are repeated as necessary: an opaque type is declared as a pointer type, and procedure headings are repeated as part of their full declaration (in the latter case, the formal parameter names need not be the same as in the definition module, though they will usually be so for readability).

11.5 Module initialization order

Before the statements of the program module are executed, module initializations are performed. The order of initialization is intended to ensure that each initialization is performed before any use of the objects being initialised can occur, but also resolves the issue of circular imports (e.g. program module A imports from module B, which imports from module C, which in turn imports from module B).

Beginning at the program module, modules are initialised in the order in which they occur in the import lists; each module initialization comprises the initialization of modules referenced by the definition module's import lists, then the initialization of modules referenced by the implementation module's import lists, then the execution of the implementation module body. This depth-first recursive initialization ignores any modules already initialised or being initialised. Thus, the example above would start from program module A, begin initialising B, find the reference to C and so begin initialising C, find the reference to B in C but ignore it as B is being initialised, complete any other initializations resulting from imports in C's definition and then implementation modules, then execute C's implementation module body, then resume initialization of B.

Chapter 12

Low-level facilities

12.1 Introduction to system modules

Modula provides a number of features which are described as being **low-level**. By this it is meant that the features deal with the underlying machine at a low level of abstraction, or that the safety features of the language are in some way being bypassed.

In general, the low-level features are exported by the pseudo-module `SYSTEM`. A pseudo-module, or synonymously a *system module*, is one which is known to the compiler. Because the compiler knows of the module, in most implementations no definition module is required, and the objects exported by the module may obey special rules and are not able to be replaced by the user. The module `SYSTEM` is the model of all other *system* modules.

The objects in system modules are known to the compiler, but are not in scope unless explicitly imported into a compilation unit. A majority of programs do not need to import `SYSTEM`, and those which do are likely to be non-portable. It is good programming practice to isolate the importation of these facilities to low-level modules which supply services to other modules of the program. Thus the non-portable features are isolated and encapsulated, and the effort in porting the program to other machines and implementations constrained.

The facilities of the module `SYSTEM` fall into a few categories. First there are the **system types** `SAL, BIN, WORD, BYTE,` and `ADDRESS`. Then there are procedures which manipulate these objects, `ADR, INCADR, DECADR, DIFADR, TSIZE` and `CAST`. Finally, there are special procedures `SHIFT` and `ROTATE`.

12.2 The system types

SAL, the smallest addressable unit

The type `SAL` is a synonym for the smallest addressable unit in the memory of the underlying machine. On most modern computers `SAL` is the same as `BYTE`, and is an 8-bit storage location.

BIN, a special binary type

The type `BIN` is a special type which is used to construct sets which map onto the underlying machine in predictable ways. In particular, any `SET OF BIN` has the property that

`SYSTEM.CAST(CARDINAL,SetType{0}) = 1`. Thus the type maps onto bit positions in the underlying implementation in such a way that bit-0 is the bit which is numerically the least significant. In the case of **gpm** `BIN` is simply a subrange `{0 .. 31}`.

WORD, the natural word size

The type `WORD` corresponds to the natural word size of the underlying machine. `WORD` is the storage size which corresponds to the types `CARDINAL` and `INTEGER`. In all versions of **gpm** this is a 32-bit word.

`WORD` is an **uninterpreted type**, that is, no meaning is attached to particular values of the type. There are no operations defined on this type. Nothing is compatible with the type, and nothing may be directly assigned to a variable of the type, except an expression also of the same type.

Formal parameters of word type

As a formal parameter type, `WORD` has very special properties. A formal word parameter, may have *any* word-sized object passed to it as actual parameter. In the case of formal parameters which are of variable mode *any* variable of word size may be used as actual parameter[1]. Thus procedures which operate on word sized quantities without regard for the semantics of the type may use this type as a formal parameter type.

As an example, a utility procedure which prints out the hexadecimal representation of a word is indifferent to the semantics of the declared type of the datum which is to be printed. Thus a formal parameter of word type would be an appropriate choice —

```
PROCEDURE HexDumpWord(wrd : SYSTEM.WORD);
 (* prints out any 32-bit sized type *)
```

Array of word formal parameters

Parameters which are open arrays of `WORD` accept any object at all as actual parameter[1]. In the case of variable mode open arrays of words, any variable may be passed as actual parameter. Unlike every other use of open arrays, the actual parameter does not have to have been declared as an array. Thus an object of any type at all may be used as an actual parameter. The hidden `HIGH` value of the actual parameter allows the called procedure to know how many word-sized locations the procedure is to manipulate.

This facility makes it possible to treat objects as uninterpreted sequences of memory locations. Here is a typical procedure —

```
PROCEDURE HexDumpBlock(blk : ARRAY OF SYSTEM.WORD);
 (* prints out SIZE(blk) 32-bit sized data *)
```

[1]Many implementations, including **gpm** require that the actual parameter is aligned at least as strictly as the word type, as a result of restrictions on some hardware platforms.

The low-level byte size

The type `BYTE` corresponds to the natural character size of the underlying machine. `BYTE` is the storage size which corresponds to the type `CHAR`, on most machines this is the same as `SAL`. In all versions of **gpm** this is an 8-bit quantity.

`BYTE` is an **uninterpreted type**, that is, no meaning is attached to particular values of the type. There are no operations defined on this type. Nothing is compatible with the type, and nothing may be directly assigned to a variable of the type, except an expression also of the same type.

Formal parameters of byte type

As a formal parameter type, `BYTE` has very special properties. A formal byte parameter, may have *any* byte-sized object passed to it as actual parameter. Thus characters, small subranges and any enumeration type object may be passed to a formal byte. In the case of formal parameters which are of variable mode *any* variable of byte size may be used as actual parameter. Thus procedures which operate on byte sized quantities without regard for the semantics of the type may use this type as a formal parameter type.

As an example, a utility procedure which prints out the hexadecimal representation of a byte is indifferent to the semantics of the declared type of the datum which is to be printed. Thus a formal parameter of byte type would be an appropriate choice —

```
PROCEDURE HexDumpByte(byt : SYSTEM.BYTE);
 (* prints out any 8-bit sized type *)
```

Array of byte formal parameters

Parameters which are open arrays of `BYTE` accept any object at all as actual parameter. In the case of variable mode open arrays of bytes, any variable may be passed as actual parameter. Unlike every other use of open arrays, the actual parameter does not have to have been declared as an array. Thus an object of any type at all may be used as an actual parameter. The hidden `HIGH` value of the actual parameter allows the called procedure to know how many byte-sized locations the procedure is to manipulate.

This facility makes it possible to treat objects as uninterpreted sequences of memory locations. Here is a typical procedure —

```
PROCEDURE HexDumpBlock(blk : ARRAY OF SYSTEM.BYTE);
 (* prints out SIZE(blk) 8-bit sized data *)
```

ADDRESS the amorphous pointer type

The type `ADDRESS` is defined as `POINTER TO SAL`. However the properties of the type are different to a user-defined type with the same definition. All pointer types are compatible with the type, so that by using type `ADDRESS` the checking of bound types of pointers is avoided. The type is said to be *amorphous* since it is without definite form, and may be made to conform to any bound type. For programs which really do need to manipulate pointers without regard to the declared bound type, this type is an important convenience.

However, any modules that use these facilities abandon the protection that Modula's type safety normally provides.

Some built-in procedures, such as NEW use this type as a formal parameter type, so that the procedure may be applied to actual parameter variables of any pointer type.

12.3 System procedures

12.3.1 ADR, the address-of operator

The function procedure ADR takes any variable as actual parameter, and returns the storage address of the object. The use of this function is a necessary part of modules which manipulate uninterpreted blocks of memory. Programs which pass addresses to coroutine libraries, or input-output system calls often require to use this function.

There are subtle dangers in the unrestricted use of this function. In particular, taking the storage address of an object allows the object to be manipulated via a *pointer alias*. This explicit alias may defeat the analysis of the compiler's code optimizer, so that optimizations become unsafe. Some compilers may simply refuse to even attempt to optimize code from modules which import this function.

12.3.2 Manipulation of addresses

On machines with so-called **flat address space**, addresses may be manipulated by ordinary arithmetic operations. Such facilities are not portable however, since in segmented architectures ordinary arithmetic is incorrect. Modula therefore supplies address manipulation procedures which capture the abstraction more portably.

The procedures supplied with **gpm** are as follows —

```
PROCEDURE INCADR(a : ADDRESS; i : CARDINAL) : ADDRESS;
(* returns the address a "incremented" by i *)
PROCEDURE DECADR(a : ADDRESS; d : CARDINAL) : ADDRESS;
(* returns the address a "decremented" by i *)
PROCEDURE DIFADR(a, b : ADDRESS) : INTEGER;
(* returns the difference (a - b) in bytes  *)
```

It is probable that these functions will change their names and/or signatures in the final ISO standard for Modula.

12.3.3 TSIZE, the type-size function

The function TSIZE is included mainly for historical reasons. It takes a type name as "parameter" and returns the storage size of the type. Its use is almost completely replaced by SIZE, which is pervasive, and more general.

In some implementations, TSIZE takes optional parameters which specify the tag types which select the active variant of the parametric type. **gpm** accepts these optional parameters, but ignores them (after giving a friendly warning).

12.3.4 Unsafe type conversions

The special function CAST provides type-casts, or unsafe type conversions. CAST takes two parameters. The first is a type name, while the second is a value of some type. The function returns the value of the second parameter but interpreted as being of the type of the first parameter. No conversion is performed, and no validity check is performed. The function is inherently unsafe, and is inherently non-portable.

Some implementations, including **gpm**, place restrictions on the objects and types which may be cast. For example, since the datum is uninterpreted and no conversion is performed the datum might need to be at least as strictly aligned in memory as the target type. In the case of objects which are too large to fit into machine registers, it may be required that the datum actually have a memory address.

Type casts are most widely used to deliberately avoid the type checking of numeric values. For example, suppose that a variable, *minus1* contains the value −1. An attempt to assign such a value to a cardinal variable will result in a runtime range violation.

```
crdVar := minus1;                (* a runtime error       *)
crdVar := VAL(CARDINAL,minus1);  (* still a runtime error *)
crdVar := CAST(CARDINAL,minus1); (* at last, not a trap   *)
```

In the final case, the value assigned to *cardVar* will be 4294967295.[2]

Type casts may be used to perform manipulations on data which do not make sense in terms of the declared semantics of the type. This is necessary, for example, in the implementation of *hash functions* where the essence of the method is to manipulate the data in a way which does not *make sense* in a conventional way.

Modula provides another method of deliberately breaking the type system when that is necessary. The second method is to declare a variant record without a tag. This method is applicable to any possible types. Here is a typical example.

```
TYPE TrickWord = RECORD
                   CASE (* oops, no tag *) : BOOLEAN OF
                   | TRUE  : fltVal : REAL; (* ieee double *)
                   | FALSE : bytes  : ARRAY [0 .. 7] OF CHAR;
                   END;
                 END;
```

In this example, a datum may be assigned to the REAL type record field, and then extracted byte by byte.

12.3.5 Bitset manipulation procedures

The draft standard specifies two procedures for operating on the BITSET type. These provide for rotation and shifting of these types. The interface as supplied by **gpm** is as follows —

```
PROCEDURE SHIFT (VAR b : BITSET, n : INTEGER);
PROCEDURE ROTATE(VAR b : BITSET; n : INTEGER);
```

[2] Or at least that is the case provided that the implementation has 32-bit binary cardinal and integer types, and uses 2's complement representation for negative quantities. Change any one of those provisos, and the value will be different.

It is possible that in future releases these proper procedures may become function procedures, so some caution is advised in their use.

The second formal parameter of each procedure is an integer, which is defined with the sense that a positive shift or rotation is to the left. Thus in the following code —

```
b := BITSET{0};
SHIFT(b,1);
```

the result is `b = BITSET{1}`. Similarly, `ROTATE(BITSET{0} = BITSET{1}`.

12.4 The Coroutines Library

The coroutines library is a system library[3]. In **gpm** it is implemented as a separate library. It is implemented in assembly language and uses the usual **gpm** *FOREIGN* mechanism. The library must be explicitly imported by user programs, in keeping with the proposals of *ISO WG-13*. However, this version implements the *old* coroutines model exactly as specified by Wirth in PIM. **gpm** will provide the new coroutines model when acceptance of the change is confirmed. In the meanwhile, the old style library will not break existing programs.

Library definition

```
(* ========================================================= *)
(* Preliminary library module for Gardens Point Modula      *)
(* ========================================================= *)

FOREIGN DEFINITION MODULE Coroutines;(* coroutines as in PIM *)

  IMPORT IMPLEMENTATION FROM "coroutines.o";
  FROM SYSTEM IMPORT ADDRESS;

  TYPE Coroutine = ADDRESS;  (* use of this type avoids SYSTEM import *)

  PROCEDURE NEWPROCESS (code  : PROC;           (* body of coroutine  *)
                       space : ADDRESS;        (* ptr to workspace   *)
                       size  : CARDINAL;       (* size of workspace  *)
                   VAR this  : Coroutine);     (* returned coroutine *)

  PROCEDURE TRANSFER (VAR thisCo : Coroutine;   (* current saved here *)
                      VAR destCo : Coroutine);  (* target to activate *)

END Coroutines.
```

[3]In PIM coroutines were in `SYSTEM`, but are now often placed in a separate library which requires separate import

12.4.1 Procedure NEWPROCESS

The procedure *NEWPROCESS* initializes a new coroutine and computes various static attributes. In particular the call of the procedure specifies the code body which the coroutine will execute, and the size and address of the workspace which it will use.

The first parameter

The first actual parameter of the procedure call must designate a parameterless procedure value, that is a *PROC*. The actual parameter must thus either be a procedure of this type, or a procedure variable of this type.[4]

The second and third parameters

The second actual parameter is of *Coroutine* or of *SYSTEM.ADDRESS* type. It is a pointer to the workspace which the coroutine will use. The memory to which this value points must have been allocated prior to the call, either by a call to *Storage.ALLOCATE* or by using a statically declared array of suitable size. It is strongly recommended that space obtained from the storage allocator be used as workspace. In particular, the workspace *must* be aligned on the most restrictive boundary which the machine architecture requires. Space obtained from *Storage.ALLOCATE* automatically fulfils this condition. The third parameter simply states the workspace size.

How much workspace does gpm require?

The imformation in this subsection is **gpm**-specific. The amount of workspace required varies with the particular implementation and on some machines may be of the order of 5k Bytes. On **gpm-pc**, at the other extreme, only about 600 bytes are required. The workspace must provide space for the coroutine state vector, usually of the order of 100 to 300 bytes, and the separate stack for open array parameters on some **gpm** implementations.

All of the remaining workspace is available as stack space for the coroutine. The procedure sets a stack overflow limit exactly 512 bytes from the end of the workspace. This safety zone provides space for cleanup procedures (which will execute in the context of the failed coroutine) to run to completion successfully.

Programs which do not require any stack space at all will thus need workspace of approximately ($SOAPSIZE + 1024$) bytes. With the system defaults this will be about 5000 bytes. For typical programs, a size of 10 000 bytes is probably more realistic. In the unusual case where a very large number of coroutines are to be created, it is recommended that the soapsize be reduced to a very small value and workspaces of as little as 1000 bytes be allocated. Remember that the size of soap is determined at *build* time, not at compile time.

Possible errors

It is an error if a coroutine (other than the main process) ends "normally", or if a coroutines runs out of stack space. In **gpm** these two errors produce the following error messages

[4]The use of parameters which are procedure variables is unusual. However, the creation of a pool of coroutines whose bodies may be specfied at runtime is an interesting technique. It allows Modula to come close to providing for *dynamically* specified processes.

respectively —

```
**** m2rts: coroutine ended without TRANSFER ****
**** m2rts: stack limit has been exceeded ****
```

A typical example

In the following program two coroutines are created.

```
MODULE CoTest;
  IMPORT SYSTEM;
  FROM Storage IMPORT ALLOCATE;
  FROM Coroutines IMPORT
       Coroutine, NEWPROCESS, TRANSFER;

  VAR  adr : SYSTEM.ADDRESS;
       init, c1, c2 : Coroutine;

    ...

BEGIN
  ALLOCATE(adr,10000); (* get workspace *)
  NEWPROCESS(Proc1,adr,10000,c1);
  ALLOCATE(adr,10000); (* get workspace *)
  NEWPROCESS(Proc2,adr,10000,c2);
    ...
  TRANSFER(init,c1);   (* init is main process *)
    ...
END CoTest.
```

12.4.2 Procedure TRANSFER

The two actual parameters to *TRANSFER* are both of *Coroutine* type, and both are of *VAR* mode. Prior to the call of *TRANSFER*, the second parameter designates the variable of *Coroutine* type which identifies the coroutine which is to be activated (or resumed, as the case may be). The first parameter, designates the variable which after the transfer identifies the coroutine which has just been suspended.

In a typical example of usage, each newly created coroutine will have a variable associated with it by the call of *NEWPROCESS*. One further variable is declared to identify the implicit main coroutine. In the previous example, *init* identifies the main coroutine, and *c1, c2* are the new coroutines.

Because of the details of the *TRANSFER* semantics, it is valid (although unusual) to use the same variable for both actual parameters. In a program with just two coroutines such a single variable can be arranged to always designate the *other* process. In this case, the call of *TRANSFER* has the meaning 'resume other coroutine'. Here is the skeleton of an example of this unusual structure.

```
  VAR  adr   : SYSTEM.ADDRESS;
       other : Coroutine;

    PROCEDURE Proc1;
    BEGIN
      ...
        TRANSFER(other,other); (* "resume" *)
      ...
    END Proc1;

BEGIN
  ALLOCATE(adr,10000); (* get workspace *)
  NEWPROCESS(Proc1,adr,10000,other);
    ...
    TRANSFER(other,other); (* "resume" *)
    ...
END CoTest.
```

Chapter 13

Differences from 'PIM'

This chapter sets out a brief list of all those changes to the language Modula which have been implemented in the current version of **gpm**. These are set out in the same sequence as the "Report on the programming language Modula-2" in Wirth's *Programming in Modula-2* (Springer-Verlag 1982, 1983, 1985, 1988).

13.1 Vocabulary and representation

Identifiers

Identifiers are now permitted to contain underscores, but with the following restrictions. An underscore may not be the first or last character in an identifier, and underscores may not be adjacent.

Numbers

Literal whole numbers are considered to be of a special, internal type called *ZZ*. These are compatible with either the signed or unsigned type. Some of the built-in functions also return this type.

Strings and character constants

Strings may be of any length, including both zero and one. Strings of length one may be treated as character strings, but are also compatible with the type *CHAR*. Strings of length zero are also compatible with the type *CHAR*, and when treated as character constants denote the string terminator. In the case of **gpm** this terminator is *Ascii.nul*.

As from this release, character constants such as *Ascii.bel* and 15C are treated uniformly with character literals such as "a". This is necessary in order to be able to concatenate single character constants.

Operators and delimiters

The symbols `~` and `NOT` are synonyms, as are `&` and `AND`. *FORWARD* is a new keyword. Details on the use of the new symbol are given later.

13.2 Declarations and scope rules

Function calls in constant expressions

Constant expressions may contain function calls to built-in functions, provided that the functions may be evaluated at compile time. Most pervasive functions may be used in this way, provided the actual parameters to the functions are themselves constants.

Constant procedures

Constants may now be declared which have (constant) procedure values. There are two clear uses for this.

In definition modules procedure constants may be used for the renaming of an imported procedure for export. In software systems which are strictly structured in layers, it allows facilities of (say) a lower layer to be passed on unchanged to a higher layer, without the inefficiency of an encapsulating procedure call.

```
DEFINITION MODULE Stacks; (* a specialization of Sequences *)
  IMPORT Sequences;
  TYPE  Stack     = Sequences.Sequence;
  CONST MakeEmpty = Sequences.MakeEmpty;
        IsEmpty   = Sequences.IsEmpty;
        Push      = Sequences.PutRight;
        Pop       = Sequences.GetRight;
END Stacks;
```

Within implementation and program modules it may be convenient to rename a procedure to avoid having to use the qualified name, or to give it a name which is more meaningful in the local context.

```
  IMPORT CardStr, IntStr, Terminal;
  CONST  IVal = IntStr.Value;
         CVal = CardStr.Value;
         NL   = Terminal.WriteLn;
  ...
```

Type declarations

Subranges

Subranges now take an optional typename which forces the subrange to take that type as *host type*. For example

```
TYPE OneToTenInt = INTEGER [1 .. 10]; (* subrange of INTEGER *)
     OneToTenCrd = [1 .. 10]; (* a subrange of CARDINAL *)
```

In the first case, variables of this type will be expression compatible with expressions of the signed type. Variables of the default subrange type would only be compatible with the unsigned type.

Records

In the syntax of records there are two minor changes. In the unusual case of variant records without *tags*, the syntax demands the presence of the colon. Thus a typical tagless variant record will now look as follows

```
CASE (* eek, no tag! *) : BOOLEAN OF
| TRUE  : ...
```

Previously, the colon would have been left out, as well as the tag field.

Secondly, variants may be empty, so that it is possible to place extra vertical bar symbols in the declaration of variants. This allows all variants to begin with the bar, as shown in the previous example.

Set types

The maximum cardinality of the base type of sets is now at least large enough to allow *SET OF CHAR*. For **gpm** the maximum cardinality is 256.

Procedure types

Procedure type declarations may contain references to types which are not yet declared. These forward declarations must be resolved within the same block in the same way as forward declarations of pointers.

This facility allows a procedure type to reference its own type in its formal parameter list, or in the case of a function procedure, to even return its own type

```
TYPE FsaStep = PROCEDURE (SymbolType) : FsaStep;
```

13.3 Expressions

Set constructors

Set constructors may now contain non-constant expressions, and *must* begin with the name of the set type. The default type *BITSET* no longer applies.

Operators

There are new operators *REM* and '/'. These apply to both the signed and unsigned type, and produce the remainder and integer part of the quotient respectively. They are the same as *MOD* and *DIV* in the case of unsigned operands.

MOD and *DIV* have their functionality either clarified or extended (depending on your point of view). Both may now take negative left operands. m *MOD* n returns the true modulus, always in the range $[0 \, .. \, (n-1)]$, while m *DIV* n returns the greatest whole number less than or equal to the rational number m/n.

Value constructors

Value constructors are proposed both for constants and variables, and native code versions of **gpm** implement both. The detailed syntax is treated in the *User Guide*. Constructors may be used for records —

```
recVar := RecType{17, Ascii.bel, "warning");
```

where the fields of the record are listed in order.

The syntax for arrays also permits repeat counts —

```
arrayVar := ArrayType{0 BY i - 1, 1, 0 BY max - i};
```

In the case of multiple dimension arrays, the names of the nested types (which are often anonymous) may be omitted —

```
CONST identity = TwoDimArray{{1.0, 0.0, 0.0},
                             {0.0, 1.0, 0.0},
                             {0.0, 0.0, 1.0}};
```

13.4 Statements

String assignments

As mentioned earlier in this section, strings of length 0 or 1 are both compatible (and hence also assignment compatible) with type *CHAR*. In the case of the empty string, the character value is the string terminator character.

Strings are now compatible with all arrays of *CHAR* of equal or greater length. The restriction that the array must index from zero has been removed.

Case statements

The syntax for case statements has been relaxed so that cases may now be empty. Essentially this permits redundant vertical bars to be inserted into the format. In particular, this modification allows a formatting style where every case begins with a bar

```
CASE expression OF
| value1 : ...
```

For statement

The lower bound expression in a for statement header may be *assignment* compatible with the for loop variable. The upper bound (if not a literal) must be expression compatible with the control variable.

The checks on threats against the for loop control variable are now very stringent indeed. Within the body of the loop the variable may not assigned to, nor sent as a *VAR* parameter to any procedure; it may not have its address taken, nor have *INC* or *DEC* applied to it. The variable must be a simple variable, it must be declared locally, and must not be imported or exported. Finally, procedures nested within the same scope as the *FOR* loop, may not threaten the variable by uplevel accesses.

13.5 Procedure declarations

Forward declarations

FORWARD is a new keyword. It anticipates a procedure declaration in order to avoid difficulties with mutual recursion when using single pass compilers. **gpm** has no such restriction, but accepts (and carefully checks) such declarations.

A forward declaration consists of the procedure heading, including formal parameters, followed by the keyword. As an example

```
PROCEDURE Thing(a, b : CHAR; VAR x : CHAR); FORWARD;
```

The forward declaration must be completed in the same block, and repeats the formal parameter list.

```
PROCEDURE Thing(lCh : CHAR; rCh : CHAR; VAR outCh : CHAR);
BEGIN
  ...
```

Note that (as in the case of definition and implementation parts) the formal parameters must match in type and order, but need not have the same names, or be declared using the same grouping lists.

Standard procedures

New standard procedures *MIN, MAX, LENGTH* have been defined, and *SIZE* (which was in *SYSTEM*) is now pervasive.

MIN and *MAX* return the minimum and maximum values respectively of their argument, which must be a typename. In the case of whole number types, or subranges of these, the functions return the type *ZZ* which is compatible with both signed and unsigned types.

LENGTH returns the length of character strings, and is evaluated at compile-time if applied to constant strings. It may thus be used in declarations

```
CONST message = "Special alert, aliens have landed";
TYPE  MesStrs = ARRAY [1 .. LENGTH(message)] OF CHAR;
```

SIZE is now pervasive, and accepts either a variable designator or a typename as parameter. It returns the storage size of the object. In the case that the parameter is a variable, the designator should not use dereference, or indexing by non-constant index expressions[1].

The functionality of the generic *value conversion* function *VAL* has been significantly extended. *VAL* now allows conversions **from** any numeric values to any ordinal or numeric type (that is, real and whole-number types, enumerations, *CHAR, BOOLEAN*, and subranges of any of these). Conversely, values of any ordinal or numeric type may be converted **to** any whole-number types (that is, *INTEGER, CARDINAL*, and subranges of either of these).

Unlike the *type cast* with which it is often confused, the use of the *value conversion* function *VAL* generates safe conversions. In most cases actual object code is generated, and a range check is performed unless **gpm** can prove that it is unnecessary to do so.

[1]Actually **gpm** does not insist on this, but returns the size that the object would have *if* it did exist. However, a program making use of this feature would not be standard conforming.

In the case of applying *VAL* to real operands, any whole number result is rounded toward zero. The following thus applies —

```
c := VAL(CARDINAL,realVal);   (* round toward zero,     *)
                              (* same as TRUNC(realVal) *)
i := VAL(INTEGER,realVal);    (* round toward zero      *)
                              (* same as INT(realVal)   *)
i := SYSTEM.ENTIER(realVal);  (* round toward -infinity *)
i := SYSTEM.ROUND(realVal);   (* rounds to nearest int  *)
```

Note carefully that the last two are non-standard. For positive values, *ENTIER* and *VAL(INTEGER,-)* return the same value.

ROUND returns an integer result. Nevertheless, implementations guarantee that the bit pattern returned by applying either *ROUND* or *INT* to any real value in the cardinal range will be correct in the sense that the returned bit pattern will correspond to the correctly transformed cardinal value. In order to make use of this property, overflow testing must be turned off. Beyond the valid range, the result is implementation dependent.

13.5.1 Parameter passing to open arrays

When actual arrays are passed to value-mode, open array formals, the element types must be *assignment compatible*. Originally, the element types had to be identical.

A consequence of this change is that values passed to open arrays must be *marshalled* in the stack frame of the calling procedure. In some cases the value must be range-checked element by element.

13.6 System-dependent facilities

Type casts

Originally Modula offered the *type-transfer-function* as a facility to bypass the compiler's type checking in low-level modules. There is general agreement that this dangerous facility should only be available as a result of an explicit import from *SYSTEM*. The procedure *SYSTEM.CAST* provides these facilities. The procedure takes a typename as its first parameter, and the object to be cast as its second. The compiler treats the result as though it were of the specified type.

From the 1995 releases, native code versions of **gpm** allow almost any value to be cast to any type.

New and changed system procedures

SIZE, which was in system has become a standard procedure, that is, become pervasive. The procedures associated with coroutines have moved to a separate module *Coroutines*. This release of **gpm** now contains this module.

There are new facilities for manipulation of the types *BITSET* and *ADDRESS*. The 32-bit sets may be shifted and rotated by the procedures *SHIFT* and *ROTATE*, while address arithmetic is provided by *ADDADR, SUBADR* and *DIFFADR*.

13.7 Compilation units

Export lists do not now appear in the definition parts of modules. It is more fruitful to consider the whole definition part file to be an extended export list.

Another consequence of this particular view, is that names are only introduced into the definition part as needed for the definitions. Imported objects which are required in the implementation part must be explicitly imported into that part.