TABLE OF CONTENTS

INTRODUCTION TO MODULA-2

The catchwords of this part are:

- EBNF-Notation
- Comparison Modula-2 - Pascal
- Modula-2 for Pascal Programmers

Chapter 2. Introduction

Up to now, you learned about the requirements of this package towards your computer system, about its features, the licensing terms, backup and implementation. Perhaps you did a little sample run.

Now, you will learn about the EBNF notation used not only throughout this manual but also in most Modula-2 books. Later, the new concepts of Modula-2 as well as the lexical and syntactic differences between Pascal and Modula-2 are pointed out. Although you probably don't know the new syntax, there are several examples of small Modula-2 programs. Because the syntactic differences to Pascal aren't that great, you shouldn't have any difficulties understanding them.

This introduction is mostly oriented towards the Pascal programmer. It is desirable for you to buy an introductory text about Modula-2. Some of the books covering the subject that are available at publishing time of this manual are listed in the bibliography appendix.

Modula-2 may be considered as a successor to Prof. Wirth's famous Pascal language. Its main differences from Pascal are

- the lack of 'built in' I/O operations leading to greater freedom about how to implement it and to a somewhat more awkward usage of I/O because you cannot write INTEGERs and STRINGs in the same Write statement. This does not affect code effectiveness. It simply means a work shift from the compiler back to the programmer, freeing the compiler from system dependency.

- an improved syntax. The very often Pascal-BEGIN..END constructs for compound statements aren't needed anymore.

- new concepts as Separate Compilation and Modules, Procedure Types, etc. The first two allow the modularization of large concepts without sacrificing the compiler's ability to type check the whole program.

Additionally, Modula-2 provides some low level facilities that make life easier for system programmers.

Most of these points will be covered in greater detail in the next subsections. First, the new concepts are introduced, followed by an overview of the changed syntax. Thereafter, a discussion of Modula-2's set of standard identifiers and standard functions as well as the reserved words follows. For an explanation of Modula-2 I/O, refer to the Standard and Utility Library Descriptions in the **Implementation Guide.**

Chapter 3. EBNF - A Way to Describe a Syntax

EBNF stands for Extended Backus Naur Form. This is a notation that provides a convenient and clear way to describe a systematic syntax. Programming languages as Modula-2, Pascal, ALGOL etc., are good samples of languages with a systematic syntax, but you can also use EBNF for instance, to describe the form of some input to your own programs. This helps you to speed up programming of these input routines. It also prevents you from overlooking problems with special cases. As another added bonus, it leads to a clear program structure.

NOTE - This is not a scientific explanation of the EBNF notation. If you already know EBNF, you can skip this chapter.

The EBNF notation can be thought of as of a language which provides some basic elements, some constructs, plus macro definitions.

As a **basic element**, you can use one of the following:

- a **literal**, i.e. a string constant enclosed in quotes (') or double quotes ("). Examples: "A", '..', ":=".

- a **symbol** that stands by itself for a string constant, whose quotes are left away just for convenience. To mark symbols clearly, it is agreed upon to write them in uppercase letters only. In Modula-2, such symbols are for example BEGIN, NOT, PROCEDURE.

The official terminology for basic elements is **terminal symbols** or just **terminals.**

There are different kinds of constructs available:

- The **selection** construct allows to offer more than one choice as a replacement for a given element. It is expressed by a vertical bar ("|"). Example: Let's assume that an answer might be either "Y" denoting Yes or "N" denoting No. So, we might write: Answer = "Y" | "N". This actually is a complete macro definition, which tells us that any occurence of the identifier Answer can be replaced by either "Y" or "N".

- **repetition** constructs allow to repeat a given sequence either zero or one time, or an arbitrary number of times including zero times.

  The zero or one time repetition is denoted by enclosing the sequence to be repeated in brackets ("[", "]"). Example: ["warm "] "water" can result in either "warm water" or just "water".

The zero or more times repetition is expressed by enclosing the items to be repeated in braces ("{", "}"). An example could be: underline = {"_"}. So, underline can be replaced for example by "_____".

- just to clarify which elements belong to which others, it is allowed to set brackets ("(", ")") around groups of symbols, constructs and elements that pertain to each other.

Macro definitions allow us, to "codename" a sequence. This sequence therefore can be replaced by its codename. A macro definition takes the form

MacroName "=" MacroSubstitution "."

An example:

Hat = "Stetson" | "Sombrero" | "Tophat" | Cap.

Here, we see that Hat can be replaced by either one of the three literals ("Stetson", "Sombrero" or "Tophat") as well as by the macro Cap, which is defined elsewhere.

The official term for such a macro definition is a **production**. A production produces the name on the left side of the equal sign. The macro name is referred to as a **nonterminal symbol** or just a **nonterminal**.

A **sequence** consists of zero or more elements and constructs.

Elements themselves can be either a basic element or a macro name.

So we see that we are back at the start: We're talking about things that we already defined. This means that this definition is **self-contained**. It does not rely on anything from the outside. To make a proper description of for example a programming language syntax, it has to be selfcontained. Otherwise, there would be no end to the description, leading eternally to productions that use other unknown nonterminals. Although this seems to be the case in some modern language stemming from France, it is definitely not the case in Modula-2.

As an example, let's express the EBNF syntax in EBNF itself:

```
Production  = Identifier "=" Expression "." .
Expression  = Term {"|" Term} .
Term        = Factor {Factor} .
Factor      = Identifier | "'" String "'" | '"' String '"' |
              "(" Expression ")" | "[" Expression "]" |
              "{" Expression "}" .
Identifier  = String .
String      = Char {Char} .
Char        = "A" | .. | "Z" | "a" | .. | "z" | "0" | .. | "9" .
```

The last production, Char, uses some imperfect EBNF shorthands: It is supposed that you yourself logically expand "A" | .. | "Z" to be one of the uppercase alphabetic letters, etc.

For a complete EBNF description of the part of the Modula-2 language implemented by this package, please refer to the appendix **Language Description.**

That's all there is to say about EBNF. Now, let's turn back to the main subject: The comparison between Pascal and Modula-2.

Chapter 4. New Concepts

In this chapter, you will learn about modules and separate compilation as well as some other noteworthy new concepts in Modula-2. With respect to our implementation, some of the concepts contained in Modula-2 on other systems aren't discussed.

Modula-2 is a general purpose language that is - among others - ideally suited for the development of large program- and programming systems; systems programming was also one of Modula-2's design goals. It achieves the necessary flexibility by extending its predecessor Pascal upwards as well as downwards.

The upward extensions are:

- Separate Compilation
- Modules
- Procedure Types

Downward extensions include:

- Relaxed Type Checking
- Low Level Machine Access

Additionally, there are several general improvements over Pascal. Among these are:

- Relaxed Declaration Order
- Constant Expression Evaluation
- Short Circuit Expression Evaluation
- Open Array Parameters

Section 1. Modules and Separate Compilation

It is difficult to talk about Modules without mentioning Separate Compilation in the same breathstroke. Therefore, the next few paragraphs will give you a basic understanding of separate compilation before the simpler to understand module concept is explained in detail.

In several Pascal dialects, there exist possibilities to split programs apart into different 'modules'. These modules are compiled as if they were complete programs just lacking the main program. You can freely use procedures declared in one such 'module' in others. All you have to do is to declare how this procedure looks like and indicate that it is 'external'. So, you provide the compiler with any and all information about that procedure. If you make an error in this external procedure declaration, well, that's it, you wanted it, you got it...

Such 'knowledge transfer' between your program parts is called **independent compilation.** The compiler totally relies on your information. It has no means to verify if you actually gave a proper definition of that procedure.

Modula-2, in contrary, distinguishes between two categories of modules: **Program modules** (which contain main programs) and **library modules.** It demands that each **library module** (one whose parts can be used by one or several other modules) be specified in two parts: A definition module, that contains the **interfaces** of all procedures, variables and types that are to be known by the **environment** of that module, and an **implementation.** This implementation is checked during its compilation with its corresponding definition for matching definitions of the **exported objects.** There may even be totally different implementations that comply to the same definition, i.e. offering the same interface. A good example to this fact is given in N. Wirth's **Programming in Modula-2,** on page 82 ff. For full type checking, it is also necessary that you have to announce from where you want an object to be imported. Here, too, the compiler checks your usage of an imported (or external) object against its definition. So, there is no way to fool the compiler for example about a procedure's parameters to allow 'fancy' type transfers.

This kind of compilation, in which the compiler is in control of the external objects, is called **separate compilation.** It is a tool that is indispensable for the successful mastery of large programming projects, i.e. projects that are not carried out by single individuals, but by groups of programmers.

1. Modules

As mentioned above, Modula-2 is well suited for large programming projects because it allows that large programs are split into possibly small units that handle a specific, well defined part of the problem. These units are called **modules.**

a) Syntax

A **module** has a **syntactic structure** that is quite similar to that of a Pascal procedure. It consists mainly of a **parameterless heading,** the **import and export lists,** followed by the **declarations of local objects** (i.e. constants, types, variables, and procedures). The end of a module builds its **body,** i.e. its statement part.

An example:

```
MODULE Demo;                          (* parameterless heading *)

  FROM AnotherModule IMPORT           (* import lists *)
    FirstObject, SecondObject;
  IMPORT MyModule;
  EXPORT MyObject;                    (* the export list *)

  CONST                               (* local object declarations *)
    MyValue = 100;

  TYPE
    Recs = RECORD
             a: INTEGER;
             x: CARDINAL;
           END;

  VAR
    MyObject: CARDINAL;
  ....

  BEGIN                               (* module body, statement part *)
    MyObject := MyValue;
  END Demo.
```

Although the syntax is procedure-like (except for the fact that import and export lists are present), there exist severe **semantic differences.** These differences get discussed in detail now.

b) Visibility and Scope Control

These differences come mostly from differing visibility rules between procedures and modules. In Modula-2 - as in Pascal - the **visibility rules for procedures** can be formulated as follows:

- Each procedure generates a new **scope.** Each identifier defined in a given scope is visible in this scope.
- Objects defined within a procedure (or scope) are invisible outside.
- Objects visible outside a procedure (or scope) are also visible inside, unless an object with the same name is defined within the procedure (or scope).

To extract the important information, this means that each identifier has to be unique in its scope.

An example program fragment:

```
VAR
  outSide, inSide, noSide: BOOLEAN;

PROCEDURE MakeNewScope;
VAR
  inSide, mySide: CHAR;

    (* inSide and mySide are visible in MakeNewScope (local objects) *)
                  (* outSide and noSide are visible in MakeNewScope
                                                and declared outside *)

BEGIN
END MakeNewScope;

              (* outSide, inSide, noSide and MakeNewScope are visible *)
                                              (* outside MakeNewScope *)
```

The **visibility rules for modules** are both more restrictive and more general. Except for the standard objects (i.e. INTEGER, NEW, ...), the **visibility of all objects is explicitly controlled by import and export lists.**

These rules are:

Export: An object defined inside a module is visible outside only if it is exported.

Import: An object visible outside a module is directly visible inside only if it is imported.

Exception: Standard Objects are implicitly visible inside a module.

Possibly you noted the word 'directly' in the import rule. This is necessary, because there are two kinds of imports: The first one is called **unqualifying import.** This is achieved with an import list of the form

```
FROM Module IMPORT
     Object, ... ;    (* provided 'Object' is exported by 'Module' *)
```

or

```
IMPORT Object, ... ; (* 'Object' is directly visible in environment *)
```

These are the normal cases. Objects imported this way can be referenced directly by their names. The second import list implies that modules can be declared local to other modules. In fact, they can even be declared local to procedures.

The other form of import is just called **import.** This is done by using an import list of the form

```
IMPORT Module;
```

In this case, you can reference any object exported by that module by using a **qualified identifier** to access it. These qualified identifiers obey to the following convention:

```
QualIdent = Ident {"." Ident} .
```

Examples of qualified identifiers:

```
MyMod.MyObject
InOut.WriteString
Tables.LookUp.Find
```

NOTE - Every element except the last identifier in such a qualified identifier has to be a module identifier. This is intuitively clear because no object but a module can export anything. The only exception is the access to an imported record variable field.

Import without unqualification serves among other things to have different objects from several modules coexist in the same module at the same time without name conflicts.

Each module has **at most one export list.** In this export list, you cannot use any qualified identifiers. This means, that you cannot export objects you imported explicitly.

The export list can take one of two forms:

```
EXPORT Object, ... ;
```

This form is called **unqualified export.** All objects exported that way can be **accessed directly by their names in the environment of the exporting module.** If the environment is another module, it can **reexport** this objects as if they were defined by itself. It also exports them indirectly by exporting the module identifier also in unqualified mode.

The second form of an export list, called **qualified export,** looks as follows:

```
EXPORT QUALIFIED Object, ... ;
```

All of the exported objects have to be imported by every other module that will use it, including the environment of the exporting module. So, even the environment knows only the module's name. The import can use either unqualification as explained above, or use the FROM clause in the import list. For separately compiled modules (see later), this is the only allowed way to export.

NOTE - Neither an import nor an export list may contain qualified identifiers.

c) Variable Existence

In Pascal- as well as in Modula-2 procedures, local variables (and all other local objects) officially start to exist on procedure entry, i.e. when the procedure's execution starts. They cease to exist as soon as the execution of the procedure they belong to ends.

Comparing this to the visibility rules mentioned earlier in this section, this leads to the conclusion that these local objects exist as long as they are visible and therefore accessible.

This leads to several problems. As an example, look at a random number generator that is formulated as a procedure.

```
MODULE UseRandom;

  VAR
    randomNumber: CARDINAL;

  PROCEDURE Random(): CARDINAL;
  CONST
    Increment = 7227;
    Range = 1717;
  BEGIN
    randomNumber := (randomNumber + Increment) MOD Range;
    RETURN randomNumber;
  END Random;


BEGIN (* main program *)
  randomNumber := 1234;
... := Random();

END UseRandom.
```

As you see, every procedure can tamper the value of the 'secret' random number. You cannot simply declare the variable inside the procedure. Although it would have a 'random' value on each call of the procedure, you may make any bet that such a random generator is not random enough just because it obeys the famous Murphy's Law ...

A typical Modula-2 approach to this random number generator problem is done as follows:

```
MODULE UseRandom;

  MODULE RandomGenerator;

    EXPORT Random;

    VAR randomNumber: CARDINAL;

    PROCEDURE Random(): CARDINAL;
    CONST Increment = 7227;
          Range = 1717;
    BEGIN
      randomNumber := (randomNumber + Increment) MOD Range;
      RETURN randomNumber;
    END Random;

  BEGIN   (* RandomGenerator *)
    randomNumber := 1234;
  END RandomGenerator;

BEGIN (* main program *)
  ... := Random();
END UseRandom.
```

This can be done that way because **module-local objects exist as long as the module's environment exist.** The **environment** is either another module or a procedure.

You certainly wonder about the use of the module body that isn't called explicitly but has to be executed to make the random number generator work correctly. This is explained in the next section.

d) Module Bodies and Initialization

Modula-2 sets the following rule:

A module's body (statement part) is executed when the module's environment begins to exist. Thus, the body is also called the module's **initialization.**

NOTE - A module's body is treated by the compiler like any other procedure except for the fact that it **can't be called explicitly** and that the compiler inserts a call to that initialization 'procedure' which ensures that it is executed just before its environment is executed.

In the above example, the actual compiled main program code of UseRandom looks as if you had written

```
BEGIN (* UseRandom *)

  CALL RandomGenerator;        (* inserted by the compiler *)

 ... := Random();              (* your code *)
END UseRandom.
```

In separately compiled modules, this initializations are handled by the linker.

e) Final Comments on Modules

The general idea behind the module concept is to enable you to package things together that belong together. For this reason, a relaxation of Pascal's stiff declaration order rule was incorporated in Modula-2. Declarations have to obey the following rules:

- Every object (i.e. constants and types) has to be declared prior to it being used in declarations.

- The above rule is relaxed for pointer declarations: A pointer referenced type can be defined later in the same scope. This is as it was in Pascal.

- Declarations can occur in any order; you can freely mix the different types of declarations (CONST, VAR, TYPE, PROCEDURE, MODULE) to group related items.

There are some more things to point out, most of them either clarifications or special cases. Please keep these in mind:

- exporting (importing) a record type makes its fields visible.

- exporting (importing) an enumeration type exports (imports) its constants, too.

- exporting (importing) a module that exports in unqualified mode, exports its objects also.

- exporting (importing) a procedure does not automatically export (import) this procedure's parameter types. You have to export (import) them separately.

NOTE - The standard objects are imported into every module. Therefore, they are also referred to as **pervasives.** This feature makes it impossible to redefine a standard identifier inside a module. In most implementations, this can be done inside procedures. In our implementation, this holds only for standard types, not for standard procedures.

The module concept itself wouldn't be flexible enough to allow efficient realization of large projects, since it does not allow to split a program in a controlled way into several modules. Therefore, another important concept will be introduced right now.

2. Separate Compilation

A Modula-2 compiler is said to accept **compilation units.** As mentioned earlier in the short overview, there are two kinds of compilation units: program modules and library modules.

Thereafter, these compilation units are also called **separate modules** or **modules.**

For modules nested within a separate module, the term **local module** will be used.

You can partition a program into separate modules **on the global level only.** It is impossible to pack a local module into a separate file and compile it this way.

The environment of a separate module is identified as the **universe** in which all separately compiled modules are embedded.

Program modules consist of a single text file; they are much like a Pascal program. When compiled, they constitute a Modula-2 program's executable main program. A program module cannot have an export list. This is evident, because there's nobody who could use the exported objects. It can have import lists, though. These serve to import objects provided by library modules. These library modules were compiled separately. Each implementation of Modula-2 provides some standard library modules as defined by Prof. Wirth in **Programming in Modula-2.** More information about this modules can be found in the **Implementation Guide.**

Some program module examples:

```
MODULE Hello;

  FROM Terminal IMPORT
     WriteLn, WriteString;

BEGIN
  WriteString('Hello,'); WriteLn;
  WriteLn;
  WriteString('       Welcome to Modula-2 on CP/M !'); WriteLn;
END Hello.
```

```
MODULE QualifiedHello;

  IMPORT Terminal;
  IMPORT InOut;

  VAR
    ch: CHAR;

BEGIN
  Terminal.WriteString('Hi there, are you a Modula-2 fan? ');
  Terminal.Read(ch);
  IF CAP(ch) = 'Y' THEN
    Terminal.WriteString('Yes');
    Terminal.WriteLn;
    Terminal.WriteLn;
    Terminal.WriteString('glad to hear it!');
    Terminal.WriteLn;
  ELSE
    InOut.WriteString('No ???');
    InOut.WriteLn;
    InOut.WriteString(' let me persuade you ...');
  END;
END QualifiedHello.
```

The first example uses unqualifying import, whereas the second demonstrates what module import and unqualification is good for.

Library modules, on the other hand, consist of two separate text files. These two parts are called the **definition module** and the **implementation module,** respectively.

The definition module serves to define all objects of a module that are to be visible from the outside, i.e. what can be imported by other compilation units. Implementation modules contain the actual code of library modules.

Definition and implementation modules have to exist in pairs that bear the same module name. They are related to each other by the compiler.

a) Definition Module Syntax

A definition module consists of a heading of the form 'DEFINITION MODULE ModuleName;', import lists, the export list, and declarations. CONST, TYPE and VAR declarations can be made as explained above; PROCEDURE declarations, however, consist only of the procedure heading, i.e. the PROCEDURE symbol, the name and the parameter list.

One extension regarding the type definition is a so called **opaque type** definition. This definition has the form

```
TYPE TypeIdent;
```

This allows to keep implementation details of specific types hidden from the user. There is one restriction to it: A hidden type is 2 bytes long. This implies that it might be implemented later as either a pointer or a standard type such as CARDINAL and INTEGER.

Neither procedures nor the module itself can have a body; furthermore, no local modules are allowed within a definition module.

NOTE - A definition module can export **in qualified mode only.** This avoids name clashes in the universe.

A sample definition module might look like:

```
DEFINITION MODULE CharIO;

  EXPORT QUALIFIED
     Write, WriteString;


  PROCEDURE Write(ch: CHAR);
  PROCEDURE WriteString(string: ARRAY OF CHAR);

END CharIO.
```

b) Implementation Module Syntax

The implementation module closely resembles a **program module.** Except for the symbol IMPLEMENTATION that precedes MODULE, it is syntactically identical. It also cannot have an export list. Two major differences exist, though: All the constants, variables and types declared in the definition module, are present in the implementation module without you having to redefine them; the procedure headings in the definition module have to be matched by procedures bearing the same name and an identical parameter list. The parameter identifiers do not have to match, but the parameter types and kinds (VAR or non-VAR) must.

The implementation module to the definition sample:

```
IMPLEMENTATION MODULE CharIO;

  FROM ASCII IMPORT
     nul;

  FROM OpSys IMPORT
     BdosFunctions, Bdos;


  PROCEDURE Write(ch: CHAR);
  VAR
    return: CARDINAL;     (* dummy return value *)
  BEGIN
    Bdos(crtOut, ORD(ch), return);
  END Write;


  PROCEDURE WriteString(string: ARRAY OF CHAR);
  VAR
    c: CARDINAL;
  BEGIN
    c := 0;
    WHILE (c <= HIGH(string)) AND (string[c] # nul) DO
      Write(string[c]);
    END;
  END WriteString;

END CharIO.
```

NOTE - Objects that were imported by the definition module aren't automatically available in the implementation module; you have to **reimport** them. In contrast to this behaviour, all objects declared within the definition module are directly available.

As in local modules, implementation module bodies serve as initializations. They are executed before the main program is started. The order of execution of these initializations is set up by the linker.

NOTE - Mutually importing library modules dictate arbitrary module initialization order. In such cases, the module's initializations cannot depend on objects imported from the other module. They generate so called **circular references.**

NOTE - Types defined in opaque mode in the definition module have to be redefined uncovering their structure in the implementation module.

Section 2. Constant Expression Evaluation

Constant expression evaluation is not really a new concept; it is done in most compilers that run on mini- or mainframe computers. On micros, C compilers and some others do already constant expression evaluation. Most Pascal compilers don't.

In Modula-2, this is defined to be a language feature. You can use this facility to parameterize your program sources better than in Pascal.

Have a look at:

```
CONST
  bufSectors = 8;          (* 8 128 byte sectors per buffer *)
  sectorSize = 128;        (* CP/M sectors are 128 bytes *)
  bufBytes = bufSectors * sectorSize;
  bufHiIndex = bufBytes - 1;
```

By changing only the constant 'bufSectors', you can accommodate different buffer sizes. In Pascal, you would have had to change 'bufSectors', 'bufBytes' and 'bufHiIndex' to achieve the same effect!

This feature is also present in statements, not only in declarations. This leads to more efficient code because everything that can be evaluated at compile time is evaluated at that time, not in runtime as in several Pascal implementations. The code thereby gets smaller and faster.

Section 3. Short Circuit Expression Evaluation

In a correct Pascal implementation, you cannot write the following IF statement without getting an error during the compilation:

```
IF (x # NIL) AND (x^.string = '....') THEN
 (* your code *)
END;
```

The error would be generated, because, if x was NIL, the second expression would make an illegal memory access. In mainframe computers, this would halt or crash the program.

In most microcomputer Pascal implementations, the compiler doesn't bother about such statements and lets you modify the memory addressed by a NIL valued pointer.

In Modula-2, the language definition states that the above IF statement is transformed to the following sequence:

```
IF x # NIL THEN
  IF x^.string = '....' THEN
    (* your code *)
  END;
END;
```

The program continues after the IF statement, as soon as the boolean expression evaluates to FALSE. If the pointer 'x' is NIL, then the second part of the condition hasn't to be evaluated at all, because the condition is certainly FALSE (the boolean expression can be true only if both parts are true, if the AND operation is used).

This rule works also for conditions that are combined by OR. In that case, the actual code is executed as soon as one of the conditions ORed together evaluates to TRUE. This is as if you had a series of IF .. THEN ELSIF .. THEN .. with the same code executed in the IF as well as the different ELSIF branches.

So, the evaluation of the expression is 'short circuited' as soon as its result can be determined. This not only avoids the abovementioned illegal memory accesses, it also accelerates a program's execution, although the programmer hasn't to care about that property.

In spite of this simplification, it is often useful to pay attention to the order in which the boolean expressions are listed. Watch for an order whose result can be determined as soon as possible. To accomplish this, set the conditions that decide the result most often in front, whenever it is feasible to do so. Correctness and security

should always take the lead over speedup attempts. In the above case, a reverse order of the conditions could be faster, but would lead to illegal memory accesses.

Section 4. Open Array Parameters

The ANSI and ISO Pascal Standards contain the so called conformant array scheme. This scheme allows to write array-handling procedures that aren't bound to a single type of array, but rather only to arrays with the same number of indices, index types, and element types. In the conformant array scheme, you can retrieve the lower and the upper bound of each dimension.

Modula-2 uses the Open Array scheme for such applications. This mechanism has some limitations compared to the above: You can have only one variable dimension, and instead of providing both the lower and the upper bound of each dimension, an index of the form **m..n** is translated to **0..n-m** and via the HIGH standard function, you can retrieve the upper bound value (m-n). You can also use the SIZE standard procedure to determine the size (in bytes) of such an open array parameter. **You can access an Open Array Parameter but element by element.** This means , an assignment like

```
string := 'string constant';
```

is **illegal** if string is an Open Array Parameter.

Examples:

```
TYPE
  Alfa = ARRAY [0..9] OF CHAR;

PROCEDURE Tabulate(name: ARRAY OF Alfa; value: ARRAY OF REAL);
VAR
  i: CARDINAL;
BEGIN
  IF HIGH(name) # HIGH(value) THEN
    WriteString('Something's rotten in the state of Denmark!');
  ELSE
    FOR i := 0 TO HIGH(name) DO
      WriteString(name[i]);
      WriteString(' = ');
      WriteReal(value,20);
    END;
  END;
END Tabulate;
```

As you see, an Open Array Parameter is always specified in the form

```
ARRAY OF Type
```

where type is any predeclared or user-defined type. Naturally, there are also **variable Open Array Parameters.**

So, whereas Pascal specifies but one parameter mode, i.e. VAR or value, Modula-2 adds a second, Open Array or 'normal' parameter.

There exists one special kind of Open Array Parameter, the ARRAY OF WORD. It is by definition the **universal parameter.** Any variable, expression or constant may be passed to it. There are some rules to remember, however:

NOTE - TSIZE(WORD) = 2. This means that an ARRAY OF WORD has twice as much elements as actual storage, i.e. you move through the array not word by word, but byte by byte! Do access the low byte of each word only! It is suggested to use a POINTER TO CHAR to handle this access. The above said leads to the conclusion that for all ARRAYs OF WORD, the equality

```
HIGH(arrayOfWord) + 1 = SIZE(arrayOfWord)
```

holds.

This seems to be strange, but that is the way this problem is handled on most byte-addressing machines like PDP-11, Motorola 6809 and 68k, the Intel 8086, 8088 family, etc. The ARRAY OF BYTE would result in more logic behaviour, but since the world spins around Prof. Wirth and not the other way around, this proposition was not enclosed in the revisions and amendments to Modula-2 which set the new standard.

Section 5. Procedure Types

In Pascal and most other programming languages, procedures are thought of as program parts exclusively. They consist of a text that specifies actions to be performed on data objects, i.e. numbers, characters, etc.

Modula-2 goes one step beyond: You can think of procedures as a special kind of constants. By introducing the associated types and variables, Modula-2 relieves the strong distinction between code and data common to most older high level programming languages. So, the procedure types and variables represent a step into the direction of **object oriented languages.**

The uses of this new concept are manyfold. The simplest ones are for instance user-specifiable error handlers in low level modules. This allows for adaptable error handling, although errors are still handled on the level where they were generated. The higher levels of a program therefore don't have to care about handling errors that occured in the lower levels. This most often simplifies the resulting code drastically. Other uses are for instance general sort procedures, to which you can specify the comparison operator by a procedure variable, etc. More advanced uses include processes and user installable device drivers.

To declare a procedure type, you have to specify the number of parameters, their types and modes (VAR or not, Open Array or not). In the case of a function procedure type, the result type has to be declared, too. In difference to normal procedure heading declarations, the **parameter list includes but the types**, and no names of the formal parameters.

An example:

```
TYPE
  TrigFunction = PROCEDURE(REAL): REAL;
```

This procedure type is compatible with the trigonometric functions as implemented in the MathLib library module, i.e. sin, cos, and arctan.

You can declare variables that are of type TrigFunction, now:

```
VAR
  func: TrigFunction;
```

Let's assume you wanted to create your own trig function table. The domain that interests you is 0.0 up to Pi/2 in different increments.

```
MODULE TrigTables;


  FROM InOut      IMPORT Write, WriteLn, WriteCard, WriteString;
  FROM RealInOut  IMPORT WriteReal;
  FROM ASCII      IMPORT ff;
  FROM MathLib    IMPORT sin, cos, arctan;


  TYPE
    TrigFunction = PROCEDURE(REAL): REAL;


  PROCEDURE PrintTable(fn: ARRAY OF CHAR; f: TrigFunction;
                       lowLimit, hiLimit, stepWidth: REAL);
  BEGIN
    WriteString(fn); WriteString(' tabulated from ');
    WriteReal(lowLimit, 0); WriteString(' up to ');
    WriteReal(hiLimit,  0); WriteString(' in steps of ');
    WriteReal(stepWidth,0); WriteLn;
    WriteLn;
    WriteString('argument           value');
    WriteString('-----------------------------------');
    WHILE lowLimit <= hiLimit DO
      WriteReal(lowLimit,18);
      WriteReal(f(lowLimit),18);    (* call procedure here *)
      WriteLn;
      IF lowLimit + stepWidth = lowLimit THEN
        WriteString('stepwidth too small');
        HALT;
      END;
      lowLimit := lowLimit + stepWidth;
    END;
  END PrintTable;

BEGIN (* TrigTables *)
  PrintTable('sine',    sin, 0.0, 3.14159 / 2.0, 0.1);
  PrintTable('cosine', cos, 0.0, 3.14159 / 2.0, 0.1);
  PrintTable('arctangent', arctan, 0.0, 0.9, 0.1);
END TrigTables.
```

There are many more applications for procedure types.

NOTE - When calling procedure variables, **specifying the parameter list is mandatory,** even if it is empty. This allows the compiler to distinguish procedure assignments from calls. For normal procedures, parameter lists have to be specified only if it is a function procedure.

WARNING - You cannot assign standard procedures like INC, DEC, etc. to procedure variables or parameters. Because they usually apply a different, faster parameter passing mechanism, they are incompatible with procedure variable parameter passing sequences. There is, however, a simple way to circumvent this restriction called **standard procedure packaging.**

You specify a procedure that in turn calls the desired standard procedure:

```
PROCEDURE Dec(VAR x: CARDINAL);
BEGIN
  DEC(x);
END Dec;
```

This applies to all standard procedures, i.e. procedures that are part of the language and not declared in a standard or utility module, or by yourself.

Section 6. Low Level Machine Access

Modula-2 provides several facilities to allow low level, machine dependent programming. These are:

- Type transfer functions that allow to circumvent the normal type compatibility rules.
- Absolute addressed variables.
- The pseudo-module SYSTEM provides data types to allow coping around with the machine at lowest level. These facilities include machine data types and functions to determine the size and location of variables.

WARNING - Please restrict the use of these facilities to single modules. Any usage of these facilities guarantees that a module is non-transportable to other processor types, eventually even to other computers using the same processor.

1. Type Transfers

Normal type identifiers may be used as **type transfer** functions. These transfers of this form, however, can occur only between types that require the same amount of storage for their representation, i.e. between INTEGERs and CARDINALs (which both use two bytes), but not between CARDINALs and CHARacters (characters occupy only one byte).

The standard type transfer functions ORD, ODD, CHR and VAL should be used instead of the above scheme whenever possible. They are more respectable than the others, because they do not imply any assumptions about the actual machine level representation of any data type.

NOTE - This implementation restricts the use of these type transfers somewhat: it is impossible to convert arrays and records into scalar types and vice versa, no matter if the sizes of the conversion argument and result type are the same or not.

Type transfer function examples:

```
MODULE TypeTransferDemo;

TYPE
  FourByteRec = RECORD
                  i: INTEGER;
                  b: BOOLEAN;
                  ch: CHAR;
                END;

  FourByteArray = ARRAY [0..3] OF BOOLEAN;

VAR
  i: INTEGER;
  c: CARDINAL;
  arr: FourByteArray;
  rec: FourByteRec;

BEGIN
  c := c * CARDINAL(i);        (* just to get a legal expression *)
  arr := FourByteArray(rec);
END TypeTransferDemo.
```

2. Absolute Variables

Another facility allows to place variables at fixed memory locations. The declaration of such a variable gives the variable's address as a cardinal constant immediately after its name in square brackets.

An example:

```
VAR
  IOByte[3H]: CHAR;      (* CP/M's I/O byte *)
  curDrv[4H]: CHAR;      (* currently selected drive as 0H..0FH *)
```

NOTE - If you are in a CP/M environment, there are several points to watch. Do not use the zero page (i.e. addresses 00H up to 100H) or addresses higher than the contents of memory cells 6 and 7 indicate on program startup, if it isn't for using CP/M facilities. The areas used by CP/M are the warm boot jump (0H..2H), the I/O byte (3H), the currently logged drive (4H), the CP/M BDOS jump (5H..7H), the two CP/M default FCB and file name areas(5CH..7DH), as well as the command line tail buffer (80H..0FFH). You can hurt your system by tampering with these variables inadvertantly.

3. The Module SYSTEM

The module **SYSTEM** is part of every Modula-2 implementation. The **system module** provides several types, functions and procedures. All of these objects allow access to the machine level, and therefore are machine dependent. They (respectively their implementation) may vary in different Modula-2 implementations.

NOTE - The system module's exported objects have special properties. Therefore, this module is known directly by the compiler. It is a so-called **pseudo-module** which indicates that it isn't a normal library module. It is built into the compiler.

Some SYSTEM procedures are compile-time functions, others are contained in the compiler's **support library**, also known as the **runtime library** MODLIB, in our implementation.

The following types are provided by SYSTEM:

TYPE **WORD**;

size is 2 bytes; it is parameter type compatible with any 2 byte sized object.

An example of WORD parameter usage:

```
PROCEDURE WriteWord(w: WORD);
BEGIN
  PutByte(output, CHR(CARDINAL(w) MOD 256));
  PutByte(output, CHR(CARDINAL(w) DIV 256));
END WriteWord;
```

This procedure can handle all 2 byte quantities; you may write INTEGERs, CARDINALs, SETs etc. to a file with this single procedure.

TYPE **ADDRESS**: POINTER TO WORD;

size is also 2 bytes; it is expression compatible with CARDINAL and assignment compatible with pointers and cardinals.

An example:

```
PROCEDURE ZeroMemory( startAddress: ADDRESS;
                      length: CARDINAL);
VAR
  i: CARDINAL;
  address: POINTER TO CHAR;
BEGIN
  address := startAddress;
  FOR i := 1 TO length DO
    address^ := 0C;
    INC(address);
  END; (* FOR *)
END ZeroMemory;
```

SYSTEM provides these function procedures:

PROCEDURE **ADR**(VAR AnyVar): ADDRESS;

This function procedure returns the address of a variable.

An Example:

```
varAddress := ADR(x);
```

You can also specify record fields, array elements or variables pointed to by a pointer.

PROCEDURE **SIZE**(AnyVar): CARDINAL;

This function procedure returns the size of a variable. SIZE has two "modes of operation: On normal, fixed size variables or parameters, it is a **compile time function**, so the size of 'AnyVar' does not vary, i.e. is a constant. If AnyVar's type is a variant record, its largest possible size is returned. On the other hand, if SIZE is applied to an Open Array Parameter, the value returned is the size of the actual parameter, which can be determined but **at run time.**

PROCEDURE **TSIZE**(AnyType, tag, tag, ...): CARDINAL;

As you can see from TSIZE's parameter list, you can use it to return the size of a specific variant of a variant record by specifying the desired tag fields. Unlike SIZE, TSIZE is always a **compile time function**, so all the tagfields have to be constants; the size is evaluated during the compilation.

Also contained in SYSTEM are

PROCEDURE **ALLOCATE**(VAR ptr: ADDRESS; size: CARDINAL);

and

PROCEDURE **DEALLOCATE**(VAR ptr: ADDRESS; size: CARDINAL);

Normally, these procedures are contained in a module called 'Storage'. For the compiler's bootstrap, they were entered into the SYSTEM module. This leads to a faster calling sequence but also to less flexibility. You cannot write your own storage management. This situation will be remedied in a future release.

Chapter 5. Differences between Modula-2 and Pascal

This chapter is devoted to the explanation of the lexical and syntactical differences between Pascal and Modula-2. To fulfill this task, it is divided into sections describing the typographic (lexical) differences, the form of declarations and of the statement parts.

NOTE - The information contained herein does not cover full Modula-2 syntax. The syntax of the part that is implemented by our implementation can be found in the **Language Description** appendix. General information is best retrieved from the standard book, **Programming in Modula-2** by Prof. Wirth (see also **Bibliography**).

Section 1. Vocabulary and Lexical Differences

There are three different topics to be covered in here: Identifiers, reserved words, symbols and standard identifiers, and comments.

1. Identifiers

In Modula-2, identifiers are case sensitive. This means the compiler differentiates between identifiers like i and I, or between TERMINAL and Terminal.

NOTE - The longer you programmed in Pascal, the more problems you will have with this rule. Your eye got acquainted to overlook different cases in identifiers and therefore, you are likely to make case mistakes. Best cure against such problems is to make yourself clear rules how to use this case sensitivity and to follow them without compromise.

2. Reserved Words, Symbols and Standard Identifiers

Modula-2 makes a difference between **reserved words** which are used to describe language constructs (i.e. BEGIN, END, WHILE,...) and **standard identifiers** that denote predeclared objects as TRUE, INTEGER, or ABS. Whereas reserved words are 'untouchable', you can redefine **standard identifiers** in procedures. So, you can create a type INTEGER with the following declaration:

```
TYPE INTEGER = CARDINAL;
```

NOTE - Although this declaration is legal, it is not recommended to use such practices.

NOTE - It is impossible to redefine any standard identifiers in modules, because these standard identifiers are imported by the compiler without you being able to intervene. Therefore, they are also referred to as **pervasives.**

WARNING - The Modula-2 System for Z80 CP/M does not allow for redefinition of standard procedure or function identifiers; an error in the code generation pass will be the result of such an attempt.

The following Pascal reserved words aren't present in Modula-2:

DOWNTO FILE FORWARD FUNCTION GOTO LABEL PACKED PROGRAM

Some new reserved words are introduced:

BY DEFINITION ELSIF EXIT EXPORT FROM IMPLEMENTATION IMPORT LOOP MODULE POINTER QUALIFIED RETURN

Symbols are used for program punctuation (';', '.', etc.) or as operators ('=', '+', '*', etc.). In addition to all of Pascal's standard symbols, three new ones are introduced:

- The vertical bar ('|') serves as a CASE variant delimiter in variant records as well as in CASE statements.

- The reserved word AND may be replaced by the shorter ampersand ('&').

- Inequality may be expressed by either '<>' as in Pascal or by '#' the unequality sign.

In the programming examples of the next chapter, these symbols will be used every now and then.

Have a look at the appendices **Reserved Words and Symbols** and **Standard Identifiers** for a complete collection of these items.

3. Comments

No curly braces ('{', '}') may be used as comment delimiters in Modula-2. The only **comment delimiters** are '(*' and '*)'.

As opposed to most Pascal Compilers, in Modula-2, nested comments are allowed. This means that you can outcomment whole sections of a program simply by setting a pair of comment delimiters at the beginning and at the end of that section regardless of any comments already present in the outcommented section.

For example, you may write

```
(*(*(*(* This comment is nested four times *)*)*) we're still in a
comment *)
```

Section 2. Declarations

One of Pascal's declaration, the Label Declaration isn't allowed in Modula-2. This enforces a more structured way of error escaping etc. It also greatly simplifies a complete implementation. The RETURN statement can be used to replace the jump to the label at the procedure end, leading to immediate return from a procedure or function.

All other declarations are similar to Pascal, although most have minor differences to their Pascal counterparts.

1. Declaration Order

Opposed to Pascal, Modula-2 declarations can be made in any order. This means that you can group related declarations together. The only rule you have to watch for is that except for pointer base types (to which they point), every item has to be declared itself before being used in a declaration.

2. Constant Declarations

> Modula-2 offers several new features regarding constants: constant expressions, typed set constants and other compatibility rules of set constants are among them. A detailed explanation of these and other items follows.

The most important difference between Pascal and Modula-2 is that the latter supports **constant expression evaluation** by its definition.

Constant expressions may contain constants only. Every constant used on the right side of a constant expression has to be declared before its usage in the expression.

Some examples:

```
CONST
  ItemSize = 469;
  BufItems = 4;
  BufferSize = ItemSize * BufItems;
  HighIndex = BufferSize - 1;

  BaseBits = {7,9};
  Mask = {0,1,2} + BaseBits;
```

NOTE - You cannot use standard- or user defined functions in the constant expressions (i.e. ABS, CAP, ...).

a) Integral Numbers

In Modula-2, there are two kinds of integral numbers: CARDINALs and INTEGERs. CARDINALs range from 0 to 65535, whereas INTEGERs can have any value between and including -32768 and 32767.

For this reason, integral constants are subdivided into three groups:

-32768 .. -1	**INTEGERs**
0 .. 32767	**INT-CARDs** (compatible with either type)
32768 .. 65535	**CARDINALs**

Modula-2 additionally provides two other **number bases** to express INT-CARD and CARDINAL constants. These additional bases are **hexadecimal** and **octal.** Their use is indicated by the letters 'H' resp. 'B' following the constant.

Valid octal digits are '0'..'7'. Octal numbers may range from 0B up to 177777B. In decimal based constants, one can use '0'..'9' as digits. As said before, they are in the range 0 up to 65535. For hexadecimal numbers, the digits 'A'..'F' are added to the decimal ones. Correct numbers are 0 to 0FFFFH.

NOTE - No lowercase letters (e.g. 'a'..'f') may be used in hexadecimal constants. You have to precede each constant beginning with a character digit by a '0' to have the compiler recognize it as a number and not as

another constant's name. Also, the base indicating letters 'B' and 'H' cannot be replaced by 'b' and 'h' respectively.

Examples: -

- Hexadecimal constants:

  0H 0FFH 200H 0FFFCH

- Octal constants:

  0B 377B 600B 177773B

NOTE - The octal base was chosen because the use of octal numbers is widespread on minicomputers as DEC's PDP series. Modula-2 was first implemented on such a mini by people accustomed to that environment. The 'B' was chosen as the base identifier because it resembles the digit '8'.

b) Characters

In addition to Pascal's usual character constants (i.e. 'a', 'K'), Modula-2 allows for direct control character constant declarations. This is achieved by using the **octal base** to specify the character's ordinal value (see Appendix **ASCII Character Set**). The base is indicated by following the constant with the letter **'C'**. The constant can have values from 0C up to 377C. This is decimal 0 to 255.

NOTE - Character constants can be declared by ordinal value but in the octal base.

Examples:

```
CONST
  ESC = 33C;
  BS  = 10C;
  BEL =  7C;
  capA = 'A';
  zed  = "z";
```

c) Strings

String constants are similar to Pascal. The only difference is that you cannot include the delimiter character in a string (i.e. ''''). Modula-2 instead provides two different delimiter characters, ' " ' and " ' ".

Examples:

'This is a correct string constant'

"Hi there, ain't this fun?"

NOTE - A string's start and end delimiters must be the same character. You cannot make a string like

'This is NOT a correct string constant".

A string constant may not contain both single and double quotes. You have to split it in that case.

NOTE - Modula-2 relaxes Pascal's string constant assignment rules by allowing you to assign any string constant to a zero based array of character variable if the constant is equal or less in length. Shorter string constants end in a 0C character that is appended by the compiler.

NOTE - String constants may not exceed a program source line. Their maximum length (as accepted by the compiler) is 128 characters.

Example:

```
VAR longString: ARRAY [0..1000] OF CHAR;

...

longString := 'This is an allowed string constant assignment.'
```

'Zero based' means that the index of the string variable has to start with 0. The Pascal rules apply to all character arrays indexed otherwise.

d) Sets

Sets differ from Pascal in several ways:

- For efficiency reasons, sets are **just 16 bits long;** they have the size of an integer or cardinal variable.

- **Set elements are** restricted to (subranges of) **constant expressions.** Variables may be included by standard procedures (INCL and EXCL).

- Set constants are delimited by **curly braces** ('{' and '}') instead of Pascal's square brackets.

- **Set constants may be typed.** The type identifier in this case precedes the set constant. In Pascal, the set type was automatically determined by its element's type.

- There is a standard type called **BITSET** that is the default set constant type. It is defined as a set with the elements 0 up to 15.

An example:

```
CONST
  bits = {0,5,7,9, 11..15};       (* this is a BITSET typed set *)
  bitset = BITSET{0..5};            (* type forced to BITSET *)

TYPE
  OurType = (firstBit, secondBit, thirdBit, fourthBit);

  OurSet = SET OF OurType;

CONST
  first = OurSet{firstBit};
```

NOTE - All type of sets can have elements with ordinal values in the range 0 to 15 at most. This means, a SET OF [100..115] is NOT possible.

e) Floating Point Numbers / REALs

Floating point constants in Modula-2 are similar to Pascal floating point constants.

The syntax of a REAL constant as accepted by the compiler is:

```
RealConstant  = Sign Number '.' [Number] ['E' Sign Number].
Sign          = ['+'|'-'].
Number        = Digit {Digit}.
Digit         = '0'|..|'9'.
```

NOTE - The **decimal point ('.')** is a required part of a REAL constant. This ensures that the compiler can detect REAL numbers in the INTEGER or CARDINAL range as such. Modula-2 does no implicit type conversions between INTEGER, CARDINAL and REAL.

Examples:

1.0 10.5 0.7689432101 1.05E10 1.1009809E-20

but NOT 1E1 14 1E1.7 .05

REAL constants may be in the range 0.0 up to $1.7014118*10^{38}$, for negative as well as positive numbers. The smallest representable floating point number is $2.94*10^{-39}$. So, the range is 2^{-128} up to, but not including, 2^{127}. The resolution of the chosen format is one part out of 2^{24}, or about 7.2 decimal digits. Consider that the difference between two representable numbers near the maximum REAL number is about maxReal/mantissa-range. The mantissa range is 24 bits or 2^{24} possible values. This means, that $2^{(127-24)} = 2^{103}$ or about 10^{31} is the difference between two adjacent representable numbers. Adding 1 to a number near MAX(REAL) doesn't make any sense, though.

WARNING - No REAL constant expressions may be specified. The compiler does not include the code to evaluate such expressions.

3. Type Declarations

Type definitions are nearby unchanged when compared to Pascal. There are new types and a different syntax for variant records and pointer definitions, however. The set type is more limited than Pascal's sets are.

There are two basically different classes of types: **structured types** and **unstructured**, or **scalar types**. The difference between the two classes is, that elements of scalar types are **atomic**, i.e. they have no sub-elements, no structure. For example, you cannot refer to the 5th bit of the CARDINAL value 13. The circumstance that the number may consist of several bits is a characteristic of its internal representation, which remains unknown. Scalar types are CHAR, BOOLEAN, CARDINAL, INTEGER, enumerations and subranges of all these types. A structured type's elements aren't atomic, they have **components**. These are record fields, and array or set elements.

The following sections will show how to declare scalar and structured types. Pointer declarations are also covered, although pointers do not fit either of the two type classes.

a) CARDINALs

Unsigned numbers in the range from 0 to 65535 are the elements of the CARDINAL standard data type. All INTEGER operations are also available for cardinals (except for ABS, of course).

You can assign cardinal and integer variables to each other; you cannot mix them in expressions. This is because the compiler obviously couldn't decide whether to use signed (INTEGER) or unsigned (CARDINAL) operations in such a mixed expression.

WARNING - An often occurring CARDINAL error is to test IF c >= 0 -- which is always TRUE! On the other hand, IF c < 0 is always FALSE.

Example:

```
PROCEDURE CardinalFun;
VAR
  c, d: CARDINAL;
  i, j: INTEGER;
BEGIN
  c := i;                          (* legal assignments *)
  i := c * d;
  d := i + j - c;                  (* ILLEGAL (mixed) expression *)
END CardinalFun;
```

b) Characters

The standard type CHAR corresponds to microcomputer Pascal implementations; the underlying character set is the ASCII alphabet, but characters may have values in the range 0C .. 377C. The eighth bit of a character can be used, though.

c) Subranges

Subrange types are declared by enclosing their range into square brackets. In Pascal, these square brackets are missing.

Example:

```
CONST
  minValue = 103;
  maxValue = 10000;

TYPE
  demoSubrange = [minValue..maxValue];
```

NOTE - The square brackets are an integral part of the subrange declaration; array index declarations are affected by this fact!

d) Pointers

Pointer declarations don't use the caret ('^') but the reserved words **POINTER TO.** You can declare a pointer's base type directly in the pointer declaration, you're no longer restricted to type identifiers as base types (see the xxPointer example).

Examples:

```
IntPointer      = POINTER TO INTEGER;
ListPointer     = POINTER TO ListElement;
TrickyPointer   = POINTER TO POINTER TO CHAR; (* C it? *)
xxPointer       = POINTER TO RECORD
                    c: CARDINAL;
                    i: INTEGER;
                  END;
```

NOTE - As in Pascal, pointer declarations may contain forward references. This is the only declaration that allows forward references.

e) Arrays

Array declarations are similar to Pascal. The only difference arises when you are using an explicitly declared subrange type as an array index. Then, the square brackets have to be omitted. If you declare the array index as usual, the square brackets are still necessary. For multidimensional arrays, there are two possible forms to specify the field indices: Aside from the standard Pascal form, i.e. "[1..2,-1..5]", you may also write "[1..2], [-1..5]" which has the same effect. This results - as you certainly guessed - because of the move of brackets to the subrange types. So, you can declare also arrays having subranges as indices (see example below!).

All one dimensional character arrays with a lower index bound of 0 are assignment compatible with string constants, i.e. you can assign a string constant to them.

NOTE - the compiler accepts string constants of 128 characters, at most. Furthermore, a string constant may not exceed a program source line.

Examples of one-dimensional arrays and string compatibles:

```
TYPE
  ArrayIndex = [0..100];
  ArrayType = ARRAY ArrayIndex OF CARDINAL;
  NormalArray = ARRAY [1..100] OF CHAR;
  StringCompatible = ARRAY [0..127] OF CHAR;
```

Examples of multi-dimensional arrays:

```
TYPE
  Rows = [0..23];
  Columns = [0..79];
  ScreenBuffer = ARRAY Rows, Columns OF CHAR;
  ChessBoard = ARRAY [0..7], [0..7] OF Position;
  Matrix = ARRAY [0..5, 0..5] OF REAL;
```

When using an array element in an expression, the type of the expression used to give the array's index or indices has to be assignment compatible to type specified in the array's declaration.

f) Records

Records differ from Pascal's records only in the variant record declaration. There are two differences:

- The CASE declaration follows the Modula-2 CASE statement convention (case delimiters aren't BEGIN END pairs but **vertical bars** ('|')).

- CASEs that have variants of the same length (just overlaid fields) may be used before the record's end. This enables you to overlay two mutually exclusive fields (see the example).

Examples:

```
TYPE
  NodePointer = POINTER TO TreeNode;     (* a binary tree *)
  TreeNode = RECORD
               ident: INTEGER;
               less,
               greater: NodePointer;
             END;

  VariantRecord = RECORD
                    i: INTEGER;
                    CASE BOOLEAN OF
                      TRUE: k, m: CARDINAL;
                    | FALSE: s: STRING;
                    END; (* CASE *)
                  END;

  ClosedVariant = RECORD
                    v: VariableKind;
                    CASE BOOLEAN OF
                      TRUE: nextObject: ObjectPointer;
                    | FALSE: IdentType: StructurePointer;
                    END; (* CASE *)
                    size: CARDINAL;
                    address: CARDINAL;
                  END;
```

NOTE - 'Fancy' type transfers as in

```
TYPE Tricky = RECORD
                CASE BOOLEAN OF
                  TRUE: c: CARDINAL;
                | FALSE: i: INTEGER;
                END; (* CASE *)
              END;
```

aren't required in Modula-2; it provides more elegant and obvious schemes to break the strict type checking system of the compiler. See in the section **Low Level Machine Access** for more details about these schemes.

g) Sets

Sets differ significantly from Pascal's sets. The main differences are:

- They are delimited by braces instead of brackets.
- Sets contain only 16 bits and occupy only one word or two bytes.
- Set inclusion and exclusion of variables can be done, but element by element by the aid of the standard functions INCL and EXCL. This was made to keep set operations efficient. Constants, however, can be included by building set constants out of several elements.

NOTE - The Modula-2 standard requires sets to be as long as one machine word. Some Modula-2 implementations (especially Volition System's) allow for longer sets. These sets aren't tolerated in a standard Modula-2 compiler or a subset thereof (as is the Modula-2 Compiler for Z80 CP/M).

There exists the standard set type **BITSET.** Its formal definition is:

```
TYPE BITSET = SET OF [0..WordSize - 1];
```

WordSize, in our case, is 16.

Sets are used to stress the memory's bits. You can use them, for instance, to clear the hi bit of a character (Hello WordStar..), to convert characters to control characters, etc. Although this requires a whole bunch of type transfers, it is the most efficent way to deal with such conditions.

The utility module **LongSets** provides for bigger sets. These sets, naturally, aren't as efficient as the normal sets.

NOTE - All type of sets can have elements with ordinal values in the range 0 to 15 at most. This means, a **SET OF [100..115] is NOT possible.** This, again, was done to keep set operations efficient.

4. Variable Declarations

Except for type declaration differences and absolute variables, which were just presented, variable declarations do not differ from Pascal.

5. Procedure Declarations

The end of a procedure declaration is marked by repeating the procedure name after the terminating END. There may be no semicolon between the END and the procedure's name. Besides augmenting the readability of longer programs, this also aids the compiler in resynchronizing after errors in the source text.

Example:

```
PROCEDURE WriteLn;
BEGIN
  ....
END WriteLn;
```

a) Formal Parameters

Formal parameters are identifiers used as 'placeholders' for actual parameters in a procedure call. Parameters can have two modes: variable and value parameters. The mode of each parameter is indicated in the formal parameter list by the presence or absence of the reserved word **VAR**.

In a procedure call, a **value parameter** can be replaced by any **expression** the type of which is assignment compatible to the formal parameter. This expression is the initial value of the parameter. In the procedure's body, value parameters can be used as if they were local variables (they are, in fact, local variables). No information may be passed back using a value parameter.

A **variable parameter**, on the other hand, are marked by the VAR reserved word and can be replaced by **variables** of the type specified in the parameter list. No 'compatible' types are allowed. Variable parameters allow passing of results to the outside of the procedure.

There is also the new Open Array Parameter construct. It has been explained earlier in this part of the manual.

b) Function Procedures

Modula-2 knows no FUNCTIONs, but only **function procedures.** There are several slight differences in the syntax:

- Pascal's reserved word FUNCTION is replaced by PROCEDURE.
- A Modula-2 function procedure has to have a (probably empty) parameter list.
- Its type is indicated as in a Pascal function by adding a colon and the result type of the function procedure.
- Instead of assigning the **result** to the function identifier, the **return value** is returned by executing a RETURN statement (see later).

Function procedure examples:

```
PROCEDURE GetChar(): CHAR;
VAR
  x: CHAR;
BEGIN
  outPointer := (outPointer + 1) MOD SIZE(CircularBuffer);
  RETURN CircularBuffer[outPointer];
END GetChar;


PROCEDURE IsOk(AccessResult: FileError): BOOLEAN;
BEGIN
  RETURN AccessResult # FatalError;
END IsOk;
```

NOTE - function procedures may return but REALs or **scalar types or subranges thereof.** Scalar types in Modula-2 are all types except arrays, records, sets and REALs.

NOTE - A function procedure call must specify a function procedure's parameter list - even if that one is empty!

Example:

```
ch := GetChar();          (* this call's accepted *)
```

but NOT

```
answer := GetChar;        (* the compiler will trap this error *)
```

c) Standard Procedures

Modula-2 has a set of predefined standard procedures. Some are **generic**, i.e. they have either a variable number of parameters (INC, DEC) or have, for instance, types as operands (VAL) or operate on many different parameter types (INCL, EXCL, INC, DEC). In short, 'generic' implies that they have more than one correct form of the parameter list.

Std Proc.	Usage
ABS(x)	function procedure; returns absolute value of 'x'. 'x' has to be an INTEGER or a REAL.
CAP(ch)	function procedure; 'ch' is uppercased, if it is 'a'..'z'. No conversion is done otherwise. 'ch', obviously, has to be a CHAR.
CHR(x)	function procedure; return the character with ordinal value 'x'. 'x' may be any scalar type (CARDINAL, INTEGER, CHAR, enumeration (incl. BOOLEAN), or subrange thereof). Only the low order byte of 'x' is taken into account, i.e. CHR(1000) is legal (but not too useful). The returned values are 0C..377C.
DEC(x)	procedure; decrement 'x' by one. Is the replacement of Pascal's PRED(x) standard function. Arguments may be of any scalar type. Watch for INTEGER underflow.

Std Proc.	Usage
DEC(x,n)	procedure; same as above, except that 'x' is decremented by 'n'. 'n' is any scalar expression.
EXCL(s,i)	procedure; exclude an element from a set. 'i' may be any scalar expression resulting in a single element.
FLOAT(c)	function procedure; returns the floating point (REAL) representation of the **CARDINAL** c. Complements the TRUNC function. There is no provision for converting an INTEGER number directly. See in the library description for such functions (MathLib).
HALT	procedure; halts a program (by doing a warm boot). Watch for non-closed output files etc. when using this procedure.
HIGH(Array)	function procedure; returns the upper index bound of an array parameter. Is usually applied to Open Array Parameters; although our implementation does not provide them, HIGH is still a legal standard procedure. No normalization to a lower bound of 0 is performed. However, there is no way to retrieve the lower bound.
INC(x)	procedure; increment 'x' by one. Counterpart of DEC above; replaces Pascal's SUCC(x) function.
INC(x,n)	procedure; increment 'x' by 'n'. Watch for INTEGER overflow.
INCL(s,i)	procedure; include 'i' in set 's'. 'i' is any expression that results in a single set element of appropriate type. Counterpart of EXCL above.
MAX(Type)	function procedure; returns the maximum value of the given scalar or subrange type. Also applies to REAL. This function is not described in **Programming in Modula-2** but is part of a revision of Modula-2 published by Prof. Wirth in the **IFI ETH Report # 59: N.Wirth: Schemes for Multiprogramming and their Implementation in**

Std Proc.	Usage
	Modula-2 / Revisions and Amendments to Modula-2. These revisions aren't fully implemented in our compiler; a later release will comply fully to those new standards.
MIN(Type)	function procedure; returns the minimum value of the given scalar or subrange type. For REALs, returns the smallest positive, non-zero REAL number. The comments given in MAX apply to MIN as well.
ODD(x)	function procedure; returns x MOD 2 # 0. 'x' may be any INTEGER, CARDINAL.
ORD(x)	function procedure; returns ordinal value of 'x'. 'x' may be any scalar.
TRUNC(r)	function procedure; returns the whole part of a REAL number as a **CARDINAL value.** REALs that get TRUNCated have to be in the range 0..65535. If these bounds are sub- or superseded, the respectively violated bound value is returned, i.e. TRUNC(-1.0) returns 0, TRUNC(1.0E10) returns 65535. No error message is given in these cases. TRUNC complements the aforementioned FLOAT function procedure.
VAL(Type,x)	function procedure; returns the element of type 'T' whose ordinal value is 'x'. 'x' is CARDINAL; 'T' may be CHAR, INTEGER, CARDINAL or an enumeration including BOOLEAN. Watch for the type's bounds as VAL doesn't do that for you.

6. Module Declaration

There is no Pascal counterpart to Modula-2 modules. Modules aren't restricted to contain a whole program file, they can as well be used to modularize a part of such a file. These modules are called **local modules.** Read also the explanation of the module concept if you aren't sure about how to use modules as tools to structure a program.

Section 3. Compatibilities

In Modula-2, there are different kind of compatibilities. The most important ones are

- **compatibility:** Generally, objects are said to be **compatible** if they refer to the same type declaration. For subranges, their base types must be compatible. Objects with an **equivalent data structure** are considered **incompatible.**

  For example, a and b in the following declarations are incompatible:

  ```
  VAR a: ARRAY[1..10] OF CARDINAL;
  VAR b: ARRAY[1..10] OF CARDINAL;
  ```

  The ADDRESS type is compatible with CARDINALs and pointers. WORD is compatible to itself only.

  For constants and procedures, these rules have to be extended.

  INT-CARD constants (0..32767) are compatible with either INTEGER or CARDINAL.

  NIL is compatible with all pointer types.

  String constants are compatible with 0-based character arrays which are at least as long as the string constant. This stands in contrast to Pascal. If the string constant is shorter than the variable it gets assigned to, at least one 0C character is appended to the string constant. So, you can check the end of a string by looking for either an 0C character or the upper index bound.

  Example:

  ```
  VAR string: ARRAY [0..5] OF CHAR;
  BEGIN
    string := '123456';     (* exact fit *)
    string := 'MOD';         (* constant shorter than variable *)
  END;
  ```

  Procedures are compatible with procedure variables with the same structure, i.e. the same parameter list and - for function procedures - the same result type.

- **expression compatibility:** The rules compiled above reflect what is called **expression compatibility.**

- **assignment compatibility:** For assignments, the operands have to be **compatible** with one exception: INTEGERs and CARDINALs and subranges thereof are considered assignment compatible, i.e. you can assign an INTEGER to a CARDINAL and vice versa. The expression's value at the time of the assignment has to be in the INT-CARD range (0..32767). This is not checked, currently.

- **parameter type compatibility: value parameters** allow for actual parameters that are **assignment compatible** with what was specified in the declaration. Actual **variable parameters** have to match the type specified in the declaration exactly.

  NOTE - WORD, ADDRESS and ARRAY OF WORD parameters are exceptions to these rules. A WORD parameter may be replaced by any actual parameter of the same size (i.e. 2 bytes). ADDRESS parameters can be replaced by either CARDINALs or POINTERs to any type. ARRAY OF WORD parameters are compatible with any kind of variable, constant or constant expression, even with single byte sized variables as for instance a character.

WARNING - You cannot pass variable expressions (i.e. 6 * i) to an ARRAY OF WORD. This is an implementation restriction.

Section 4. Expressions

The differences from Pascal concerning expressions are mostly minor. The next few lines are cited from **Programming in Modula-2** by Prof. Wirth:

> An **expression** is in general composed of several operands and operators. Its **evaluation** consists of applying the operators to the operands in a prescribed sequence, in general taking the operators **from left to right.** The operands may be constants, variables, or functions. The allowed operators depend on the operand types.

1. Operands

Expression operands are, as said above, in general constants, variables, or functions. With the exception of literal constants, i.e. numbers, character strings, and sets, operators are denoted by **designators.** A designator consists of an identifier referring to the constant, variable or procedure to be designated. It may be a qualified identifier, and may contain so called **selectors,** i.e. array indices, record fields, or up arrows. The selector - not too big a surprise - selects an element of a structured variable.

Naturally, an expression's operands have to be expression compatible.

2. Operators

Modula-2's expression syntax distinguishes the usual four classes of operators: inversion, multiplication, addition and relation operators. This is also how strength is assigned to the classes: inversion operators have greatest strength, relational operators are the weakest ones.

a) Arithmetic Operators

Symbol	Operation
+	addition
-	subtraction
*	multiplication
DIV	division
/	REAL division
MOD	modulus

There are two cases to distinguish: Floating Point (REAL) and integer numbers. In the case of integer numbers, the operands may be of type INTEGER, CARDINAL, or a subrange of either type. Both operands must be either CARDINAL (subranges) in which case the result is of type CARDINAL, or they must be both INTEGER (or subranges thereof), which leads to an INTEGER result. If REAL numbers are used, the DIV operator is not available, but the true division ('/') has been provided.

b) Locigal Operators

Symbol	Operation
OR	logical conjunction
AND, &	logical disjunction
NOT	negation

These operators apply to BOOLEAN operands and the result value is BOOLEAN, too.

c) Set Operators

Symbol	Operation
+	set union (OR)
-	set difference (a AND NOT b)
*	set intersection (AND)
/	set symmetric difference (EXOR)

All set operations take any set type as operand type and yield to the same result type. No 'mixture' of set types may be used!

In parentheses, you see the corresponding logical operations.

d) Relational Operators

Symbol	Operation
=	equal
#, <>	unequal
>	greater
<	less
>=	greater or equal (set inclusion)
<=	less or equal (set inclusion)
IN	contained in (set membership)

Equality and non-equality may be tested on all scalar types, REALs, sets and pointers.

Greater and less apply to scalar types and REAL only, i.e. to types on which an ordering relation is defined. An ordering relation allows to determine the predecessor and successor of each element.

Greater or equal and less or equal are applicable to sets, scalar types and REALs.

The membership test is defined on sets only.

NOTE - This implementation supports comparisons of character arrays for equality, inequality, less and greater **as long as they aren't Open Array Parameters.** Open Array Parameters have no compile-time definable size. This would make it very hard to find the actual maximum comparison lengths.

3. Function Operands

Function procedure calls are denoted by **following** the function's **identifier by** a (possibly empty) **parameter list.**

NOTE - Function procedure calls cannot be selected; e.g. the expressions 'func()^' or 'func()^.adr[4].name' are **illegal.**

4. Mixed Expressions

Basically, no mixed expressions are allowed. But since Modula-2 offers a less restrictive form of type transfers than Pascal does, you can use this to fool the compiler about the facts.

To allow mixed operand types in an expression, you have to use the low level type transfers. It is strongly recommended to restrict the use of these facilities as far as possible.

Section 5. Statements

Modula-2 statements differ slightly from Pascal's. The most important difference is the lack of compound statements, i.e. sequences of statements delimited by BEGIN and END. This was possible because of several little syntax changes in structured statements; they now include an explicit end statement. In Pascal, only REPEAT .. UNTIL had a terminating part (UNTIL). In Modula-2, the

1. Statement Sequences

take the place of Pascal's compound statement. A statement sequence is a series of statements separated by semicolons. They are delimited by the enclosing structure instead of Pascal's explicit BEGIN .. END.

Some examples may clarify any difficulties:

```
PROCEDURE StatementSequence;
BEGIN
  REPEAT
    ;;;;;;;;;;;;;;;;;;;
    ;;;; Statement ;;;;
    ;;;;;;;;;;;;;;;;;;;
    GetNextSymbol;
  UNTIL sym # SemiColon;
END StatementSequence;
```

NOTE - As you see, you can use semicolons to highlight some program parts. Modula-2 allows for that more liberal usage of the semicolon.

```
PROCEDURE YesOrNo(): BOOLEAN;
VAR
  answer: CHAR;
BEGIN
  LOOP
    WriteLn;
    WriteString("(Y/N): ");
    Read(answer);
    IF CAP(answer) = 'Y' THEN
      RETURN TRUE;              (* returns directly; no EXIT
necessary *)
    ELSIF CAP(answer) = 'N' THEN
      RETURN FALSE;
    END;
  END; (* LOOP *)
END YesOrNo;
```

2. Assignments

Assignments have to comply to the assignment compatibility rules to be legal.

3. Procedure Calls

Procedure calls are similar to those in Pascal. They consist of an identifier possibly followed by a parameter list enclosed in parentheses.

Empty parameter lists can be specified, but - contrary to function calls - you don't have to specify them.

An example:

```
MODULE CallProcedure;

  PROCEDURE xx;
  BEGIN END xx;

BEGIN
```

```
  xx();          (* call it with optional parameter list *)
  xx;            (* call it without *)
END CallProcedure;
```

Parameters obey the parameter type compatibility rules. Basically, value parameters are assignment compatible to what was specified in the declaration and variable parameters have to match exactly the type given in the definition. Notable exceptions are WORD parameters. These may be substituted by any two byte scalar type, i.e. CARDINAL, INTEGER, enumerations of more than 256 elements and subranges of all of these types, as well as by sets.

4. IF Statement

The IF statement's syntax has changed slightly compared to Pascal. It now includes a closing END symbol as well as the ELSIF variant to save nesting levels. Its general form is now

```
IF <condition1> THEN
  <statement sequence>
ELSIF <condition2> THEN
  <statement sequence>
...

ELSE
  <statement sequence>
END;
```

Naturally, the ELSIF clauses as well as the ELSE clause are optional. The number of ELSIF's isn't limited.

An example:

```
IF programStop THEN
  CloseFiles;
  HALT;
ELSIF errors THEN
  ErrorHandling;
ELSE
  DoItAgain;
END;
```

5. CASE Statement

The CASE statement's syntax has also changed a little bit:

- there is an optional ELSE clause.
- the variants are separated by the semicolon and the vertical bar ('|') respectively the ELSE symbol.
- case variants can be constant expressions or even constant ranges.

WARNING - If you use constant expressions and ranges, be careful to avoid double specifications of the same case variant value!

Still, only scalar typed variables may be used as CASE selectors, i.e. as the variable whose values determine what to do in the case statement.

An example:

```
CASE i OF
  -10   : WriteCard(int, 5);
| -5..5 : WriteString(' ok, in Range. ');
| 70    : WriteCard(error, 3);
ELSE
  WriteString('unrecognized command...');
END;
```

NOTE - If a CASE statement that has no ELSE clause is entered in runtime with a value that isn't a case variant, a runtime error occurs. The program is halted with an appropriate message. This can be avoided by specifying an empty ELSE clause.

WARNING - There may be 256 CASE variants, at most. In ranges, each element of a range counts as a single variant[1]. This is an implementation restriction.

[1] the term 'label' may be used in the place of 'variant'.

6. WHILE Statement

Like in the IF statement, a terminating END symbol has been added to the WHILE statement. This is the only difference to Pascal's WHILE.

An example:

```
PROCEDURE FindElement(k: CARDINAL; VAR element: Element);
VAR
  locElement: Element;
BEGIN
  locElement := elementRoot;

  WHILE (locElement # NIL) AND (locElement^.key < k) DO
    locElement := locElement^.next;
  END; (* WHILE *)

  IF (locElement = NIL) OR (locElement^.key # k) THEN
    element := NIL;
  ELSE
    element := locElement;
  END;
END FindElement;
```

7. REPEAT Statement

This statement doesn't have any differences to Pascal.

8. FOR Statement

The FOR statement now includes a terminating END symbol. Instead of the DOWNTO reserved word, a more liberal BY clause has been added to it. The argument of the BY clause has to be an INTEGER or CARDINAL typed constant expression; no variable step width is allowed.

The FOR control variable has to be of scalar type. It is suggested to use local variables as FOR control variables, but you are free to use any variable - including record fields, imported variables, etc as control variable.

The FOR loop bounds are calculated upon entry into the statement. The control variable can be changed by the program, but it is a good idea to treat it as a read-only item inside the FOR loop. There is no anonymous FOR loop variable, in other words.

The FOR loop is executed until the end bound is surpassed. If it is surpassed before the loop got executed, it won't be executed at all.

WARNING - The only restrictions concerning FOR-statements are: they may not use MAX(INTEGER) resp. MAX(CARDINAL) as upper bound values, because of the way in which the bounds are tested in runtime would lead to eternal loops in these cases; the same goes for downcounting enumerations to the first element (ORD(element) = 0). This an implementation restriction.

Some examples:

```
FOR i := 1 TO 4 DO
  a[i,50] := 100;
END; (* FOR *)


k := 100;                    (* this loop is never executed at all *)
FOR i := 150 TO k BY 2 * 5 DO
  number := number DIV 10;
END; (* FOR *)
```

9. LOOP and EXIT Statements

These two statements are newly introduced. They form the most flexible **LOOP** construct, solving some of Pascal's deficiencies: you can program endless loops without having to resort to "REPEAT UNTIL HellFreezesOver". The **EXIT** statement completes the construct by offering a possibility to terminate it. LOOPs containing multiple EXIT statements serve to build loops with multiple end conditions that cannot be formed into a single condition as required by Pascal's looping constructs.

The EXIT statement can stand anywhere in the LOOP; upon its execution, control is passed to the first statement after the LOOP.

NOTE - The LOOP statement can be used to simulate all other loop constructs. Albeit, it is recommended NOT to use it that way. Use WHILE, REPEAT and FOR whenever possible; resort to LOOP only if it is really unavoidable to do so.

Examples:

```
LOOP
  GetCommand;
  ExecuteCommand;
  WriteString('Done. Your Next Command: ');
END;
```

```
LOOP
  DisplayItems;
  WriteString("What's up now? ");
  Read(ch);
  IF ch = 'Q' THEN EXIT;
  ELSIF ch = 'N' THEN CreateNewItem;
  ELSIF ch = 'D' THEN DeleteItem;
  END;
END;
```

NOTE - It is also possible to leave a LOOP statement by executing a RETURN statement in a (function) procedure. This means that the RETURN statement is also operable from within a LOOP statement.

10. WITH Statement

WITH statements include a terminating END symbol. In Pascal, you can give multiple variable references in one WITH statement. In Modula-2, a separate WITH statement for each variable is necessary.

WARNING - Also after the base variable of a WITH statement is changed inside it, the item referred by the WITH statement is still the same. So, these two examples have different effects:

```
WITH a^ DO                      WHILE a^.next # NIL DO
  WHILE next # NIL DO             WITH a^ DO
    a := next;                      a := next;
  END; (* WHILE *)                END; (* WITH *)
END; (* WITH *)                  END; (* WHILE *)
```

While the first example could loop forever (the condition 'next # NIL' is a loop-invariant), the second one is able to find the actual end of that chain.

Example:

```
NEW(bar);
WITH bar^ DO
  name := 'moonshine club';
  GetAddress(address);
  next := barRoot;
  barRoot := bar;
END; (* WITH *)
```

11. RETURN Statement

The return statement has two forms: either the reserved word **RETURN** alone, or **RETURN** followed by an expression. In both cases, it indicates a procedure's termination (same for module bodies).

The expression specifies a function procedure's **return value.** Its type must be assignment compatible with the function's declared result type. Therefore, function procedures demand the presence of at least one RETURN statement. Although only one return statement is executed per call, several may be present. In Pascal, it can be simulated by an assignment to the function's name and an immediately following GOTO to the function's end.

In proper procedures, a return statement is used to terminate the procedure before the end of its body is reached. It is some sort of emergency exit. In Pascal, it is can be simulated by using a GOTO to the end of the procedure's body. There is no equivalent to Pascal's unstructured GOTO that allows to leave and hop over blocks.

Examples:

```
PROCEDURE GetChar(): CHAR;
VAR
  ch: CHAR;
BEGIN
  ch := buf[i];
  INC(i);
  RETURN ch;
END GetChar;

PROCEDURE Emergency;
BEGIN
  LOOP
    GetCommand;
    IF emergency THEN RETURN;
    InterpretCommand;
  END;
END Emergency;
```