FTL Modula-2

Fast Interactive Modula-2 Compiler

Language Reference

Published exclusively in Europe by HiSoft

First printing August 1987

ISBN 0 948517 06 9

Set using an Apple Macintosh™ and Laserwriter™ with Aldus Pagemaker™.

Table of Contents

Language Reference

1 Determining how you should approach this Manual 3

2 An Introductory Example 7

3 The Lexical Items 9

3.1 Marks 10
3.2 Identifiers and Reserved Words 11
3.3 Literals 13
3.4 Comments 16

4 The Module Structure 17

4.1 IMPORT and EXPORT 19
4.1.1 Nested Modules 21
4.1.2 EXPORT 23
4.1.3 Some Advanced Rules 24
4.2 CONST Declarations 24
4.3 TYPE Declararions 25
4.3.1 The Predefined Types 26
4.3.2 Type Equivalence 28
4.3.3 Enumerations 28
4.3.4 Subrange Types 29
4.3.5 Set Types 31
4.3.6 Array Types 33
4.3.7 Record Types 34
4.3.8 Pointer Types 37
4.3.9 Procedure Types 39
4.3.10 Private Types 39
4.4 Type Compatibility 41
4.5 Variable Declarations 43

4.6 Procedure Definitions 46
4.6.1 Open Array Parameters 49
4.6.2 Procedure Variables 50
4.6.3 FORWARD Declarations 53
4.7 The Statements Part 53

5 Expressions 55

5.1 Entire Variables 55
5.2 Constants 56
5.3 Function Calls 56
5.4 The Set Builder 57
5.5 Operators 60
5.5.1 Arithmetic Operators 61
5.5.2 Boolean Operators 62
5.5.3 Set Operators 62
5.5.4 Relational Operators 63

6 Statements 65

6.1 Procedure Calls 66
6.1.1 The RETURN Statement 66
6.1.2 Standard Procedures 67
6.2 Assignment Statement 68
6.3 Conditional Statements 69
6.3.1 The IF Statement 69
6.3.2 The CASE Statement 71
6.4 Looping Statements 72
6.4.1 The WHILE Statement 72
6.4.2 The REPEAT Statement 72
6.4.3 The FOR Statement 73
6.4.4 The LOOP Statement (and the EXIT Statement) 75
6.4.5 The WITH Statement 77

7 Syntax 81

8 Extensions and Restrictions 87

9 Compiler Error Messages 91

10 Linker Error Messages 109

11 Compiler Limits 113

12 A Comparison Of Modula-2 and Pascal 115

12.1 Lexical Differences 115
12.2 Declaration 116
12.3 Expressions 117
12.4 Statements 117
12.5 Separate Compilation 119

13 Revisions and Amendments 121

This page is intentionally left blank
(excepting, of course, this message)

FTL Modula-2 Language Reference

General

We cannot guarantee the correctness of the compiler. In a product of this size, and despite considerable efforts by several people, it is unlikely that there are no bugs. It is your responsibility to ensure that any programs you produce with the compiler work as expected. If you do find a bug, please tell us about it. We will attempt to fix all bugs reported but we cannot guarantee to do this within a specific period of time (usually, though, bugs are fixed within a couple of days of being reported, although distribution can consume several weeks). If a bug makes it difficult for you to continue with a project, please make this clear when you report the bug and we will try to expedite the repair.

Please report all bugs (and suggested improvements) in writing, even if you have telephoned initially, since otherwise it is easy for the bug notification to become lost and, in any case, phone conversations often lead to a lack of precision in the definition of the problem.

The programs, relocatables and symbol files produced by you with the compiler are your property. There is no need to inform us of such programs, or to pay a licence fee. However, you may not distribute any part of this system in source.

All supplied modules, and the supplied utility programs may be freely distributed as linked programs.

1 Determining how you should approach this Manual

This is the reference for **FTL Modula-2**. It describes the Modula-2 language as implemented in the compiler.

This manual is common to all versions of the compiler while there are separate user manuals for each implementation. This manual describes the language. The user manual describes how you convert your programs in the language into executable programs. The user manual also describes the standard modules supplied with the particular implementation, since they vary slightly from machine to machine so as to make good use of the underlying hardware and operating system.

If you already know Pascal, or some other modern language, such as Algol 68, Simula 67, or perhaps even C, then you should have little difficulty learning Modula-2 from this manual. If you have no knowledge of programming, or have only programmed in Basic or COBOL, then there are a number of concepts that are probably not sufficiently described in this manual to meet your needs.

In any case, you will probably find it useful to purchase a text book on Modula-2. In recent months, a number of books on Modula-2 have appeared. While none of the books which we have seen is perfect, there are several which are useful. Check at your nearest technical bookshop. The Addison-Wesley Publishing Company has several titles, at least two of which the texts by Edward J Joyce and Professor Arthur Sale are worth examining.

Other books which you may find useful are books on data structures which give examples in a real (or almost real) programming language (such as Niklaus Wirth's 'Data Structures + Algorithms = Programs'), and books which teach programming Pascal (You can find a shelf full in any technical bookshop). In the latter case, look for a book that emphasizes programming technique rather than language details.

Pascal programmers should be aware that the similarity between Modula-2 and Pascal has been somewhat exaggerated in the popular press. Modula-2 is a much more powerful language than Pascal. It has been designed to support software engineering, whereas Pascal was mainly intended to be didactic.
For example, Modula-2 provides excellent separate compilation facilities, and allows you to break free of the strong typing constraints of Pascal when it is essential to do so. See **Section 12** for a comparison of Pascal and Modula-2.

C programmers need to be aware that the emphasis of Modula-2 is rather different from that of C. C is intended to be a high level assembler. Like an assembler, it does little checking. As a result, it will rarely pick up blunders at compile time (though UNIX users have the excellent program `LINT` to do this for them). Modula-2, on the other hand, attempts to find as many errors as possible at compile time, but gives you facilities to override this checking.

If you are already an experienced Pascal programmer, the appendix showing some of the principle differences between the languages is a quick way into Modula-2. You will also want to look at the sections on Modules, since these have no counterpart in Pascal. Also, you will probably find a copy of Wirth's book 'Programming in Modula-2' (Springer-Verlag) useful. This is very definitely a book for experienced programmers. Make sure you get the latest edition (edition 3 at the time of writing this manual), since the language keeps changing from edition to edition. We are attempting to keep the compiler up to date with this book. In addition, there are a certain number of programs in source on the distribution disks. These will provide extended examples of modules written in Modula-2. They should not, however, be assumed to be paragons of good programming style.

In the manual, we will want to distinguish between the Modula-2 language and this implementation of Modula-2. When we refer to *Modula-2*, we mean the language. When we refer to *the compiler*, we mean this particular implementation. Occasionally, we will refer to the *standard language*. This means the language as defined in the report section of Niklaus Wirth's book. Unfortunately, this report is not sufficiently complete to serve as a full language standard. Many things which you are left to deduce would need to be made explicit in a standard.

In addition, there is a set of changes and clarifications released by Niklaus Wirth between February and March 1984. These are implemented by this compiler. They are listed in Part II of the manual. As a result of these changes, some modules which were valid under the old compiler are no longer valid. The biggest difference is the removal of the EXPORT statement from definition modules.

2 An Introductory Example

We shall start with a very small example of a Modula-2 program so that the material which follows is easier to follow.

```
 1 MODULE SumNumbers;
 2 FROM SmallIO IMPORT ReadInt,WriteInt;
 3 FROM Terminal
 4     IMPORT WriteString,WriteLn;
 5 VAR i,j:INTEGER;
 6 BEGIN
 7     WriteString('Enter some numbers');
 8     WriteLn;
 9     WriteString('Enter 0 to end');
10     WriteLn;
11     i:=0;
12     LOOP
13          WriteString('Number:');
14          ReadInt(j);
15          IF j=0 THEN EXIT END;
16          i:=i+j;
17          END;
18     WriteString(' The sum is ');
19     WriteInt(i,7);
20     WriteLn;
21     END SumNumbers.
```

This program simply outputs a message to the terminal, reads in a list of numbers, and then prints their total. We have numbered the lines of the code for ease of reference in the text that follows. These numbers are not part of the program; if you were to type them in as part of the program, it would not compile.

Note the form of the program. It takes the form of a module. The module is the basic unit of compilation in Modula-2 you cannot compile anything smaller. A module starts with a module header (line 1), and ends with an END followed by the repetition of the name of the module and then a period. (line 21)

A typical program contains a number of modules. To produce an executable program, you must bind together these modules with the

linker.
In this module, references are made to routines in the modules `SmallIO` and `Terminal`. The routines are used throughout the code, but they are requested right at the beginning of the code, on lines 2 through 4. These import statements are a central feature of Modula-2; whenever you want to access an object that exists in another module, you must tell the compiler what module it is in and give its name.

When you link this module to produce an executable program, the other modules which it needs will be included automatically.

Line 5 of the program contains some variable declarations. In Modula-2, all variables must be declared. You cannot simply *declare by use*, as in Basic or Fortran. This helps protect you against mis-spelling variable names.

Line 6 through 20 of the module form the executable part. Execution starts with line 6. When line 20 is reached, the program is terminated.

The rest of this part of the manual describes the language in detail.

3 The Lexical Items

Programs in the language contain a series of symbols called lexical items. Lexical items are the basic words and punctuation marks of a program.

Each lexical item consists of one or more characters. There can be no blanks or tabs within a lexical item (except within string literals), nor can there be a line break (line breaks are not allowed in strings, either).

For example, the first few lexical items in the example in the preceding section are `MODULE`, `SumNumbers`, `;` and `FROM`.

Line breaks have absolutely no significance in Modula-2 programs. (If you put one in the middle of a lexical item, the text is no longer a Modula-2 program).

Lexical items fall into three groups. Firstly, there are the punctuation symbols, or marks, such as `;` and `:` in the example above. Marks can contain more than one character. For example, `:=` is one mark, not two.

Secondly, there are the identifiers and the literals. These items have a value as well as being lexical items. For example, `Fred` and `George` are both identifiers, even though they are different. Similarly, `42` is a numeric literal as well as representing the number 42. That is, these symbols have meaning in addition to the type of lexical item that they represent.

Finally, there are the reserved words. Reserved words look like identifiers, but each reserved word is a different lexical item, and it has no meaning other than that it represents a particular lexical item. For example, `MODULE` and `FROM` in the preceding example, are reserved words.

Reserved words are like highway signs. They tell the compiler what to expect for the next few symbols.

3.1 Marks

The lexical constants are:

`+ - * /`	The usual arithmetic operators. Note that / can only be used for real number division.
`:=`	(pronounced 'becomes') The assignment separator.
`&`	An alternative logical AND operator.
`=`	Test for equality operator (Note: **never** used for assignment). It is also used in `CONST` and `TYPE` declarations.
`# <>`	Two alternate forms for the not equal operator. We always use <> since this is the same as Pascal, and in any case, # isn't really # , it's really national currency symbol 2. If you are in England, it will print as a pound sign (£) , in Japan as a yen sign (¥), etc.
`< > <= >=`	The usual relational operators.
`( )`	Parenthesis for use in expressions.
`[ ]`	Index brackets for use on arrays, and in some other places, such as subrange definitions.
`{ }`	Set braces. Pascal programmers note: Not used for comments.
`(* *)`	Comment braces. Comments may be nested. This makes it possible to comment out code which contains comments. This is different from Pascal which does not allow nested comments.
`^`	The dereference operator. This operator is used with pointers.
`, . : ..`	Various punctuation marks.

| Case variant terminator. This is not the exclamation mark (!). Its usually somewhere up the top right hand side of your keyboard. (The IBM PC, naturally, has it down the bottom left). There is sometimes a gap in the middle of the bar.

; Statement separator. You only need a semi-colon when the next symbol is the start of a statement. For example, you would not need one before and `END` or an `ELSE` since these symbols occur in the middle of a statement. Even so, including a semi- colon in these places does no harm and can save compilation errors occurring when you add more statements.

~ The alternate form of the boolean `NOT` operator. Avoid this character as it often fails to print on dot matrix printers with old ribbons, because it usually only uses two pins on the print head.

3.2 Identifiers and Reserved Words

An identifier is a symbol which you can use to name an object. A reserved word is an identifier that has been reserved by the language to represent (part of) some language construct.

The first character of an identifier must be alphabetic. This may be followed by any number of alphanumeric characters.

The alphabetic characters may be upper or lower case. However, **case is important**: the identifier `FIDO` is not the same as the identifier `Fido`, or the identifier `fido`. These are three different identifiers!

It is conventional to spell identifiers in Modula-2 in mixed case with the first character of each word in the identifier in upper case and the remaining characters in lower case. In addition, it is convenient to use all lower case for local or unimportant variables.

Examples:

```
LineLength  LionsInAfrica  CupWinnersCup
```

This makes it easier to get the spelling correct.

In this compiler, only the first thirty-two characters of an identifier are significant. Characters after the thirty-second are ignored. Also, in this compiler, you can use the characters $ and _ (dollar and underline) in an identifier. These characters are significant. They cannot start an identifier.

For example, `Foo$1`, `Foo_1` and `Fool` are all different identifiers. The use of $ and _ in identifiers makes your programs non-portable.

Reserved words look like identifiers, but they are not simply special identifiers. To the compiler, a reserved word is more like a mark (such as [) than it is like an identifier. You should keep this in mind when you are fixing compilation errors.

Reserved words are always completely upper case. Here is a complete list of reserved words:

```
AND             ARRAY           BEGIN
BY              CASE            CONST
DEFINITION      DIV             DO
ELSE            ELSIF           END
EXIT            EXPORT          FOR
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
```

In addition to the reserved words, there are some identifiers called standard identifiers. These are simply identifiers which are defined in the compiler you can reference them without first declaring them.

There is a difference in the way this compiler handles standard identifiers and the way they are handled in the standard language. The standard states that these identifiers are implicitly imported at the global level of any module (even modules nested inside other modules), and cannot therefore be redeclared at the module level.

In this compiler, they are declared at a level one below the level of the outermost module.

They can therefore be redeclared at the module level. We recommend that you do not do so to retain compatibility with other Modula-2 compilers. In any case, using the standard identifiers for other purposes is confusing.

The standard identifiers are:

```
ABS       BITSET    BOOLEAN   CAP       CARDINAL  CHAR
CHR       DEC       DISPOSE   EXCL      FALSE     FLOAT
HALT      HIGH      INC       INCL      INTEGER   MAX
MIN       NEW       NIL       ODD       ORD       PROC
REAL      SIZE      TRUE      TRUNC     VAL
```

In the MSDOS and 68000 versions, the additional standard identifiers `LONGINT`, `LONGCARD`, `SHORT` and `LONG` are also defined.

All these standard identifiers are described in detail later in the manual. (See the index for page numbers.)

3.3 Literals

Literals are either numeric literals or string literals.

A numeric literal is a string of digits. It may be in octal, if followed by `b` or `B`, or in hex if followed by `h` or `H`. It may not contain commas, or any other non-numeric (except, of course, for hexadecimal literals).

For example:

```
1 23 0ffffH 777B
```

These last two are a hexadecimal and an octal constant respectively.

Note that the hex value `FFFFh` had to be introduced with a zero so that it could be distinguished from an identifier.

Real constants may contain a period and optionally an exponent. For example: `2.3 3.7E-9 2.3E9 5E4`

The last example (`5E4`) is an extension permitted by this compiler. The `E` notation is used to give an *exponent*, which is a power of ten. Hence, `5E4` is the number `50000`.

String literals can either be sequences of characters enclosed in either single or double quotes:

```
'Barrossa Valley'
"Cunningham's Gap"
```

Note that a string opened with a single quote is terminated with a single quote and cannot contain a single quote. A string opened with a double quote is terminated with a double quote and cannot contain a double quote. (Pascal's `''''` convention is not supported.)

A string literal may contain zero characters. If it contains exactly one character, it may be used as either a character constant (i.e. of type `CHAR`) or a string constant. A string has type `ARRAY[0..n] OF CHAR`, where `n` is the number of elements in the string, less `1`.

This compiler contains an extension which allows you to include non-printing characters in strings. For compatibility with other compilers, this extension is normally disabled. To enable it, use the pseudo-comment:

```
(*$A^*)
```

A pseudo-comment is a comment that actually changes the way the compiler works. A pseudo-comment always has a `$` character immediately following the opening `(*` and this is followed in turn by a character identifying the pseudo-comment. Other pseudo- comments are defined in the user guide in the section on compiler flags.

Here, the character `^` in `(*$A^*)` is an escape character which is to be used to prefix control characters. It can be any non-alphanumeric character.

If the compiler finds this escape character in a string, and the following the character is alphabetic, the two characters together represent the corresponding control character. For example:

```
'Line 1^M^JLine 2^G^G^G'
```

Which when listed to the terminal, types out as two lines of output and rings the bell three times. This is because `^M` represents the carriage return character (`0dx`), `^J` represents the line feed character (`0ax`) while `^G` represents the character to ring the bell (`07x`).

If the following character is not alphabetic, the next character represents itself. This allows you to include the escape character in a string (by repeating it) and even allows you to include quote characters in a string: `'^'An^' ^^Example^^'`

which prints as: `'An' ^Example^`

To disable the use of the extension, enter the pseudo-comment `(*$A *)`. That is, setting the flag character to space turns the facility off. Do not enter `(*$A*)` without the space. This sets the escape character to be * and the comment is not yet terminated!

Character literals are string literals which contain exactly one character. In addition, a character literal can be entered by giving the value of the character in either octal or hexadecimal, followed by the character `C` or `X`.

For example: `13C 0dx`

Note that the second form (`0dx`) is an extension supported by this compiler for hex character values.

If the suffix is a `c`, (or `C`) the ordinal value is in octal. If it is an `x`(or `X`), it is hexadecimal. It is not possible to give the value in decimal!

Hence: `0dx`, `0Dx`, `0DX`, `15c` and `15C` all represent the ASCII character *carriage return*.

3.4 Comments

Comments are enclosed in the symbols `(*` and `*)`.

Comments can be nested. This is useful in commenting out code which itself contains comments. For example:

```
(* The following code is commented out

WHILE (i<Tabcnt) AND (Tab[i]<>Key) DO
      (*keep searching*)
      INC(i);
      END;
*)
```

In Pascal, the `*)` after `searching` would terminate the comment and the next two lines would be compiled. In Modula-2, because comments nest, the lines following `(*keep searching*)` are still commented out.

The { and } characters of Pascal cannot be used for comments. They are used in set constructs.

4 The Module Structure

The basic unit of compilation in Modula-2 is the Module. Any program you write will consist of a number of modules. You will write some of these modules yourself. Others will be modules which have been provided with the compiler. The latter modules are described briefly in the user guide. In addition, the sources for these modules is on the release disks.

To create a program, you compile all of your modules individually and then call the linker to combine them into a program. When you make a change to a module, you will often only need to recompile that particular module. You then call the linker again to combine all the modules together.

With a few exceptions which will be noted on the `README.NOW` file on your distribution disks, all the supplied modules have been compiled ready for use when you receive the compiler. You do not need to recompile them to use them in your own programs.

There are three types of module. The type of module is identified by the first reserved word or words in the text of the module:

```
MODULE
DEFINITION MODULE
IMPLEMENTATION MODULE
```

A (plain) module is a piece of code which may use other modules in its definition, but is never used by another module. Hence, this is the top level of a program and most programs will only contain one such module (though they may contain more and need not have any).

Definition and implementation modules go together in pairs.

The definition module acts as a prefix to the implementation module. It defines the objects of the module which can be accessed from outside the module.

Any variables, constants or types defined in the definition module are automatically accessible in the implementation module.

You can use them in the implementation module as if they were declared at the beginning of that module.

The reserved word IMPLEMENTATION is the signal to the compiler that it must look for a definition module. If this word is omitted, no search for a definition module is performed.

What the compiler actually looks for is a symbol table file. This file has the extension .SYM or .LMS and a file name taken from the first eight characters of the module name. It does not read the source code of the definition module. It also looks for a relocatable binary file, which will have the extension .REL, .SMR or .LMR depending upon which version of the compiler you are using. It needs this so that it can append the code generated for the implementation module to the code generated for the definition module. The compiler also looks in the libraries SYMFILES.LBR and MODULA2.LBR if it cannot find the .SYM or .REL files.

You can replace the implementation module associated with any given definition module. Provided that the new implementation works, any module which imports the module will not be able to tell the difference.

However, if you change (and re-compile) a definition module, you must also recompile the implementation module. You must also recompile all modules which import the module you have just changed. If you fail to do so, the linker will give you an error message (X recompiled since Y) and your linked program may not work. This is discussed more fully in the user guide in the sections on the linker.

A module starts with one of the above headers and ends with an END followed by a repetition of the module name, and a period. For example:

```
MODULE Example;
(*lots of code*)
END Example.
```

The module construct allows you to divide a program into logically related parts. Each part is written as a separate module. This reduces the amount of code that must be recompiled whenever you make a change to your program. It also allows several people to work on a program at once, since they can each work on a different module.

4.1 IMPORT and EXPORT

Because your program is written as separate modules, rather than as one great mass of code, as would be the case in Pascal, when you compile a module, you have to be able to tell the compiler what resources you want to use from other modules. This is achieved with the IMPORT statement.

The IMPORT statement is used at the beginning of a module to import identifiers from outside the module. In a module, the only identifiers which are automatically available are the standard identifiers, as listed in section 3.2. Any other identifier must be imported.

The IMPORT statement takes one of the forms:

```
FROM module-identifier IMPORT identifier,...;
```

or

```
IMPORT module-identifier,module-identifier,...;
```

The module-identifier is the name of a module which contains resources that you want to use. In the first form, following the IMPORT reserved word, you give a list of the identifiers for the resources that you want to access. These resources can be procedures, variables, types or constants. Any identifier that has been declared in the definition module of the named module can be accessed.

For example:

```
FROM Terminal IMPORT WriteString,WriteLn;
```

Terminal is the name of a module which is supplied with the compiler. WriteString and WriteLn are two procedures contained in that module. The FROM statement announces that you intend to use these procedures in the module you are writing. Because of this statement, when the compiler sees WriteLn, it knows that the procedure WriteLn from Terminal is the one to use.

You may reference the imported identifiers just as if they were declared in the current module.

This means, for example, that you cannot declare the identifiers again, as they would then be doubly declared. Neither may you import them a second time, for the same reason.

When you import from a module, only the identifiers named are imported from the module. `Terminal`, for example contains other identifiers as well as `WriteLn` and `WriteString` but these have not been mentioned in the `FROM Terminal IMPORT` statement and are therefore not accessible in our module.

There is one exception is this: when you import an identifier which is an enumeration type identifier, all the constants associated with that identifier are also imported implicitly. For example, the module `ScreenIO` from the Editor Toolkit contains an enumeration `Edits` which contains a list of all the possible screen operations that can be performed by `ScreenIO`. Importing the identifier `Edits` makes all the identifiers for the various screen operations available as well. We shall return to this point in the section on enumeration types.

The second form of import took the form:

```
IMPORT identifier,...;
```

In this form, each identifier is the name of a module to be imported. Every identifier in each named module is imported, but it is qualified by the name of the module. To reference them, you must precede each reference by the name of the module from which the identifier has been imported. For example:

```
IMPORT Terminal;
(*import from standard module Terminal*)
...
BEGIN
      Terminal.WriteString(' hi there!');
      Terminal.WriteLn;
```

Contrast this with the way we access these procedures after using the first form of the `IMPORT` statement:

```
FROM Terminal IMPORT WriteString,WriteLn;
...
BEGIN
      WriteString(' hi there!');
      WriteLn;
```

In the preceding example, `WriteString` had to be qualified by `Terminal`.

When an identifier is imported in this qualified manner, it will not clash with any other identifier. As a result, if an identifier is available in two modules and you want to use both of them in a certain module, you can import one (or both) using qualified import and this will avoid the `doubly declared identifier` problem.

In this compiler, when you compile a definition module, a file with the extension `.SYM` or `.LMS` is created. This file contains all the symbols defined in the definition module in a form which can be quickly loaded by the compiler. When you import from another definition module, the compiler expects to find an appropriate `.SYM` or `.LMS` file. For example:

```
IMPORT Conversions;
```

requires that the compiler can find a file `CONVERSI.SYM` or `CONVERSI.LMS` depending on the version of the compiler that you have. Note that only the first eight characters of the module name are used in the file name and that they are converted to upper case. If you use `$` or `_` in a module name, these characters will be ignored.

4.1.1 Nested Modules

Modules can be nested inside other modules. This isolates the identifiers inside the module from code outside the module. In addition, it prevents code inside the nested module from accessing identifiers from outside the module (other than the standard identifiers which are always available) unless you explicitly import them.

If you are new to Modula-2, it is probably best to ignore nested modules for the time being.

In a nested module, you can only import identifiers from the surrounding module. For example:

```
 PROCEDURE Fred;
VAR i:INTEGER;
      MODULE InFred;
      IMPORT i;
      ...
```

i is the variable local to Fred.

The FROM form of the import statement cannot be used, because the separately compiled definition modules are not visible from a nested module.

Note that, if we import a module using the second form (e.g. IMPORT Terminal), the identifier of the module is defined in the importing module, and hence cannot be used for another purpose. It can also be imported into a nested module:

```
MODULE Outer;
IMPORT Terminal;
      MODULE Inner;
      IMPORT Terminal
      ...
```

We can import Terminal into the inner scope because it is a known identifier in the outer scope the fact that it is a module identifier is irrelevant.

Also, we can import identifiers which have themselves been imported. For example:

```
MODULE Outer;
FROM Terminal IMPORT WriteString;
      MODULE Inner;
      IMPORT WriteString;
```

Modules can be nested inside other modules. They can also be nested inside procedures. The nested module is part of the declarations of the surrounding object, so it must precede the executable code for that surrounding module or procedure.

4.1.2 EXPORT

The EXPORT statement is only used in modules which are nested inside other modules. This is a change from earlier versions of Modula-2 in which the EXPORT statement was also used in definition modules. Since the clarifications and modifications introduced by Niklaus Wirth, all identifiers are automatically exported from a definition module.

If a module contains an EXPORT statement, it must immediately follow any IMPORT statements, or the module header if there are no IMPORT statements.

The statement consists of the keyword EXPORT followed by a list of identifiers that are to be exported:

```
MODULE Inner;
IMPORT WriteString;
EXPORT Proc1,Proc2;
...
```

The keyword EXPORT may be followed by the keyword QUALIFIED:

```
MODULE Inner;
EXPORT QUALIFIED Proc1,Proc2;
```

In this case, to use any of the exported identifiers in the surrounding module, you must qualify the identifier with the name of the module from which it has been imported. For example:

```
Inner.Proc1
```

If the QUALIFIED keyword is not used, the identifier is given without the qualifying module name and, once again, you must ensure that the identifier does not conflict with any other identifier in the scope to which you are exporting.

There may be several IMPORT statements in any module, but they must be grouped together at the beginning of the module. If there is an EXPORT statement, it must immediately follow the IMPORT statements.

4.1.3 Some Advanced Rules

In this compiler, if an identifier is imported twice in a single `IMPORT` statement, no error will be produced. This is because the compiler marks the symbols to be imported and then enters them into the symbol table after they have all been marked.

However, importing an identifier twice in different import statements will produce an error.

If we use the first form (`FROM Terminal IMPORT ...`), then only the identifiers after the `IMPORT` are defined; `Terminal` could be used for something else, though it would be poor style to do so.

4.2 CONST Declarations

Constant declarations are introduced by the keyword `CONST`. The keyword is followed by one or more constant declarations.

Constants can be defined to be numeric values (Real, integer or cardinal), strings and elements of enumerations.

For example:

```
CONST pi=3.141592653589793; (*etc*)
      twopi=2.0*pi;
      ProgName=' Areas of Ellipses';
      cases=100;
      smallversion=cases<50;
      (*this is a BOOLEAN*)
```

Notice that we can use expressions (eg `2.0*pi`) provided that the expression can be evaluated by the compiler. For example, an expression line `SIN(pi/2.0)` requires the use of a `SIN` routine which can only be called when the program is run. This prevents the use of this expression in a constant.

As well as simple constants such as those above, the compiler can create constants using information from type declarations.

```
TYPE   Context=(weak,firm,strong);
       ContextArray=ARRAY[0..cases-1] OF Context;
       ContextSet=SET OF Context;

CONST  assigncontext=firm;
       ArraySize=TSIZE(ContextArray);
       ParamContext=ContextSet{firm,strong};
```

(Type declarations are described in the next section)

A constant declaration associates a value with an identifier. This identifier is sometimes called a *manifest constant*, since the declaration makes the constant easy to find and to change.

You should use constant declarations for any value which could conceivably change. It is also useful to name dimensionless constants, like pi above, even though they never change.

On the other hand, you should avoid naming constants just for the sake of naming them; you should avoid infelicities like:

```
CONST two=2;
```

4.3 TYPE Declarations

Type declarations allow you to construct new types from simpler types and to equate type identifiers. They are introduced by the keyword `TYPE`. The keyword is followed by one or more type declarations.

A type declaration is a means of specifying a set of values which variables of the type can take. If you are only familiar with simple languages like BASIC or FORTRAN, or with languages like COBOL which confuse types and variables, then it is important that you make sure you understand the difference between a variable and a type.

A type declaration does not allocate any memory into which values can be placed when the program is run. Only a variable will do this. This means that the following is wrong:

```
TYPE InnerPlanet=(Mercury,Venus,Mars,Earth);
...
InnerPlanet:=Venus;
```

As there is no memory associated with the type `InnerPlanet`, you cannot assign a value to it. `InnerPlanet` is an enumeration type, as described below. Before you can use a type in a program, you must have a variable of the type. You manipulate the data by using the variable's identifier. You can have any number of variables of a given type. Each such variable is called an *instance* of the type.

A type is rather like a design, let us say for for a kitchen kettle. You cannot actually boil water in the design. Before you can do that, you have to use the design to build a kettle this is an instance of the design and then you can make your cup of coffee (you will probably be ready for one).

Modula-2 provides a number of constructs for creating types composed of simpler types and restricting types to subranges of other types. These simpler types are either types you have already declared (perhaps in another module) or else they are from a set of predefined types which are built into the compiler.

4.3.1 The Predefined Types

Modula-2 contains a number of predefined types. Each such type has an identifier.

The types `INTEGER` and `CARDINAL` take whole numbers as their values. Integers are signed values, while cardinals are un-signed.

In this compiler, an integer value must be between -32768 and 32767 (inclusive). A cardinal value must be between zero and 65536 (inclusive).

So, for example, `-10` is an integer value, `23` can be either integer or cardinal, while `40000` is cardinal, since it is too large to be an integer. `4.3` is neither integer or cardinal since it contains a fractional value.

You can assign an integer to a cardinal (and vice versa) but you cannot directly compare an integer to a cardinal.

Because the types are so similar, it is very easy to forget whether a given variable should be INTEGER or CARDINAL. A good rule to follow is to make all variables CARDINAL unless it is possible for them to have negative values.

The type CHAR takes as it's values any ASCII character. There are 256 such values although only the first 128 are usually used. The characters in the second 128 tend to vary from computer to computer.

The type BOOLEAN takes the values FALSE and TRUE. FALSE and TRUE are standard identifiers representing constant values. FALSE is less than TRUE. This means that you can perform tests like:

```
IF a<=b THEN
```

where a and b are BOOLEAN variables. This returns TRUE if a is FALSE or if b is TRUE.

The type REAL takes as its values a set of rational numbers. (The term *Real number* is a misnomer which was first perpetrated by FORTRAN). The exact format of these values is described in the user guide. For most purposes, it will suffice to know that, in **FTL Modula-2**, a real value has about 16 significant digits, and that the largest real value which can be represented is about 1E152 while the smallest positive real which can be represented is about 1E-152.

Many Modula-2 compilers will have ranges of values and precisions that are substantially less than these.

There is also a type BYTE, which only takes one byte of storage, and takes values from 0 through 255. This type is not a standard type. It must be imported from SYSTEM, which is described in full later.

The MSDOS and 68000 compilers contain the additional types LONGCARD and LONGINT which are 32 bit cardinals and integers. In these versions, you can have long literals which, such as 100000. This is too large to be a CARDINAL but it is a LONGCARD or even a LONGINT.

4.3.2 Type Equivalence

The simplest form of type declaration is to declare that a new type is the same as an existing type.
For example:

```
TYPE LabelNo=REAL;
```

Whenever `LabelNo` is used, it is exactly the same as using the predefined type `REAL`.

This is sometimes useful when you may want to change the type later. For example, you may have some code that could conceivably use either reals or integers.

4.3.3 Enumerations

An enumerations is a list of possible values. Each value is represented by an identifier.

For example:

```
TYPE InnerPlanet=(Mercury,Venus,Earth,Mars);
```

The identifiers in the brackets are declared as constants. They have values which represent their position in the list. Their type is the type being declared, in this case `InnerPlanet`.

These constants are ordered. `Mercury` is less than `Venus`, which is less than `Earth`, etc. So if we have two variable of type `InnerPlanet`, we can compare their values.

```
VAR a,b:InnerPlanet;
...
IF a<b THEN
```

which is true if the value of `a` occurs before the value of `b` in the enumeration. The constants are not compatible with variables of type `INTEGER`, `CARDINAL` or any other enumeration type. The following is wrong:

```
VAR    c:CARDINAL;
       a,b:InnerPlanet;
       ...
       c:=a;
```

The compiler will diagnose an error. Even though, internally in the program, `Mercury` is represented by zero, `Venus` by one, and so on, they are not compatible with `CARDINAL`.

This is a feature of Pascal and its derivatives which marks a major advance in programming reliability over earlier languages like PL/I. It is called strong typing. It increases the reliability of your programs since more errors are detected when the program is compiled instead of being (possibly) detected when the program is tested.

Recall that, when an enumeration type identifier is imported into a module, all the identifiers in the enumeration are imported implicitly. This means, for example, that if a module contains this statement:

```
FROM Planets IMPORT InnerPlanet;
```

then not only `InnerPlannet` but also `Mercury`, `Venus`, `Earth` and `Mars` are imported.

You can use the enumeration constants to define other constants:

```
CONST ThirdPlanet=Earth;
```

Which results in `ThirdPlanet` being identical in both value and type to `Earth`.

4.3.4 Subrange Types

A subrange allows you to specify that a new type only contains values that are a subrange of another type.

```
TYPE   EarthLikePlanets=[Venus..Mars];
       UpperCase=['A'..'Z'];
```

The type `EarthLike` has been limited to the values `Venus`, `Earth` and `Mars`. `UpperCase` has been limited to the uppercase characters.

In general, a variable of the type may take on any value between the lower bound and the upper bound (inclusive). Subranges of REAL are not permitted.

The lower bound must be given first. This means that the declaration

```
EarthLikePlanets=[Mars..Venus]
```

would give an error; the low bound may equal the upper bound, but it must not exceed it.

Each enumeration type has an underlying base type. The enumeration is a restricted range of its base type. For example, the base type of EarthLikePlanets is InnerPlanets while that of UpperCase is CHAR. The same amount of storage is required for a variable of the subrange as is required for a variable of the base type.

Because EarthLikePlanets is derived from InnerPlanets (InnerPlanets is the base type), variables of the two types can be used together in comparisons and assignment statements.

We can also declare subranges of the predeclared types INTEGER and CARDINAL. For example:

```
SmallInts=[-10..10];
SmallPositives=[0..10];
```

The first of these must clearly have base type INTEGER, since it can take negative values. The base type of the second could be either INTEGER or CARDINAL.

This compiler differs from the standard slightly in this area. In the standard language, as the statement stands above, SmallPositives has base type CARDINAL. However, in this compiler, it is of a type compatible with both INTEGER and CARDINAL. There is actually a *hidden type* which is used for constants which is used as the base type of any subrange whose bounds are compatible with both INTEGER and CARDINAL.

If this were not the case, then the following would produce an error:

```
VAR    i:INTEGER;
       s:SmallPositives;
       ...
       IF i<s THEN ...
```

Since the base type of s (CARDINAL) cannot be directly compared with a variable of type INTEGER.

You can force a subrange to have one type or the other as its underlying (or base) type. For example, we could make it INTEGER by preceding the right hand side with INTEGER like this:

```
SmallPositive=INTEGER[0..10];
```

In this compiler, there is also a type BYTE. If you want to reference BYTE, you must import it from the module SYSTEM (SYSTEM is described in full later). We can use BYTE to qualify a subrange.

```
SmallPackedPositive=BYTE[0..10];
```

Now the base type of SmallPackedPositive is BYTE. BYTE is declared as:

```
BYTE=[0..255];
```

However, if you declared it like this in your program, it would require two bytes of storage for each variable. The version of BYTE in SYSTEM only uses one byte, as will variables of type SmallPackedPositive when they are given the explicit base type BYTE.

4.3.5 Set Types

A set is a data type whose values are any combination of an enumeration or subrange data type. For example:

```
TYPE   NumberSet=SET OF BYTE;
       CHARSET=SET OF CHAR;
       SetOfPlanets=SET OF InnerPlanets;
```

The first set (NumberSet) would declare a set which could contain elements between 0 and 255. The values of the second set (CHARSET) are collections of characters. For example:

```
CONST AlphaNumeric = CHARSET{'A'..'Z','a'..'z','0'..'9'};
```

This defines AlphaNumeric to be a CHARSET containing the alphabetic characters (both upper and lower case) and the numeric characters.

The final set (SetOfPlanets) takes collections of planets as its elements. For example:

```
CONST EarthLike=SetOfPlanets{Venus,Earth,Mars};
```

In a set, a value is either in the set, or not in the set. The compiler actually represents a set as a list of bits. Each possible element has a bit which is 1 if the element is present and 0 if it is not present.

Some of the values of NumberSet are

NumberSet{} (the empty set),
NumberSet{0},
NumberSet{0,1} and
NumberSet{5,7,234}.

There is a predeclared set type BITSET. This is declared to be

```
BITSET=SET OF [0..15];
```

Normally, a set construct is preceded by the identifier for the set type to which the construct is to belong. If this identifier is omitted, the construct defaults to type BITSET:

```
VAR t:CHARSET;
...
t:=CHARSET{'C','A','B'};

VAR s:BITSET;
...

s:={0,1,2};
(*The same as s:=BITSET{0,1,2};*)
```

In Modula-2, the lower bound of the set is normally required to be zero. This compiler allows any lower bound. So, in this compiler, you can declare:

```
OffSet=SET OF [1024..1056];
```

and variables of this set type will only require 5 bytes each.

This compiler allows at most 1024 elements in any set. This is rather more than most compilers allow. In fact, this compiler allows you to have a `SET OF CHAR`. Actually, the only reason why the limit of 1024 elements exists is because you cannot use any literal that contains more than 128 bytes.

4.3.6 Array Types

An array is a type in which values are lists of values of another type, each value being associated with an identifying value. For example:

```
Diameters=ARRAY InnerPlanet OF REAL;
```

The array `Diameters` has four elements. In each element, we will (presumably) store the diameter of the associated planet:

```
VAR Diameter:Diameters;
(*we store values in a variable, the type merely
 describes the possible values*)
...

Diameter[Venus]:=12.63e6; (*metres*)
```

Note, once again, that we must declare a variable of the type to store values in we cannot store values directly in the type.

The array type differs from the set type because each element in an array has an associated value and is always present. In a set, it is simply present or absent.

The index type must be a subrange or an enumeration. An explicit subrange can be given:

```
Vector=ARRAY[1..10] OF REAL;
```

Note that the brackets of the subrange must be included.

You can also use the types CHAR and BOOLEAN as the index types of arrays:

```
KeyChar=ARRAY CHAR OF CARDINAL;
BoolArr=ARRAY BOOLEAN OF CARDINAL;
```

We can have more than one index in an array:

```
Matrix=ARRAY[1..10],[1..10] OF REAL;
```

It would be an error to write this declaration as:

```
Matrix=ARRAY[1..10,1..10] OF REAL;
```

This is allowed in Pascal, but not in Modula-2. Also, you must use the symbol ... Some Pascal compilers allow you to use a colon (:) instead of ... This is not permitted in Modula-2.

4.3.7 Record Types

In an array, all the elements have the same type. The record construct allows us to create a type composed of different types of data.

Unlike an array, in which the individual elements are identified by members of a subrange or an enumeration, in a record, each element is given its own identifier. For example:

```
Months=(Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
                        (*an enumeration type*)

Date=RECORD
       Day:[0..31];
       Month:Months;
       Year:CARDINAL;
       END;
```

The value of this record are dates. Suppose we have a variable of type

Date:

```
VAR BeethovensBirthday:Date;
```

We reference the individual elements of the variable as follows:

```
BeethovensBirthday.Day:=16;
BeethovensBirthday.Month:=Dec;
BeethovensBirthday.Year:=1770;
```

The WITH statement, described later, overcomes the tedium of specifying BeethovensBirthday three times.

This construct would allow us to write:

```
WITH BeethovensBirthday DO
      Day:=16;
      Month:=Dec;
      Year:=1770;
      END (*WITH*);
```

Once we have declare a record type, we can use it to create arrays and in other record types.

```
TYPE   BirthdayArray=ARRAY[1..1000] OF Date;
VAR    BirthDays:BirthdayArray;
...

      Birthdays[10].Year:=1983;
```

Sometimes, we want to use any of several variants of a record depending upon circumstances. For example, we may receive our dates in either the form given above, or as a Julian date:

```
Datetype=(Normal,Julian);
Date=RECORD
       Year:CARDINAL;
       CASE class:Datetype OF
            Normal:    Day:0..31;
                       Month:Months|
            Julian:    JulianDay:[1..366]
                       END (*CASE*);
            END (*Date*);
```

At any time in any given variable, only one of these variants can be active. Which one it is is determined by the value of the variable `class`. For example:

```
VAR d:Date;
...
       d.class:=Julian;
       d.JulianDay:=254;
```

Note that we assigned a selecting value to the *discriminant* (`class`) before assigning to any field in the variant part. Some compilers may check that when you reference a variable in the variant part, that it is in the currently active variant. Our compiler is not yet that clever (actually, we do not know of any Modula-2 compiler that is. Some Pascal compilers are capable of checking this).

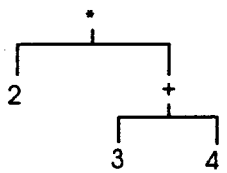
Whichever variant is active, the variable `Year` is available. Because only one variant is active at any one time, the compiler conserves space by using the same memory for every variant. The total memory required for the variant part is therefore equal to the memory used by the largest variant.

You can use more than one variant in a record description and you may nest variants within other variants. Pascal programmers will note that there is a separate `END` for the variant. (In Pascal, you can only have one variant part and it must be last.)

4.3.8 Pointer Types

A pointer type is a type whose values are the addresses of other values. Pointers are particularly useful in processing structures which vary, either because they grow as the program proceeds, or because they depend upon the input data (or both). Such data structures are called *Dynamic data structures.*

For example, suppose we are reading arithmetic expressions from the terminal and we wish to represent these expressions by a tree. On this tree `2*(3+4)` will look like:



More complicated expressions will produce more complicated trees. The size and shape of the tree is not known until we read the data.

The declarations to handle this structure contain two parts. Firstly, we declare a pointer to the (yet to be declared) node.

```
PNode=POINTER TO Node;
```

This is the only situation in which you can reference an identifier before it is declared.

Next, we declare a record type for `Node`. Each instance of `Node` is going to represent one operator. For each node, we need to record the operator used and pointers to its operands.

```
Node=RECORD
       CASE Terminal:BOOLEAN OF
            FALSE:
                 Operator:CHAR;
                 LeftOperand:PNode;
                 RightOperand:PNode|
            TRUE:
                 Value:INTEGER;
                 END;(*CASE*)
            END;
```

If we declare a variable of type PNode:

```
VAR BaseNode:PNode;
```

The compiler reserves space, not for the entire tree, but for a pointer to the base of the tree. To actually generate the base of the tree, we must use the procedure NEW.

```
NEW(BaseNode);
```

This allocates storage for an instance of the type Node and returns a pointer to it in BaseNode. We can now reference the elements of the record as:

```
BaseNode^.Operator:='*';
```

Notice that BaseNode^ serves the same role here as did BeethovensBirthday in a previous example. The extra symbol ^ is used because, whereas BeethovensBirthday is a record, BaseNode is a pointer which points to a record.

To make this clearer, suppose we have another variable of type PNode:

```
VAR    ThisNode:PNode;
       ...
       ThisNode:=BaseNode;
```

The assignment assigns the pointer contained in BaseNode to ThisNode. As a result, they both now point to the same object.

However, if we had written:

```
       ThisNode^:=BaseNode^;
```

Then the contents of the value pointed to by BaseNode would be written into the area pointed to by ThisNode. They would now still point to different areas of memory, (assuming they were not equal to start with) but those areas would contain the same value.

If we had another variable which has as its type a pointer to some other object:

```
VAR AnotherVariable:POINTER TO Date;
```

then the statements:

```
AnotherVariable:=ThisNode;
AnotherVariable^:=ThisNode^;
```

are both illegal, since the pointers do not point to the same type.

Finally, there is a constant `NIL` which can be assigned to any pointer. It is used to denote that the pointer is not pointing to any object. It is compatible with all pointer types.

Of course, pointers are not restricted to pointing to record types; they can point to any type at all. For example, it could be useful to use a character pointer:

```
pCHAR=POINTER TO CHAR;
```

4.3.9 Procedure Types

A variable with a procedure type takes as its values (the addresses of) procedures. A procedure type is a first class type; you can use them as freely as you would use any other type. For example, you can have arrays of them, or pointers to them.

A procedure type describes precisely a type of *procedure* there is not just one type for all procedures. These types are described in detail in section 4.6.2, after the description of procedures.

4.3.10 Private Types

In a definition module, you can declare a type without giving a definition.

For example:

```
DEFINITION MODULE MakeEdit;
       TYPE EditFile;
       ...
```

This allows other modules to use the type, but not to perform any operations with it other than assignment and allocation of space. The type is called private (or sometimes, *opaque*) because external modules are unable to access the internal structure of the type.

The rest of the definition of the type is given in the implementation module.

Most compilers require that the private type be no larger than a pointer. In practice, this means that you often have to allocate objects on the heap which you would prefer to allocate statically.

This compiler removes this restriction: any type can be a private type, there is no limitation on size.

When you compile the definition module, the compiler assumes that the size of the type will be equal to that of a pointer. If, when the implementation module is compiled, the size is discovered to be different, the symbol file, which retains the exported information, is updated.

This update only takes place if the size has changed. You will receive the message `*** SYM file rewritten ***` at the end of the compilation if this was necessary. You must recompile all modules which import this module after receiving this message, just as if the definition module had been recompiled.

There are two restrictions on this facility: If you use the private type in the definition of another type in the definition part, then the type must be equal in size to a pointer, and if you use the private type to define another type in the implementation module, you must fully define the private type first, or else ensure that its size is the same as a pointer.

That is, you cannot use the type until you give its actual size by declaring it in full.

Remember that, if you use this feature, your programs will not be portable.

Pointers, Integers and Cardinals all use two bytes in the CP/M-80 and small memory MSDOS version of this compiler.

Pointers are 4 bytes in the 68000 and large memory MSDOS versions of the compiler. Integers and cardinals are still two bytes in those compilers.

4.4 Type Compatibility

Modula-2 is a strongly typed language. This means that in many contexts, the type of an expression must be of a particular type or types.

This contrasts with some other languages which, given a dubious expression, will try to find a valid interpretation for it no matter how outlandish it may be. With a strongly typed language, you sometimes have to write a little more code to make clear what you want to do but the advantage is that the compiler can detect coding blunders much more reliably.

In Modula-2 there are two classes of compatibility: Strong compatibility, and assignment compatibility. Strong compatibility is usually referred to just as compatibility.

Two types are (strongly) compatible if any of the following is true.

i) If they are the same type, or equivalent types. For example, if `T` has been declared as `T=S` Then `T` is compatible with `S`.

   But, if `T` and `S` merely have the same structure:

```
T=ARRAY[1..10] OF CHAR; S=ARRAY[1..10] OF CHAR;
```

   Then they are not compatible.

ii) If both types are subranges of the same type, or one is a subrange of the other.

   In addition, when both types are subranges, this compiler requires that the subranges overlap.

   For example, If we have the declarations `a:[1..10]; b:[3..20];`, then `a` and `b` are compatible, both with each other and with variables of type `INTEGER` or `CARDINAL`.

However, the variables declared as `c:["A".."Z"];` `d:["a".."z"];` are not compatible, even though they share the base type `CHAR`, because their ranges do not overlap.

iii) If the first operand is of the formal type `ARRAY OF sometype` and the second operand is an array of the same type. The form `ARRAY OF sometype` can only be used in a procedure header. It is often called an *open array parameter*. It is discussed in detail in the section on procedures.

iv) If one operand is a string literal of length one and the other is of type `CHAR`.

v) Depending on value, integer literals can be compatible with either `INTEGER`, `CARDINAL`, or both. For example, `7` and `30000` are compatible with both these types, `-9` is only compatible with `INTEGER` while `50000` is only compatible with `CARDINAL` because it is too big to be an integer value.

vi) For procedure and function types, structural equivalence is always used. This is described more fully later (See 4.6.2).

Two types are assignment compatible if they are compatible. In addition, two operands are assignment compatible in the following situations:

i) If each operand is of type `INTEGER`, `CARDINAL` or `BYTE` (or a subrange of these). In the case of subranges, the ranges must overlap. Hence, a variable of type `BYTE` is assignment compatible with one of type `INTEGER`.

ii) If the second operand is a string literal and the first operand is an array of `CHAR` with as least as many elements as there are characters in the string. If there are fewer characters in the string than elements in the array, a null character (the character `0C`) is appended to the string. Any remaining elements in the array are undefined. If the formal parameter is of type `ARRAY OF CHAR`, no null character is appended (since it is always the correct length).

iii) If the left hand side (or the formal parameter) is of type `ADDRESS` and the right hand side is a pointer.

iv) You can assign from a cardinal or integer to a real but not the other way around. Similarly, you can assign a short integer or cardinal to a long integer or cardinal but not the other way around. (Note that the Z80 compiler does not have longs because of memory constraints.)

v) There are several type imported from SYSTEM that relax the type checking rules. These are WORD, ADDRESS and BYTE. ARRAY OF BYTE also relaxes the type checking rules. See the section on the standard module SYSTEM for details. In function and procedure calls, Modula-2 requires strict compatibility for variable parameters and assignment compatibility for value parameters. By strict compatibility, we mean that the types must be identical (not merely assignment compatible). However, this does not apply to the types from SYSTEM.

For example:

```
VAR  w:WORD;
     i:INTEGER;
     c:CARDINAL;
     r:REAL;
     ...
     (*These are OK*)
     i:=c;c:=i;
     w:=i;i:=w;
     c:=w;w:=c;
     r:=i;r:=c;
     (*These are not OK*)
     i:=r;i:=c;
     IF i=c THEN ...
     IF w=i THEN ...
```

4.5 Variable Declarations

The variable declarations are introduced by the keyword VAR. The keyword may be followed by one or more variable declarations.

For example:

```
VAR  i,j,k:INTEGER;
     c:CARDINAL;
```

The identifiers to the left of the colon are the variables being declared. The identifier to the right is the type to be given to the variables.

You can use a type construct in place of the type identifier on the right hand side.

```
VAR    a,b:ARRAY[1..10] OF CHAR;
       c:ARRAY[1..10] OF CHAR;
```

However, this is usually not advised since, while a and b are compatible in the above declarations (`a:=b;` is acceptable), neither is compatible with `c`. These variables have as their type an *anonymous type*, but they are different anonymous types. Hence the statement `a:=c;` will give a compilation error. Note that, in any case, `a[2]:=c[3];` would be accepted, since the array elements are both of type `CHAR`.

When you declare a variable, the memory for it is allocated in one of two ways, depending upon where in a module the declaration occurs. If a variable is declared at the outermost level of a module, the variable is allocated statically, even if the module is nested inside another module.

If the variable is declared in a procedure, then the variable is allocated dynamically. If you have a module nested inside a procedure, then the global variables of that module are allocated dynamically, and so would be the global variables of a module nested inside that module. In other words, once you are inside a procedure, all variables are dynamic.

If the variable is at the outermost level of the main module, or is in a module nested inside that module (but not inside a procedure), the variable is allocated statically.

Static variables have memory allocated to them by the linker. The one piece of memory is always allocated to the variable so that if you place a value in the variable, it will retain that value until it is given a new value.

Dynamic variables are allocated memory when their scope (the procedure in which they are declared) is entered. When that scope is exited, the memory is freed. As a result, the next time you enter the procedure, the variable may not retain its value from the preceding call, since it may not be allocated the same area of memory, or the given area may have been used by another dynamic variable in between calls.

If a variable must retain its value from one call of a module to another, then it must be a static variable, so it must be declared at the level of the main module or of a module nested in the main module.

On the other hand, if a variable is only temporary, then it should be dynamic since not all dynamic variables are allocated memory at any one time and so memory is conserved.

In addition, recursive procedures, which are procedures which may call themselves, need dynamic variables since more than one copy of a variable may be needed at any one time, since the procedure may be activated several times simultaneously.

Consider the following example (PNode was declared a few pages ago):

```
PROCEDURE WalkTree(p:PNode);
BEGIN
      IF p<>NIL THEN
            WalkTree(p^.LeftOperand);
            WalkTree(p^.RightOperand);
            END
      END WalkTree;
```

This procedure is recursive because it call itself. If the variable p was the same for the recursive calls as for the outermost call, then by the time you returned from walking the left subtree, p would be NIL and you would never walk the right subtree.

There is one other way in which you can affect the allocation of a variable: If a variable is to be statically allocated (i.e is global to a module), you can give an explicit address for the variable. For example:

```
TopOfTPA[6]:CARDINAL;
```

TopOfTPA is a variable which will be allocated at absolute address 6 in memory. (In CP/M, this is the address field of the jump instruction into the Basic Disk Operating System (BDOS)). The explicit address has no affect on any other variable in the declaration list. The constant inside the brackets may be a constant, an identifier, or a constant expression. Note that, once you start giving variable explicit addresses, it' up to you to know what you are doing. For example, changing TopOfTPA without indulging in some other smart programming would cause your system to crash.

The usefulness and meaning of this absolute address construct depends upon the version of the compiler you are using and also the machine you are running on. See the user guide for more information concerning your particular machine.

4.6 Procedure Definitions

The final form of declaration is the procedure declaration.

A procedure declaration is introduced by the keyword PROCEDURE. Unlike the other declarations, the keyword PROCEDURE is required at the beginning of each declaration. For example:

```
PROCEDURE Example(k:CARDINAL);
VAR i:CARDINAL;
BEGIN
      FOR i:=1 TO k DO Write(chs[i]) END;
      END Example;
PROCEDURE AnotherExample;
BEGIN
      WriteString(' Another Example');
      END AnotherExample;
```

Note that the name of the procedure is repeated at the end of the declaration.

The declaration consists of three parts:

i) The header (in this case the first line, but it may of course spill over several lines since the ends of lines have no significance to Modula-2).

ii) Declarations (in this case, the VAR declaration, but CONST, TYPE, MODULE and PROCEDURE declarations may also appear).

iii) The body of the procedure, which contains the executable code.

In addition to the variable declared in the declarations, we can declare variables in the header line. k is an example of such a variable.

These variables in the header are formal parameters. When the procedure is called, actual values must be given for each formal parameter. These actual values become the initial values of the formal variables. So that, the statement:

```
Example(2);
```

causes the example procedure to be executed with k equal to 2.

During the execution of the procedure, the values of these formal variables can be changed, just as if they were ordinary variables.

In the example above, k has been *passed by value*. This means that a copy was made of the value of the actual parameter when the procedure was called. As a result, changing the value of k does not change the value of the original actual value (the constant 2 in the example).

If we want any change in a formal variable to also change the value of the associated actual parameter, we precede the formal declaration with the VAR keyword:

```
PROCEDURE Example2(VAR i,j:INTEGER;k:REAL);
```

In this case, i and j are *passed by reference*. That is, the address of the actual parameter is passed to the procedure so that the actual variable can be updated whenever the formal variable is changed.

In this example, k is still passed by value: we would have to repeat the VAR keyword before the k to make it a reference parameter.

```
PROCEDURE Example3(VAR i,j:INTEGER;VAR k:REAL);
```

When a formal parameter is a reference parameter, the associated actual parameter must be a variable (that is, something that could appear on the left hand side of an assignment statement).

For example, the statement:

```
Example2(i,2,3.0);
```

is illegal because a value cannot be used on the left hand side of an assignment.

The compiler will diagnose an error when it finds the constant 3 associated with the reference parameter j, since you cannot make an assignment to a constant.

The definitions we have given so far are called proper procedures. They are called with the procedure call statement. Procedures may also return a value as a result of the call. This allows them to be used in expressions. For example:

```
PROCEDURE SIN(x:REAL):REAL;
BEGIN
      ...
      END SIN;
```

Note that, unlike Pascal, there is no FUNCTION keyword.

The type returned by the procedure is given after a semi-colon after the parameter list. In the case of an empty parameter list, just the parentheses are given:

```
PROCEDURE RANDOM():REAL;
```

whereas with a proper procedure, the parentheses may be omitted. The reason for this will become apparent in a following section.

To call a function, the function name is given followed by the actual parameter values in parentheses. For example:

```
a:=Radius*SIN(Angle);
b:=RANDOM();
```

Note again the presence of the parentheses in the empty parameter list case. The function's returned value is given in a RETURN statement.

```
FUNCTION RANDOM();
BEGIN
      Seed:=3276*Seed MOD 2783;
      RETURN FLOAT(Seed)/2783.0
      END RANDOM;
```

The return statement is followed by an expression which evaluates to give the value returned by the function. The RETURN statement need not be the last statement in the function, but executing a RETURN statement causes the function to return immediately.

A function should never return by 'falling out the bottom', since this implies that no result has been returned.

4.6.1 Open Array Parameters

There is a special form of type declaration which is available only in the formal parameter list of a procedure. This is the open array. The form of the declaration is:

```
PROCEDURE WriteString(string:ARRAY OF CHAR);
```

In this case, the parameter can be any array of characters.

Within the procedure, the arrays always have a low bound of 0 and an upper bound of one less than the number of elements in the array, irrespective of the actual bounds of the actual parameter.

For example, if the variable `a` passed as a parameter to `WriteString` had type `ARRAY [10..20] OF CHAR`, then the first element of the array (`a[10]`) would be accessed in the procedure as `string[0]`. The last element would be `string[10]`.

The inbuilt function `HIGH` will always return the upper bound of such an array:

```
FOR i:=0 TO HIGH(string) DO
      Write(string[i]);
      END (*FOR*)
```

You can also use the `SIZE` function to determine the size of the parameter (in bytes).

Open array parameters can be passed by either value or reference. A reference parameter produces more efficient code, but value parameters are to be preferred for arrays of characters as you can then use a string literal as a parameter.

Of course, you can have open arrays of any type not just of characters. In this compiler, there are no restrictions on the use of open arrays parameters.

There is one special open array type ARRAY OF BYTE. Recall that BYTE is a type that you must import from SYSTEM. An ARRAY OF BYTE matches any type at all. This allows you to write procedures that manipulate blocks of data (good for Input-Output routines). The parameter is passed as if it was an array you can use both HIGH and SIZE.

4.6.2 Procedure Variables

A procedure type takes as values the addresses of procedures. The types and numbers of the parameters for the procedure must be declared. Two procedure types are compatible if their parameter lists are compatible. This is called *structural equivalence* because the entire structure of the declaration is examined to determine compatibility.

Open array parameters may be used within a procedure type declaration. Some examples:

```
TYPE  procex1=PROCEDURE (INTEGER);
      procex2=PROCEDURE (VAR INTEGER;VAR INTEGER; REAL);
      procran=PROCEDURE ():REAL;
      procstr=PROCEDURE(ARRAY OF CHAR);
```

Notice that the right hand side looks like the header line of the procedure with the identifiers (and the colons) left out. Because these are omitted, in the second example, we must repeat VAR INTEGER for the second parameter.

Suppose we have a variable of one of these types:

```
VAR pr:procran;
```
then we can do the following:

```
pr:=RANDOM; (*make pr point to procedure RANDOM*)
a:=pr();    (*call the procedure pointed to by pr*)
```

Where RANDOM is the function declared in section 4.6. There are several things to note about this example.

First, the parentheses are omitted from the reference to RANDOM. If they were included, the reference would be to the value returned by the function, not to the function itself.

Second, the function RANDOM has a type determined by its declaration. As a result, the assignment is allowed (pr and RANDOM are assignment compatible). If we tried to assign SIN to pr, we would get an error because the parameter lists differ.

Third, the call to pr in a:=pr() is written just as if pr were a procedure rather than a procedure variable. This statement has exactly the same effect as a:=RANDOM().

Of course, you can use procedure types in other types. For example:

```
dispatch:ARRAY CHAR OF PROC;
      (*PROC is the predeclared id*)
      ...

      Read(ch);dispatch[ch];
```

This calls a procedure depending upon the character read.

You can also use procedure variables as formal parameters and then pass a procedure name as the actual parameter:

```
PROCEDURE ex3(p:PROC);
...
ex3(AnotherExample);
```

PROC is a standard identifier which is predefined to be a procedure type for parameterless proper procedures.

There are some restrictions in the use of procedure variables. Only procedures which occur at the outermost level of a module can be assigned as values to a procedure variable. This restriction is not enforced by this compiler, so you must be careful to avoid the following:

```
VAR Trouble:PROC;
PROCEDURE A;
VAR i:INTEGER;
      PROCEDURE B;
      BEGIN
            ...
            i:=2;
            END B;
BEGIN (*body of A*)
      Trouble:=B; (*assign procedure*)
      END A;
      ...
      A;(*call to A which defines Trouble*)
      Trouble; (*Oops*)
```

In this example, we are calling a procedure which is not at the outermost level, referencing it through `Trouble`. The trouble is, when we do call it, `A` is not active, so the variable `i` does not exist on the stack (even if it did, the display might not be pointing to it). As a result, the call will overwrite the stack more or less at random.

The *display* is an area of memory which is set aside for pointers to the activation segments of procedures. When a procedure is called, an activation segment is set up for the procedure. A pointer is needed so that procedures nested inside this procedure can access this procedure's variables. The display is used for this purpose.

The second restriction is that you can not use any of the standard procedures, such as `MIN`, `SIZE` etc, as parameters to procedure variables, nor can you assign them to procedure variables.

Procedure variables give you the opportunity to write very powerful constructs. It also gives you the opportunity to write programs that no-one can understand and which never work! Use them sparingly. See the module `KEYBOARD` (which is supplied as part of the Editor Toolkit) for an example of their use.

4.6.3 FORWARD Declarations

Because this is a one pass compiler, you sometimes need to declare a procedure before you define it. This occurs if two procedures are mutually recursive (each procedure calls the other). Without the FORWARD declaration, it would be necessary to place the procedures in separate modules.

To forwardly declare a procedure, use the form:

```
PROCEDURE ForwardExample(i,j:INTEGER);FORWARD;
```

The keyword FORWARD takes the place of the rest of the procedure definition. When you are ready to define the procedure, the procedure header is repeated in full (unlike in Pascal, where only the name is given). The compiler will check that the declarations agree.

Recall that the definition module is a prefix to the implementation module. A procedure declared in the definition module need never be forward declared in the implementation module.

4.7 The Statement Part

The last section of a procedure or module is the statement part.

The statement part is introduced by the keyword BEGIN and is terminated by the keyword END. The END must be followed by the name of the procedure or module which it terminates. In the case of a procedure or nested module, that name is followed in turn by a semi-colon. In the case of an outer module, the name is followed by a period.

Definition modules cannot have statement parts. In implementation modules, the statement part is optional. If the statement part is omitted, only the terminating END (and the identifier and the period or semi-colon) are given the initial BEGIN should also be omitted. In the implementation module, you cannot omit the statement part associated with a procedure or function (except if you are using FORWARD).

The statement part of a module is executed when any program containing the module is loaded. If a module imports another module, then the statement part for the imported module will be executed before the statement part of the importing module. This ensures that any initialization is performed before you call any procedures in a module.

If two modules import each other, the order is undefined, but in this case the loader will issue a diagnostic giving the order actually used.If a module is nested inside another module, its main program part is executed just before the main program part of the surrounding module. If it is nested inside a procedure, it is executed upon entry to the procedure and it is executed every time the procedure is called.

Section 6 describes the statement forms which may be used in the statement part. Example:

```
MODULE Example;
FROM Terminal IMPORT WriteString,WriteLn;
VAR i:INTEGER;
BEGIN
      i:=0;
      END Example.
```

When the program is run, the main program part of `Terminal` will be executed before the main program part of `Example`, because it is imported by `Example`.

5 Expressions

Expressions are used in a variety of ways to produce values which are then used by the statements in which the expressions occur.

An expression is built up by combining several classes of element; entire variables (also called *designators*), function calls, constants and operators. More complicated expressions can be created by adding more elements to a simpler expression. That is, a complex expression is built up from simpler expressions.

5.1 Entire Variables

An entire variable is an identifier, possibly followed by symbols and identifiers to further define the value to be used. An entire variable is sometimes called a designator.

Like expressions, entire variables can be built up from simpler entire variables. Here are some examples of entire variables:

i) A simple identifier. For example:

```
Diameters
BeethovensBirthday
ThisNode
```

ii) An array identifier followed by an array element expression in brackets: `Diameters[PlanetNo]`

iii) A record identifier followed by a period and the name of a field within a record: `BeethovensBirthday.Day`

iv) A pointer variable followed by an up arrow: `ThisNode^`

These are the basic constructs. An entire variable can be built up by repeatedly applying any or all of these constructs to an entire variable.

```
A^[2].rather^.fanciful[example]
```

This entire variable is built up by repeatedly adding more constructs to the simple `A` at the left hand end.

An entire variable can be used in two ways; It can represent either the value which it has been given or it can represent the address into which a new value is to be placed. Which is required depends upon the context of any given instance.

If it appears on the left hand side of an assignment statement, or as the actual parameter of a formal parameter preceded by VAR, then the address is used, otherwise the value is used.

5.2 Constants

Constants are just numeric and string constants, as described in the section on lexical items, and identifiers which have been declared to have constant values in CONST statements. For example:

```
2 'A String' 27c pi
```

The compiler *folds* constants, so you can quite happily write `pi/2.0` without worrying about doing an unnecessary division at run time. The compiler will calculate `pi/2.0` when it compiles the program. However, do not write `pi/2` as type conversions are not folded.

5.3 Function Calls

We have already seen an example of a function call. It consists of the name of the function followed by a parameter list. If the function has no parameters, an empty parameter list must be given. The parentheses are compulsory. For example:

```
SIN(PI/2.0)
RANDOM()
```

A number of standard functions are supplied with Modula-2. These should not be confused with functions which are supplied in standard modules, since they do not need to be imported to be accessible.

The standard functions are:

`ABS(x)` Returns the absolute value of an integer or real parameter. The returned type is the same as the parameter type.

`CAP(ch)` Returns the character `ch`, converted to upper case if it is lower case alphabetic. Characters with the top bit set (that is, with ordinal values greater than 127), are never changed.

`CHR(x)` Returns the character with collating sequence value `x`. `x` must be an integer or a cardinal. If the value is out of range, the value returned uses only the bottom eight bits of the value.

`FLOAT(x)` Converts a cardinal number to a real number.

`HIGH(x)` Returns the upper bound of array `x`. `x` can be any array variable.

`MIN(x)` Returns the minimum value that can be taken by a variable with type `x`. For example, `MIN(INTEGER)` is `-32768`, `MIN(CARDINAL)` is `0`. This function can be used with `INTEGER`, `CARDINAL`, `CHAR` and `BOOLEAN` as well as any subrange enumeration.

`MAX(x)` Returns the maximum value a variable with the given type can take. It can be used with the same types as `MIN`.

`ODD(x)` Returns `TRUE` if `x` is odd. `x` must be a cardinal or an integer value. It cannot be real.

`ORD(x)` The function `ORD` in this compiler works somewhat differently to the way it is described in Wirth's book. Rather than implement the letter of Wirth's book, we have implemented the function as it is implemented in the PDP-11 compiler, and in a way which is compatible with Pascal and is generally more useful.

According to the book, `ORD` returns a cardinal corresponding to the ordinal value of the value of the parameter in its set. Thus, if `i` is an integer variable containing the value `0`, `ORD(i)` ought to be `32768`. This compiler will return `0`, which is compatible with the PDP-11 compiler.

Also, the type of ORD will be compatible with both INTEGER and CARDINAL if the type of the parameter can only take values within the range 0 through 32767. If the low bound is less than 0, it will be compatible with INTEGER, otherwise, if the high bound exceeds 32767, it will be compatible with CARDINAL. This definition of ORD is much more useful than that in Wirth's book.

`SIZE(x)` Returns the size (in bytes of memory) of the variable given as a parameter. In Revision 2 of Wirth's book, this function had to be imported from SYSTEM. This compiler allows you to use it as a standard function, or to import it.

`TSIZE(x)` Returns the size (in bytes of memory) of a type identifier given as a parameter. This function must be imported from SYSTEM. Note that you use SIZE for variables, TSIZE for types.

`TRUNC(x)` Converts a real number to a cardinal. It is truncated, not rounded. Do not use for integers as a negative value cannot be truncated.

`VAL(t,x)` Returns the value from type t which has ordinal value x. x must be a cardinal.

In addition, you can convert from one type to another by using the name of the required type as a function. For example, if t is a variable of type:

```
VAR t:PNode;
```

Then we could write:

```
t:=PNode(CARDINAL(t)+SIZE(t^));
```

(or, to achieve the same thing)

```
t:=PNode(CARDINAL(t)+TSIZE(Node));
```

Which would make t point to the next Node in memory.

Of course, it is the programmers responsibility to ensure that such a use is valid: Once you use type breaking like this, the compiler will not detect typing errors (unless you don't get the type breaking right!).

Note the use of the pointer symbol (^) in the first example. This is not an error: if this was omitted, the size returned would be the size of `t`, which is 2 or 4 (depending on the version of the compiler you're running). What we want, of course, is the size of the object pointed to by `t`.

You can also use this type breaking (at least in this compiler) for array types. For example:

```
PROCEDURE Open(fn:ARRAY OF CHAR);
BEGIN
        Lookup(a,FileName(fn),reply);
        END;
```

The procedure `Lookup` (from module `Files`, say) needs an array of fixed length, rather than an open array, as the second parameter. Breaking the type like this allows the procedure to be called with an open array actual parameter.

With type breaking, the onus is on you to know what you are doing.

5.4 The Set Builder

There is a special construct for creating sets. A set builder construct consists of a list of values and ranges inside braces. ({ and }). The order of the elements in the list on no way changes the value of the set being built. The elements may be constants, or they may be expressions, and the two may be freely intermixed.

`{2,3,4}` `{4,3,2}` (these are the same set)

If a range of values between two values is required, the subrange notation can be used:

`{2..4}` (which is the same as `{2,3,4}`)

The smaller value of the subrange must be given first. Ranges and simple constants may be freely intermixed:

```
{2,3,7..10} {2,i,7..j}
```

The types of all elements used in a set builder must be compatible with the element type of the set.

The preceding examples are all of type `BITSET` because no explicit type was given as part of the set builder (See 4.3.5). To make the set a different type, precede the opening brace with the identifier for the type:

```
CharSet{'A'..'Z','a'..'z'}
```

5.5 Operators

The above constructs can be combined together with operators. The operators impose constraints on the types of the objects which can be used as their operands. In some cases, the meaning of the operator depends upon the type of its operands.

Each operator has a precedence which determines the order of evaluation of operators. Unless parentheses are used to override the order, operators with highest precedence are performed first.

For example, in the expression `2*3+4`, multiplication has a higher precedence than addition, so the result of this expression is `10`, not `14`.

There follows a table of all the operators ordered by decreasing precedence.

Operator	Use
Highest precedence	
NOT (~)	Boolean complement. ~ is a pseudonym
Next in precedence	
*	Integer, Real and Cardinal multiply; Set intersection
/	Real division; S9et symmetric difference
DIV	Integer and Cardinal division
MOD	Integer and Cardinal modulus
AND (&)	Boolean and; & is a pseudonym
Next in precedence	
+	Integer, Real and Cardinal add; may be unary or binary; Set union
-	Integer, Real and Cardinal subtract; may be unary or binary ; Set difference
OR	Boolean or
Next in precedence	
=	Test for equality
<> (#)	Test for inequality. # is a pseudonym
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
IN	set membership

5.5.1 Arithmetic Operators

The arithmetic operators are +,-,*,/,DIV and MOD. All except / can be used for integer or cardinal operands and perform the usual operations. / can only be applied to real values.

DIV returns an integer or cardinal result. It returns the whole value result of a division. For example, 7 DIV 2 gives 3, -7 DIV 2 gives -3, -7 DIV -2 gives 3. MOD returns the remainder of a division by a positive value: 15 MOD 6 gives 3.

5.5.2 Boolean Operators

The boolean operators are AND, OR and NOT. AND returns true if both its operands are true. OR returns true if at least one of its operands is true. NOT is a unary operator which returns the complement of its operand.

Note that AND and OR are at a higher level of precedence than the relational tests, such as a test for equality. This means that an expression such as: a=b OR a=c will give an error, because the compiler will attempt to compile it as ((a=b) OR a)=c which would only be a valid expression if a, b and c were all boolean variables. To avoid this, you must write the expression as (a=b) OR (a=c).

5.5.3 Set Operators

The set operators are +,-,*, / and IN.

+ (Set Union) Returns a set containing all the elements of its operands. A+B={x | (x IN A) OR (x IN B)}

```
{2,3,5}+{1,3,6}={1,2,3,5,6}
```

- (Set Difference) Returns the elements of its left operand with any elements which also occur in its right operand removed. A-B={x | (x IN A) AND (NOT (x IN B))}

```
{2,3,5}-{1,3,6}={1,5}
```

* (Set intersection) Returns a set whose elements are those contained in both its operands. A*B={x | (x IN A) AND (x IN B)}

```
{2,3,5}*{1,3,6}={3}
```

/ Returns a set whose elements are those contained in one operand set or the other, but not both. A/B=A+B-A*B

```
{2,3,5}/{1,3,6}={1,2,5,6}
```

IN Returns true if its left operand is an element of the set which is its right operand. For example:

```
2 IN {2,3,5} is TRUE

6 IN {2,3,5} is FALSE
```

5.5.4 Relational Operators

The relational operators can all be applied to INTEGER, CARDINAL, BOOLEAN, CHAR and REAL operands, to subranges and to enumerations. This compiler also allows them to be applied to arrays of characters. When arrays of characters are compared, the comparison stops at the end of the shorter string or at the first zero byte in the string. (Recall that a zero byte terminates a string). Hence, the following comparison return TRUE:

```
VAR t:ARRAY[1..10] OF CHAR;
...

      t:='ABC';
      IF t='ABC' THEN ...
```

In addition, = and <> can be applied to pointers and sets. <= and => can also be applied to sets.

When applied to sets, <= requires that every element in its left operand be an element in its right operand. That is, that the left set is a subset of the right set (they may, of course be equal). >= is the same with the left and right operands reversed. For example:

```
{2,3,5}<={2,3,5,6}
```

since every element of {2,3,5} is also an element of {2,3,5,6}.

Note that it is not possible to directly compare pointers other than for equality or inequality. If you really want to compare pointers in this way, coerce the pointers to cardinals:

```
CARDINAL(Pointer1)<CARDINAL(Pointer2)
```

Of course, this usage is not portable. For example, on the MSDOS large memory model compiler, the pointer will contain both an address and a segment register and simple comparisons are impossible since there are many different pointer values for any given address.

6 Statements

Statements are the constructs which actually do the work of your program. Consecutive statements must be separated by semi-colons. When a statement is followed by a reserved word which belongs to a surrounding statement, the semi-colon is optional. Modula-2 is much more lenient on the use of semi-colons than either C or Pascal. However, in C, the semi-colon is a statement terminator, rather than a separator between statements.

Furthermore, you cannot use a semi-colon at the end of a compound statement in C. C programmers may similarly be tempted to leave out the semi-colon after the end of a structured statement in Modula-2. Doing so may produce an error (depending on what follows).

Modula-2 will never complain that you have too many semi-colons, only too few. For example, in Pascal, the following statement would be wrong because of an extra semi-colon (after `a:=c`):

```
IF a=b THEN a:=c; ELSE a:=d
```

In C, this statement would be erroneous:

```
IF (a==b) a=c ELSE a=d
```

Because a semi-colon has been omitted (after `a=c`). In Modula-2 both the equivalent statements are legal and produce the same result:

```
IF a=b THEN a:=c; ELSE a:=d END
IF a=b THEN a:=c ELSE a:=d END
```

Unlike Pascal, Modula-2 does not require that you use `BEGIN...END` pairs around structured statements. In fact, such usage is not allowed. Rather, each structured statement (such as `IF` and `WHILE`) is terminated by an `END`.

Because it is the statements of a program which actually do the work, many people think that the statement forms are the most important part of a language.

They are mistaken: computing is about manipulating data. If you can get your data structures right, everything else will follow. The code will merely be a reflection of those structures.

A badly designed data structure is far more damaging than a swag of GOTO's: structured programming is data structuring. As you read the following descriptions of statement types, consider the types of data structure that each is most suited to.

6.1 Procedure Calls

Procedure calls have already been discussed in the section on procedure declarations (**Section 4.6**). See that section for examples.

A procedure call consists of the name of the procedure followed by a list of parameters in parentheses. The number and types of the actual parameters must agree with the number and types of the formal parameters. If no parameters are required, the parentheses should be omitted.

A procedure variable may be used in place of the procedure name. It may be any entire variable (**Section 5.1**).

6.1.1 The RETURN Statement

The return statement can be used to return from a procedure or a function. If it is used to return from a function, it must be followed by an expression of the type returned by the function. The value of this expression is the result value of the function.

For example:

```
PROCEDURE Ex1;
BEGIN
      ...
      IF i>10 THEN RETURN END;
      ...
      END Ex1;
```

```
PROCEDURE Ex2():INTEGER;
BEGIN
        IF i>10 THEN RETURN 10
        ELSE RETURN i END
        END Ex2;
```

Note that a function must always return by a return statement: it should never return by 'falling out the bottom'. If it does so, the returned value will be undefined.

6.1.2 Standard Procedures

There are several standard procedures in Modula-2. The standard procedures should not be confused with the standard functions: The procedures cannot be used in an expression.

These are:

`DEC(x,n)` subtract `n` from `x`. `x` may be any arithmetic type. `n` must be an integer or cardinal. The second parameter may be omitted (together with the preceding comma), in which case a decrement by one is performed.

`INC(x,n)` Like `DEC`, but for incrementing.

`INCL(x,n)` Adds the value `n` to the set `x`. `n` must be of the element type for the set. `n` can be an arbitrary expression. `x` must be an entire variable. (`INCL` is an abbreviation of *include*).

`EXCL(x,n)` Removes an element from a set.

Note that `EXCL(x,n)` is equivalent to `x:=x-{n}`, and `INCL(x,n)` is equivalent to `x:=x+{n}`. These two procedures are a hang-over from the original definition of Modula-2 which only allowed constants in set constructs, so they could be regarded as archaic. However, these procedures are more efficient than the corresponding assignment statements, and they are a standard part of the language. (`EXCL` is an abbreviation of *exclude*).

`HALT` Unconditionally terminates program execution. This procedure does absolutely no end-of-run processing, it simply aborts to the operating system.

`NEW(p)` Allocates storage for a variable of the type pointed to by pointer variable `p`, and places the pointer to that storage in `p`.

`DISPOSE(p)` Releases the storage for the variable pointed to by `p` and sets `p` to `NIL`.

`NEW` and `DISPOSE` are peculiar in that they are really short-hand forms for other procedures. `NEW` is expanded to:

```
ALLOCATE(p,SIZE(p^));
```

While `DISPOSE` is expanded to:

```
DEALLOCATE(p,SIZE(p^));
```

What is more, these routines (`ALLOCATE` and `DEALLOCATE`) must be imported into the module in which `NEW` and `DISPOSE` are used. They can be found in the standard module `STORAGE` but you are, of course, free to replace these implementations with your own. Generally, this would be done by writing a new module with a different name (for example, `MyStorage`) and then importing `ALLOCATE` and `DEALLOCATE` from there.

6.2 Assignment Statements

Assignment statements contain a left hand side and a right hand side separated by a *becomes* symbol (`:=`).

```
i:=2; p:=q;p^:=q^;a[2]:=3;
```

The left hand side must be an entire variable. The right hand side can be an arbitrary expression, but the two variables must be assignment compatible. Because the left hand side must be an entire variable, expressions like:

```
INTEGER(c):=-9;
```

are illegal, even though the INTEGER function simply changes the type.

Be sure you understand the difference between the assignment p:=q and the assignment p^:=q^, where p and q are pointers variables (see 4.3.8). The first assignment statement p:=q assigns only the pointer contained in q to the variable p. The second form assigns the area pointed to by q to the area pointed to by p. In the second case, the value of p remains unchanged.

Note that you can assign any compatible objects, not just elementary typed objects:

```
VAR a,b:ARRAY[1..20] OF CHAR;
      ...
      b:='Hello Salesman';a:=b;
```

Because Hello Salesman is only 14 characters long, and the character array it is being assigned to is longer than this, a null character (0c the smallest valued character) is appended to the end of the string. The remaining 5 characters of b (after the string and the null character) remain undefined.

6.3 Conditional Statements

The conditional statements allow one of several sets of statements (some of which may be empty) to be selected depending upon some condition.

The conditional statements are the IF statement and the CASE statement.

6.3.1 The IF Statement

```
IF <boolean expression> THEN
      statement;...
ELSIF <boolean expression> THEN
      statement;...
      (*possibly more ELSIFs*)
ELSE
      statement;...
      END (*IF*);
```

The IF statement performs the first of a series of alternatives for which the *boolean expression* is true. If none of the expressions are true, then the ELSE part is executed. If the else part is omitted and none of the alternatives are true, nothing is executed.

Each part may contain any number of statements (including zero) and each part (except the IF part) may be omitted. For example:

```
IF i<j THEN i:=j END;
IF j<k THEN
      i:=j;
ELSE
      i:=k;
      END
```

It is a good idea to always include the semi-colon before an ELSE, since when another statement is added:

```
IF i<j THEN
      i:=j
      k:=1;
      END
```

It is easy to overlook the absence of a (now required) semi-colon on the preceding line, as is the case with this example.

We can use any number of ELSIF parts in an IF statement. For example:

```
IF ch='A' THEN
      i:=1
ELSIF ch='C' THEN
      i:=2
ELSIF ch='E' THEN
      i:=4
ELSE
      WriteString(' bad character');
      WriteLn;
      END;
```

However, whenever you find yourself writing a string of ELSIFs, ask yourself if the same result could not be better achieved with a CASE statement.

The last example would be far better encoded as a CASE statement. It would be clearer and would even run a little faster.

6.3.2 The CASE Statement

A case statement contains a selecting expression which determines which of several variants are to be executed. Each of the variants is labeled with one or more constant values. The variant labeled with the value produced by evaluating the selecting expression is the one executed.

An optional ELSE part can be included which will be executed whenever the required value is not found in the list of constant labels. If the else part is omitted and the selecting expression has a value which is not to be found in any of the variants, then the statement is by-passed.

The selecting expression must have type INTEGER, CARDINAL, CHAR (or a subrange of these), BOOLEAN, or must be of an enumeration type. It cannot be REAL.

For example:

```
CASE ch OF
' ':skipblanks|
'a'..'z','A'..'Z':getalpha;|
'0'..'9':getnumber;
      IF ch IN CHARSET{'X','x'} THEN
            value:=hexvalue
          END
          |
ELSE WriteString(' bad character ');
      Write(ch);
      WriteLn;
END (*CASE*);
```

Note that we can denote all the values in a given range with the .. notation, so that the alternative with the labels '0'..'9' will be executed whenever ch is numeric.

Note also that each alternative is terminated with a bar (|), unlike in Pascal where only a single (possibly compound) statement can be present.

The semi-colon before the bar is entirely optional, as it was before the ELSE in the IF statement. Also, the bar before the ELSE is optional. We always put one in since it makes the code easier to read and modify.

6.4 Looping Statements

The looping statements are the WHILE, REPEAT, FOR and LOOP statements.

6.4.1 The WHILE Statement

```
WHILE <boolean expression> DO
      statement;
      statement;...
      END (*WHILE*)
```

The statements in the while statement are repeated until the boolean expression becomes false. If the boolean expression is false the first time it is evaluated, the statements are never executed. For example, the following code looks through an array for a particular value:

```
i:=0;
WHILE (i<=HIGH(a)) AND (a[i]<>j) DO
      INC(i)
      END;
```

Note the use of the *short circuit* evaluation of the WHILE condition. The array is accessed only if the variable i is in bounds.

6.4.2 The REPEAT Statement

```
REPEAT
      statement;
      statement;...
      UNTIL <boolean expression>;
```

The statements are repeated until the boolean expression becomes TRUE. The statements are always executed at least once.

REPEAT statements should be used with caution, since it is easy to overlook some condition which causes the statements to be executed when they should not be. Before using a REPEAT statement, consider if it can be conveniently replaced by a WHILE statement. For example, the following code skips until a numeric character is found. It will always read at least one character, and it makes no use of the value of ch when the statement was entered. Therefore, it may cause you to lose characters.

```
REPEAT
      Read(ch);
      UNTIL ch IN CharSet{'0'..'9'};
```

To produce the same effect with a WHILE statement, you would have to perform an extra Read in front of the loop:

```
Read(ch);
WHILE NOT (ch IN CharSet{'0'..'9'}) DO
      Read(ch);
      END;
```

Also, in the WHILE statement, the condition is the complement of the condition in the REPEAT statement, because a WHILE statement keeps looping while the condition is TRUE while a REPEAT statement loops until the condition is TRUE. The WHILE form may look more complicated but it has the advantage of allowing you to read text in which some items are optional.

6.4.3 The FOR Statement

```
FOR identifier:=expression
            TO expression
            BY expression DO
      statement;...
      END;
```

The FOR statement causes the enclosed statements to be repeated while the identifier takes on values between the values of the first and the second expressions in the header. The types of the expressions giving the range must be compatible with the type of the control loop variable.

The BY keyword and its associated expression may be omitted, in which case the identifier is incremented by one on each pass through the loop.

If included, the expression following the BY must be a constant expression. It gives the increment to be added to the identifier after each iteration. It may be negative.

For example:

```
FOR i:=1 TO 10 DO
      a[i]:=0
      END;
FOR i:=9 TO 1 BY -1 DO
      a[i]:=a[i+1]
      END;
```

As well as using INTEGER or CARDINAL ranges, ranges from enumerations can be used. For example:

```
FOR c:=Venus TO Mars DO ... END
FOR b:=FALSE TO TRUE DO ... END
```

The expressions denoting the range are evaluated at the beginning of the execution of the statement. They are not re-calculated at the start of each iteration. Hence the statement:

```
j:=10;
FOR i:=1 TO j DO DEC(j) END;
```

Loops 10 times, and j ends up with the value 0. Of course, doing this sort of thing is not recommended.

The identifier should not be altered within the statements under the control of the FOR statement. Nor should it be altered in any procedure called by those statements. However, this compiler will not detect such a modification, but such modification will not affect the number of times the loop is executed and, at the beginning of the next iteration, the variable will be set back to the value it would have had if the illegal modification had not taken place. As a result, the statement:

```
FOR i:=1 TO 10 DO
      WriteCard(i,6);
      i:=0;
      END;
```

Will print the values 1 to 10 and no more. Note that this fragment is not valid Modula-2, but that this compiler will not currently diagnose the error.

In standard Modula-2, the control variable (the variable before the becomes (:=)) symbol cannot be a component of a structured variable, a formal parameter, or an imported variable. This compiler will not detect such usage. The code will still execute correctly in these cases.

In the Z80 version of the compiler flag, there is a linker flag /F which changes the code generated for FOR loops. Using the flag allows large iteration counts (>32767) to be used at the expense of execution speed. See the Z80 User Guide for details.

6.4.4 The LOOP Statement (and the EXIT Statement)

The loop statement is a generalized form of the WHILE and REPEAT statements in which the exit condition can be placed anywhere in the loop. As with the REPEAT statement, you should only use it when really necessary.

In particular, watch out for LOOP statements in which the only EXIT is in an IF statement at the beginning of the loop. These should probably be recoded as WHILE statements. The code will work, but it is inelegant.

```
LOOP
      statement;
      statement;...
      END;
```

The statements are repeatedly executed until an EXIT statement is executed. The EXIT statement causes the innermost LOOP to be exited immediately.

```
LOOP
      IF id<l^.id THEN
           l:=l^.lower
      ELSIF id<l^.id THEN
           l:=l^.upper
      ELSE
           EXIT
           END (*IF*);
      END (*LOOP*);
```

The EXIT statement can only be used with a LOOP statement. It cannot be used with WHILE, FOR and REPEAT statements. If you use an exit statement within such a statement, the nearest surrounding LOOP will be exited and if none exists, an error will be produced. There is no way of exiting more than one level of LOOP statement from a single EXIT statement. In practice, you will rarely need it. On those few occasions, you can introduce a boolean variable which is set in the inner loop to signify that the outer loop should also exit:

```
ForceExit:=FALSE;
LOOP
      ...
      LOOP
           ...
           IF a=b THEN
                ForceExit:=TRUE;
                EXIT;
                END;
           ...
           END;
      IF ForceExit THEN EXIT END;
      ...
      END;
```

A slightly naughty but sometimes useful way of getting out of an inner loop is to use a RETURN statement. This exits the entire procedure immediately.

In this compiler, there are no restrictions on using EXIT statements within other structured statements. For example:

```
j:=0;
LOOP
      FOR i:=1 TO 10 DO
      IF j>50 THEN EXIT END;
            j:=j+i*i;
            END;
      END;
```

This entire construct terminates in the sixth iteration of the FOR loop. This example is a good example of how not to use the LOOP statement, since we are using the LOOP statement to simulate a GOTO statement (Since we never actually loop at all we always exit in the first time through).

This particular example could be much better written as:

```
      j:=0;
      i:=1;
      WHILE (i<10) AND (j<=50) DO
            j:=j+i*i;
            INC(i)
            END;
```

Replacing a FOR loop with a WHILE loop often produces cleaner code. If your first language was an old-fashioned BASIC, or even FORTRAN, you will probably find that you tend to use FOR loops excessively. If your first language was FORTRAN, you will also tend to use variables like i and j excessively, too!

6.4.5 The WITH Statement

The WITH statement gives you a way of accessing the elements of record variables without having to repeatedly give the name of the record variable.

```
WITH entire variable,... DO
      statement;
      statement;...
      END;
```

The entire variables are record variables whose elements are to be referenced. If there is a clash of identifiers, the innermost identifier (which is that of the last WITH statement) is used. Similarly, if any field identifier is the same as another identifier in the program, the field identifier is visible within the WITH statement, not the other variable. For example:

```
TYPE ExampleType=RECORD
          ch:CHAR;
          END;
VAR   ch:CHAR;
      r:ExampleType;
      p:POINTER TO ExampleType;
BEGIN
      ch:=' ';(*this is the VAR ch*)
      WITH r DO
          ch:='a';(*this is r.ch*)
          NEW(p);
          WITH p^ DO
               ch:='c'(*this is p^.ch*)
               END;
          END;
      END;
```

Be careful when using WITH statements with recursive types, such as trees. If you are working with multiple levels of the tree simultaneously, it is easy to get a variable from the wrong level:

```
TYPE  PTree=POINTER TO Tree;
      Tree=RECORD
               Left,Right:PTree;
          ch:CHAR;
          END;
      ...

      WITH Express^ DO
          IF ch='*' THEN
          WITH Left^ DO
               ...
               Left:=NewExp; (*Error*)
               ...
```

We wanted to evaluate the left subexpression of an expression, and then replace the left subexpression by its evaluated version. But the code at `Error` actually replaces the left subexpression of the left subexpression with the evaluated expression!

If the entire variable in the `WITH` statement is a pointer (eg `WITH Express^ DO`), and you change the value of the pointer within the `WITH` statement, the fields referenced in the remainder of the `WITH` statement are the same as before the assignment. However, you should not rely on this fact, as other compilers may work differently (particularly if they don't take advantage of the `WITH` statement to produce improved code).

The ability to place more than one entire variable in a `WITH` statement is an extension provided by this compiler. In the standard, only one entire variable is permitted, so using the multiple form may result in your programs being non-portable.

```
WITH a,b DO ... END;
```

is exactly equivalent to

```
WITH a DO WITH b DO ... END END;
```

It even generates the same code.

7 Syntax

The following pages contain a consolidated syntax for Modula-2. We have avoided including BNF in the preceding sections of the manual since it tends to frighten the beginning programmer, and because it is more use if gathered together in one place.

The *Extended Backus Naur* Form popularized by the description of Pascal is used. In this form, each *production* takes the form.

Left hand side = right hand side.

This means that the left hand side can be replaced by the right hand side as a step in generating a program.

To see if a program is a valid Modula 2 program, we start with the distinguished symbol *program* and keep replacing left hand sides by right hand sides until no more identifiers can be replaced. If a program is a valid Modula 2 program, replacing the left hand sides in a suitable fashion will produce your program.

At any intermediate point, we have a string of intermixed terminal and non-terminal symbols. The terminal symbols are those that cannot be replaced while the non-terminals are the symbols that have appeared on the left hand side of a production.

Fortunately, it doesn't matter which non-terminal in a string we chose to replace next (this is called the *Church-Rosser Property*), but it does matter what we replace it with, since any non-terminal will usually be associated with a number of alternative right hand sides.

A program which, given a grammar and a program, can choose the appropriate substitutions is called a parser. A good book to read to learn more about parsers and language processing generally is the 'New Dragon Book' (Compilers: Principles, Techniques and Tools, Aho, Sethi and Ullman, Addison-Wesley Publishing Company, 1986).

This isn't quite the whole story, since the syntax only specifies certain parts of the language. For example, it does not specify that the number of parameters in a procedure call must agree in type and number with the formal parameter list.

These latter requirements are usually referred to as the (static) semantics of the language. They can be specified by several extended methods based on BNF, but the results are difficult to comprehend.

To get used to using BNF, first try expanding out some simple constructs for simple statements. For example, you can try expanding `Statement` out to an `if` statement. Here are a few steps in one such derivation:

```
Statement
      ->IF Statement
      ->IF expression THEN Statements END
      ->IF Simpleexp relation Simpleexp
           THEN Statements END
      ->IF term relation Simpleexp
           THEN Statements END
```

At each step, we have replaced one non-terminal (as it happens, we have always replaced the left-most non-terminal, which is how the compiler does it).

In the right hand side, some conventions are used to save space and to make the syntax more readable.

Braces ({ and }) are used around a part of the right hand side which may be repeated zero or more times.

Brackets ([and]) are used around parts which may be omitted.

A bar (|) is used between alternative right hand sides. It may also be used inside brackets or braces to separate various possible choices.

Keywords (eg `MODULE`) are always in upper case. Lower case, or mixed case is used for symbols which are left hand sides of some production (that is, non-terminals). Symbols in quotes are actual marks that appear in a Modula-2 program.

So, here is the syntax (presented in width first order):

```
1 program = ProgramModule | DefinitionModule
2 ProgramModule = [IMPLEMENTATION] MODULE ident ';'{import}
                  block ident '.'
3 DefinitionModule = DEFINITION MODULE ident ';' {import}
      {definition} END ident '.'
4 import = IMPORT ident {',' ident} ';' |
                        FROM ident IMPORT ident {',' ident}
5 block = {declarations} [BEGIN Statements END]
6 definitions = CONST {ConstantDeclaration';'} |
                  TYPE {ident ['=' type]}';'} |
                  VAR {VarDeclaration ';'} |
                  ProcedureHeading ';'
7 declarations =  CONST{ConstantDeclaration';'} |
                  TYPE {ident '=' type ';'} |
                  VAR {VarDeclaration ';'} |
                  ProcedureDeclaration ';' |
                  ModuleDeclaration ';'
8 Statements = Statement {';' Statement}
9 ConstantDeclaration = ident'='ConstExpression
10 type = SimpleType | ArrayType | RecordType |
                  SetType | PointerType | ProcedureType
11 VarDeclaration = identlist ':' type
12 ProcedureHeading = PROCEDURE ident[FormalParameters]
13 ProcedureDeclaration=ProcedureHeading ';'block ident |
            ProcedureHeading ';' FORWARD
14 ModuleDeclaration = MODULE ident ';'{LocalImport}
                        [export]block ident
15 Statement = [ assignment | ProcedureCall | IfStatement |
                        CaseStatement | WhileStatement |
                        RepeatStatement | LoopStatement |
                        ForStatement | WithStat |
                        EXIT | RETURN [expression]
16 ConstExpression = SimpleConstExp [relation SimpleConstExp]
17 SimpleType = qualident | enumeration | SubrangeType
18 Arraytype = ARRAY SimpleType{',' SimpleType} OF type
19 RecordType = RECORD FieldSequenceList END
20 SetType = SET OF SimpleType
21 PointerType = POINTER TO type
22 ProcedureType = PROCEDURE [FormalTypeList]
23 identlist = identspec {',' identspec}
24 FormalParameters = '(' [FPSection {';' FPSection}] ')' [':' qualident ]
25 LocalImport = IMPORT ident {',' ident} ';'
```

```
26 export = EXPORT ident {',' ident} ';'
27 block = BEGIN Statements END
28 assignment = designator ':=' expression
29 ProcedureCall = designator[ActualParameters]
30 IfStatement = IF expression THEN Statements
                    {ELSIF expression THEN Statements}
                    [ELSE statements]
                    END
31 CaseStatement = CASE expression OF case
                    {'|' case}
                    ['|']
                    [ELSE Statements]
                    END
32 WhileStatement = WHILE expression DO Statements END
33 RepeatStatement = REPEAT Statements UNTIL expression
34 LoopStatement = LOOP Statements END
35 ForStatement = FOR assignment TO expression
                    [BY ConstExpression] DO
                    Statements
                    END
36 WithStat = WITH designator{','designator} DO Statements END
37 expression = Simpleexp [relation Simpleexp]
38 SimpleConstExp = ['+'|'-'] ConstTerm {AddOp ConstTerm}
39 relation = '<'|'<='|'='|'>='|'>'| '<>'|'#'| IN
40 qualident = ident {'.' ident}
41 enumeration = '(' ident {',' ident } ')'
42 SubrangeType = '[' ConstExpression '..' ConstExpression']'
43 FieldSequenceList = FieldList{';' FieldList}
44 FormalTypeList = '(' [ FormalTypes] ')' [':' qualident]
45 identspec = ident [ '[' ConstExpression ']']
46 FPSection = [VAR] ident {',' ident } ':'FormalType
47 designator = qualident {'.' ident |
                    '['expression {',' expression}']'| '^'}
48 ActualParameters = '(' expression {',' expression} ')'
49 case = [ CaseLabels ':' Statements ]
50 Simpleexp = ['+'|'-'] term {AddOp term}
51 ConstTerm = ConstFactor {MulOp ConstFactor}
52 AddOp = '+' | '-' | OR
53 FieldList = [ ident {',' ident } ':' type |
                    CASE [ident] ':' qualident OF variant {'|' variant}
            ['|'] [ELSE FieldSequenceList] END ]
```

```
54 FormalTypes = [ VAR ] FormalType {';' [ VAR ] Formaltype }
55 FormalType = [ ARRAY OF ] qualident
56 CaseLabels = CaseLabel {',' CaseLabel }
57 term = factor { MulOp factor }
58 ConstFactor = qualident | number | string | set |
                    '(' ConstExpression ')' | NOT ConstFactor.
59 MulOp = '*' | '/' | DIV | MOD | AND | '&'
60 variant = [CaseLabels':'FieldSequenceList ]
61 CaseLabel = ConstExpression ['..' Constexpression]
62 factor = number | string | set | designator [ActualParameters] |
                    '(' expression ')' | NOT factor
63 set = [qualident] '{' [element {',' element} ] '}'
64 element = expression ['..' expression]
```

Notes on the productions

The syntax given above describes the language as implemented by this compiler. It differs from the syntax in Wirth's book because of this, and because some changes have been made by Wirth himself since the book (Revision 2) was published.

3 A definition module no longer contains an export list. instead, all defined identifiers are exported.

6 The definition of a type is optional in a definition module. It is possible to simply declare it. This results in an *private* (or *opaque*) type.

12 Note that the (and) are not optional in a function declaration.

13 The alternative form FORWARD which is required because this is a one pass compiler.

14 The import statement for a module nested inside another module is limited to importing identifiers from the surrounding scope: you cannot import directly from global objects. We believe that this definition is what was intended by Wirth.

15 The entire right hand side is optional, so the null statement is acceptable.

18 The correct form of declaration for a multiple dimensioned array is (for example)

```
ARRAY [1..10],[1..8] OF ...
```

and that if an identifier is used instead of an explicit range, the brackets are omitted (unlike Pascal). Note also that it is possible to give a complete enumeration as an index type!

35 This compiler allows any designator to be the control variable in a `FOR` loop, so this production has been modified appropriately.

36 This compiler allows an extended form of the `WITH` statement similar to that of Pascal.

40 All the identifiers in a qualified identifier except the last one must be module identifiers.

8 Extensions and Restrictions

Here is a (we hope) complete list of the extensions and restrictions in this compiler. They are given together because what we consider an extension may be considered by someone else to be a restriction!

You should also check the distribution disks for a `README.NOW` file which may give further extensions and restrictions, or which may remove some existing ones.

You should also look at the section in your user guide which gives the physical (size) limits of the particular compiler you are using.

i) A declaration of a pointer to a type which has not yet been declared and which has the same name as a type in a surrounding scope, will not be correctly resolved. The pointer will point to instances of the type in the surrounding scope. For example:

```
TYPE Fred=INTEGER;
PROCEDURE a;
VAR a:Fred;
TYPE Fred=CARDINAL;
     ...
```

`a` will be an `INTEGER` when it should be a `CARDINAL`. This is because of the one pass nature of the compiler.

ii) The priority number cannot be used on the module heading. In standard Modula-2, using a priority number makes a module a *monitor*. (Actually, its a degenerate form of monitor since it inhibits interrupts rather than enabling a scheduler). Only one process may be in a monitor at a single time. However, on a single processor machine, the concept of a monitor is not required. If a module does have a priority number, it will be ignored, and a warning message `priority numbers not implemented` will be given.

iii) Sets may have up to 1024 elements and the bottom element does not have to be 0. Hence, sets of CHAR can be used.

iv) The type of a subrange is compatible with both INTEGER and CARDINAL provided that both its bounds are compatible with these types.

v) Variables may be given initial values statically by following the type with an equals (=) and the required value. The individual elements of an array may be initialized by enclosing the elements brackets. ([and]).

```
Int:ARRAY[1..5] OF INTEGER=[0,2,4,3,2];
Chars:ARRAY[1..10] OF CHAR='Hi There !';
TChars:ARRAY[1..3] OF CHAR=['T','H','E'];
```

This facility is only mentioned here because it is not fully implemented. In particular, no type checking is performed, so you can say:

```
Int:INTEGER=FALSE;
```

and also, no checking is performed to ensure that the list is the same length as the elements to be initialized.

vi) Full constant expression folding is provided. You can use SIZE and TSIZE in constants. Constant expressions using reals are permitted. This isn't really an extension, but we suspect that some compilers will skimp in this area.

vii) Hexadecimal character constants can be defined by following the hexadecimal value with x (e.g. 0ax is the character *line feed*). In standard Modula-2, only the octal notation (e.g 12c) is available.

viii) String literals are compatible with all ARRAY [m..n] OF CHAR types even though the low bound is not zero. Some compilers require that the low bound be zero. At this time, we regard this as a restriction in those compilers rather than an extension in this one.

ix) Private types are not limited to two bytes. However, see **Section 4.3.10** for details.

x) Processes share a display, which limits the places in which context exchanges can be performed. See the description of *Processes* in the user guide for details. This restriction has been removed from some versions of the compiler.

xi) Comparison of strings is supported. The ASCII collating sequence is used. The string is compared up to a zero byte (that is, the comparison routine obeys the zero byte terminator convention). If the shorter of two strings matches the leading substring of the longer substring, it is less than the longer substring.

9 Compiler Error Messages

The error messages are to be found in the file ERRMSG.DAT. Most of the messages are self explanatory. For others, some explanation can be found in the list below. In some cases, the explanation describes some of the more obscure cases in which the error can occur.

A good rule to remember when attempting to interpret errors is that an error is never caused by anything to the right of the pointer which indicates where the error has occurred. For example, if you get the message DO expected and you can see the relevant DO to the right of the pointer, it means that something under or to the left of the pointer has caused the compiler to look for the DO too soon. For example, in a FOR statement, you may have mis-spelled BY.

You will notice that some messages have more than one number (for example, 2 and 5) while some error numbers correspond to no error message. This is not an error in the documentation. This error message file started out as an error message file for a Pascal compiler and the one file still serves both compilers.

```
1 undefined identifier
```

Check that the identifier is spelled correctly. Recall that case is important.

```
2 doubly defined identifier
```

There are two (possibly contradictory) definitions of one identifier. Even if the definitions are identical, it is still an error.

```
3 This symbol cannot start a statement - elided
```

This error occurs when the compiler is trying to identify the type of statement it has just reached. The symbol is not one that can start a statement. Check that you really are at the start of a statement it may be that some preceding text is wrong.

Two common causes of this error are:

(1) You have omitted an END from a statement and the compiler has just found the identifier at the end of your procedure or module. However, if the procedure is a proper procedure with parameters you will get error 14 (too few arguments) in this situation.

(2) You have tried to use BEGIN and END for a compound statement, as in Pascal. Recall that Modula-2 only uses BEGIN for the beginning of the statement parts of procedures and main programs.

```
4 procedure name expected
```

```
5 doubly defined identifier
```

Identical to error message 2. Having multiple copies of some error messages was useful during debugging.

```
6 ; expected
```

```
7 procedure declared forward twice
```

```
8 terminating period omitted
```

If this message occurs before the genuine end of your program, you may have too many ENDs.

```
10 wrong type of module
```

Caused by trying to compile a definition module with the ordinary module compiler, or vice versa. It will also occur if the word MODULE is missing or mis-spelled. Compiling your documentation is a good way to get this message!

```
11 , missing between parameters or VAR misspelled (or wrong
case)
```

The compiler has found a symbol in a formal parameter list which it cannot process. A common cause of this is mis-spelling VAR (such as spelling it VAr).

```
12 too many actual arguments
```

A procedure or function has been called with more arguments than the procedure requires.

```
13 type mismatch
```

The types of a formal parameter and an actual parameter, are incompatible. See section 4.4 of this manual for details on type compatibility. This message can also occur on the header of a FOR loop if either of the bounds for the range are not compatible with the control variable.

```
14 too few arguments
```

There are fewer arguments in a parameter list than in the procedure definition.

This error can occur if you are missing an END from a structured statement and the compiler is attempting to treat the terminating identifier, which repeats the name of the procedure, as a procedure call.

```
15 ) expected
```

This error occurs if the compiler cannot continue while processing an expression. You may have too many opening parentheses, or it is possible that part of the expression is completely missing.

```
18 type conflict
```

The operands of an operator in an expression are not compatible, or an element in a set construct is not compatible with the type of the set, or the two sides of an assignment statement are not compatible.

```
21 } expected
```

A set construct is missing a closing }. Of course, this may mean that there is something in the set construct that should not be there.

```
22 pointer required
```

A dereference symbol (^) has been used with an expression that is not a pointer.

```
24 not array type
```

You have tried to subscript a variable which is not an array.

```
25 ] expected
```

The compiler expected to find the] at the end of an array subscript.

```
26 no such subfield
```

You have tried to reference a subfield in a record which does not exist.

```
27 not a record
```

The variable in which you have tried to reference a subfield is not a record.

```
28 arithmetic operand expected
```

A non-arithmetic operand has been used in an arithmetic expression. Some types, such as `WORD`, are not arithmetic types. For example, if you want to compare two variables of type `WORD` you can do `CARDINAL(a)<CARDINAL(b)` or `INTEGER(a)<INTEGER(b)` to give either a signed or unsigned comparison.

```
29 boolean expression expected
```

```
30 set type required
```

```
32 type conflict on operand
```

You can only take the negative of an `INTEGER`, `CARDINAL` or `REAL` operand.

```
33 ) expected
```

A closing bracket is expected at the end of a sub-expression, or at the end of a standard function.

```
34 identifier bad
```

The compiler has found an identifier at the start of a statement, but the identifier cannot start a statement. For example, trying to use a type transfer on the left hand side of an assignment (`INTEGER(i):=2`) would cause this problem. It can also be caused by losing an `END`, so that a `PROCEDURE` or a `BEGIN` becomes part of the current procedure.

```
35 cannot parse expression
```

The compiler has found an element in an expression which cannot possibly be part of an expression. For example, if you had an expression like `(2*)`, you would get this error on the second parenthesis `)`, because the compiler is looking for an operand.

```
36 invalid operands for MOD or DIV
```

The `MOD` and `DIV` functions require that the operands are `INTEGER` or `CARDINAL`. You have probably tried to use them with `REAL` operands.

```
38 types do not match
```

You have tried to compare expressions of differing types, or you have attempted to test for a value in a set of when the value has a type different from the elements of the set.

```
39 invalid pointer comparison
```

You can only compare pointers for equality and in-equality. If you really want to use other relations, type transfer them to `CARDINAL` first.

```
40 invalid operand type
```

This can occur when you attempt to compare operands which cannot be compared. For example, records cannot be compared.

```
41 OF expected
```

```
42 constant value or identifier expected
```

The compiler is looking for a constant and has found a variable.

A constant can be either a literal or it can be a constant identifier. Functions like SIZE and HIGH can also be used.

A constant identifier is an identifier which has been declared as a constant, or an element from an enumeration, such as red declared as part of colour=(red,green,blue).

43 : expected

44 END expected

46 record type required

This may occur in a WITH statement. A variable in the list is not a record variable. You may have omitted a pointer (^) after a pointer variable.

47 DO expected

The DO at the end of the header for a FOR statement or a WHILE statement is missing. If the DO exists but it has not yet been reached, there is something wrong with the expression being pointed to.

48 found = but assuming := was intended

49 found : but assuming := was intended

These last two errors are warnings only the compilation will continue correctly.

50 := expected

This message can occur if you use a variable as a procedure name in a procedure call.

51 size error

The left hand side of an assignment is of differing size to the right hand side. You should get another error message first. If you get this error by itself, please let us know. The types of the left hand side and right hand side do not match.

```
52 THEN expected
```

```
53 TO expected
```

```
54 UNTIL missing or END misplaced
```

You have closed a structured statement surrounding a REPEAT statement before the UNTIL for the REPEAT statement has been found.

```
57 identifier expected
```

```
58 = expected
```

```
59 identifier expected
```

```
60 OF expected
```

```
61 ; expected
```

```
63 incompatible types
```

The type of the high bound in a subrange is not the same as the type of the low bound.

```
64 high bound <low bound
```

The high bound of a subrange must not be smaller than the low bound. It can be equal.

```
65 identifier should be a type identifier
```

```
66 OF expected
```

```
67 [ expected
```

```
68 ] expected
```

```
72 = expected
```

```
73 : inserted
```

The compiler, while processing variable declarations, has found two consecutive identifiers and is trying to recover. There should, of course, be either a comma or a colon between the identifiers.

```
77 : found but assuming = was intended
```

You have used a colon when an equals was expected. For example `TYPE a:RECORD` instead of `TYPE a=RECORD`.

```
76 initialization error
```

You have tried to use the initial value extension of **FTL Modula-2** (e.g. `VAR i:INTEGER=2`) when the variable is not statically allocated.

```
78 type identifier expected
```

```
81 end of file reached before program complete
```

This commonly is the result of failing to terminate a comment. Recall that comments nest in Modula 2. Make sure that you have at least one more character after the terminating full stop (usually a carriage return), since the lexical analyzer, upon finding a period, looks for a second period in case the symbol is ...

```
82 output disk full
```

Either there is no more space on the output disk, or else you have run out of directory entries. Delete your `.BAK` files and try again. If you still get the message, you will have to re-organize your disks. If you are using the `T`, `R` or `U` compiler flags (see the user guide for descriptions of these flags) then the code file produced by the compiler is rather larger than the default.

```
85 Parameter list conflicts with earlier definition
```

This error is produced when the definition for a procedure which has been declared in the definition module, or declared as forward, is found and its parameters conflict with those of the original declaration. The parameters are not checked until the complete header has been examined, so you will have to examine the entire declaration.

If the parameter lists appear to be identical, it is possible that one of the types used in the header was imported from another module, and that modules definition module has been recompiled between the compilation of the current modules definition module and its implementation module.

```
86 ( expected
```

This error can occur if you use a type name in place of a variable name. This is easy to do if your type is called `Class` (say) and the variable is called `class`.

```
87 type of parameter inappropriate for standard function
```

```
88 procedure declared but never defined
```

Either you have declared a procedure in the definition module but it has never been defined in the implementation module, or you have declares a procedure to be `FORWARD` and then you have forgotten to define it.

```
90 import file not found
```

You have attempted to import from a module, but the `.SYM` or `.LMS` file for the module could not be found. The name of the `.SYM` or `.LMS` file is the same as the name of the module, with all characters converted to upper case and any characters after the eighth removed.

If the `.SYM` or `.LMS` file is on one of your disks, check that your responses to `SETSEARC` were correct.

```
91 IMPORT expected
```

You have omitted, or mis-spelled the word `IMPORT` following `FROM module-name`.

```
92 name not exported
```

You are attempting to import an identifier which does not appear in the referenced definition module. Check your spelling and that you have got the right module.

```
93 export not allowed from implementation module
```

You can only export from a nested module. EXPORT statements are not required in definition modules either. The definition module compiler will skip any such statements, giving a warning message (error 145).

```
95 FROM only allowed in outermost module
```

You cannot use the FROM module-name IMPORT ... form of the IMPORT statement in a nested module. Actually, if you import a full module (IMPORT Terminal) then it makes sense that you should be able to say FROM Terminal IMPORT WriteString in a nested module. However, this is not implemented yet see your README.NOW file if you want to make sure.

```
96 ; inserted
```

The compiler is attempting to recover at the end of a statement.

```
97 END missing from IF statement
```

```
98 END missing from WITH statement
```

Actually, the last two errors will rarely occur, because something else usually goes wrong first. Also, if an END is missing from a statement, the statement is terminated by an END which should have belonged to a surrounding statement. This works its way out until you hit a *hard* error, such as trying to use the identifier at the end of a procedure as a statement, which often results in error 14, or trying to treat the following PROCEDURE or BEGIN as a statement, which causes error 3, or hitting the end of a case variant, which also causes error 3.

```
99 EXIT statement is not within a loop
```

An EXIT can only be used with a LOOP statement. If you thought the EXIT was in a loop, check your ENDs.

```
100 LOOP statement nested more than 10 deep
```

You have hit a compiler limit. You will have to remove some of your inner LOOPs into another procedure. Ten deep loops is rather a lot. Should the code be re-structured?

```
101 END missing from LOOP statement
```

See the comments for error 97/98.

```
102 Terminating id not same as module id
```

Make sure that the error pointer is really pointing to the end of the program you may have too many ENDs again.

```
103 END missing from RECORD
```

See the comments for error 97/98.

```
104 TO missing after pointer
```

The declaration should be `POINTER TO something`.

```
105 END missing from module
```

See the comments for error 97.

```
106 not a definition module
```

Caused by trying to compile an implementation module with the definition module compiler.

```
108 END missing from case statement
```

See the comments for error 97.

```
109 Definition module could not be found
```

When you compile an implementation module, (that is, one which starts with `IMPLEMENTATION`), the compiler looks for the symbol file for the corresponding definition module. See also, error 90.

```
110 REL file could not be found
```

The code generated by the implementation module is appended to the end of the code generated by the definition module. In order for the compiler to do this, the `.REL` file must be on the disk to which you are sending the `.REL` file for this compilation.

```
111 RETURN not in procedure
```

You cannot use a RETURN statement in a main program part. Perhaps you wanted to use HALT.

```
112 END missing from FOR statement
```

```
113 END missing from WHILE statement
```

See the comments for error 97.

```
114 Body of procedure missing
```

The compiler has found a symbol near the beginning of a procedure which it cannot handle. This message should probably read BEGIN expected, since it belongs to the same class of error.

```
115 Ident missing at end of procedure
```

You must repeat the procedure identifier at the end of the procedure, so that the compiler can check the block structure.

```
116 Formal type not allowed here
```

A formal type is an open array. They can only occur in parameter lists. Evidently, the compiler does not believe this one is in a parameter list.

```
118 Character constant expected
```

Character constants are strings of length one, or constants using the special notations 0dx, 07c. (the quotes are not part of the construct)

```
119 types not assignment compatible
```

Either on an assignment statement, or when passing a parameter by value. Review the rules on assignment compatibility.

```
120 Attempted divide by zero when folding constant expression
```

```
121 Negative of CARDINAL constant not allowed
```

Because it would then have to be INTEGER. You could always use -INTEGER(2) for example.

```
122 Type of open array actual parameter is wrong
```

The element type of the actual parameter must match the element type of the formal parameter.

```
123 Open array not allowed as function result
```

You cannot declare a procedure as PROCEDURE Thing():ARRAY OF CHAR. The result must be of a known size.

```
124 BY expression must be integer or cardinal constant
```

```
125 BY expression must be a constant
```

You cannot use an expression for the BY expression in an IF. You can use an integer or cardinal constant with an enumeration range.

```
126 This expression cannot be reduced to a constant value
```

You have used an expression in a constant declaration, but the compiler cannot reduce the expression to a constant. For example, the declaration CONST SIN45=SIN(PI/4) would give this error, because the compiler cannot perform SIN functions this can only be done at run time.

```
127 Empty parameter list [ () ] assumed
```

Recall that a parameterless function declaration must have the brackets with nothing inside them.

```
128 At most 10 sets of static variables can be used in a module
```

This is a compiler limit. You have more than 10 VAR statements in a module. The compiler merges adjacent sets of VAR statements. You only get a new set of static variables if a procedure intrudes between too VAR declarations. To overcome this problem, put some of your declarations in the same section of code.

129 Constant in set construct out of set bounds

130 Constant too large - must not exceed low bound+1023

This is a compiler limit you cannot have more than 128 bytes in a set constant, so the maximum element is 1023 greater than the low bound of the set.

131 function required

You have tried to use a (proper) procedure as a function.

132 variable is not a procedure variable

You have tried to call a procedure using a variable which is not a procedure variable. This can be caused by using round brackets ((and)) instead of square brackets ([and]) for a subscript.

133 function should not be called as a procedure

134 type section header, PROCEDURE BEGIN or END expected

The compiler expected the start of a major section of code, but the symbol it found was not the start of such a section.

135 octal constant bad

You have used a non-octal digit in an octal constant. For example, 08c.

136 invalid base type for set type

You can only have sets of subranges, enumerations (including BOOLEAN) and CHAR (or subranges thereof).

137 more than 8 open array value parameters in a procedure

This is a compiler limit. Pass some of them by VAR or reduce the number of parameters. We would have severe doubts about any procedure which had eight parameters in total, never mind eight value open array parameters.

```
138 constant identifier used when variable identifier
required
```

You cannot assign to a constant.

```
139 string too long
```

The maximum length for a string in this compiler is 128 characters.

```
140 string not closed before end of line
```

The closing quote must be on the same line. Strings cannot span lines.

```
141 Variable is not pointer
```

You have tried to dereference (using ^) a variable which is not a pointer.

```
142 Character value invalid
```

You have used the `07c` or `0fx` construct with a value which is invalid. A common cause of this error is forgetting that the value must be in octal or hex. For example, the constant `255x` would cause this error you should have written `0ffx`.

```
144 Priority numbers not supported
```

This is a warning. The module will compile correctly, but the priority number is ignored.

```
145 Export in definition module is old Modula-2 - export
ignored
```

The modified definition of Modula-2 states that all identifiers in a definition module are exported. This is a warning message. The compiler will skip the statement and continue.

```
146 out of memory - use /S or reduce imported modules
```

The module is too large. Try to reduce the number of imported modules. If you are using the Z80 version, try using the `/S` compiler flag to free up an extra 1500 bytes for the symbol tables.

If you are importing a large module from which you only use a few identifiers, try importing those identifiers in the definition module of the module which is giving trouble instead of the implementation module; it saves space. Unfortunately, it also makes this modules definition module larger, which can cause a module which imports this module to give `out of memory`.

`147 Constant out of range`

This error is only detected when you compile using the `/R` flag. It occurs when you use a constant that is not in the range required by an associated type. For example, using an array subscript which is out of bounds, or assigning a constant to a subrange when the constant is not in the subrange, will cause this error.

`151 Version numbers inconsistent`

When you import a symbol file during compilation, the symbol file will usually contain references to other symbol files. If you then import another of these symbol files - either explicitly or implicitly because it is referenced by yet another module (eg many definition modules reference SYSTEM so you can end up importing SYSTEM implicitly a number of times), there is the possibility that things have been recompiled at different times and that two references to a given symbol file are two different recompilations of the definition module.

`152 Relocatable file bad or missing`

This message will appear when you attempt to compile an implementation module for which the relocatable file is either missing or it is shorter than it should be. (this means that the file has been clobbered or the compiler found an old version or some-such.) When you compiler a definition module, the compiler saves, as part of the symbol table file, the number of bytes written to the relocatable file. The implementation module compiler starts writing to the relocatable file at this point.

In addition to the above error messages, you may get the error message:

```
no source file
```

The file to be compiled was not found. In some versions(eg CP/M 80), you must include the extension with the file name. For example, `m2 fred.mod` rather than just `m2 fred`. The latter would compile from a file with no extension.

10 Linker Error Messages

The linker can produce a number of error messages. In addition to those below, there are some which contain a question mark followed by some numbers. These should never appear. If they do appear, recompile the module which was being linked at the time and try again. If the problem persists, send a copy of the module to us for examination.

`output disk full`

There is no more space for the executable file on your output disk.

`fill over`

This message should never appear. It is caused by the address patching table becoming full, but this table is always flushed when it becomes full. If this message occurs, we want to know about it.

`too many labels`

There is a limit of two hundred labels in any procedure or main program part. In addition, one label is used for each procedure, so that slightly less than 200 labels will be available to any given procedure. To overcome the problem, divide the procedure into several procedures. This will always be possible, since you can place some of the code in a procedure nested within the current procedure, which will allow it to access all of the current procedures variables (though it will not be able to do a `RETURN` for the current procedure).

`literal number too large`

There is a limit of 200 string, real and set literals in a module. As well as the obvious literals, every set constructor produces a set literal. To overcome this problem, you will have to divide your module into several modules. Note that an imported literal is not included in the number of literals for this purpose.

```
literal pool overflow
```

There is too much literal text in a procedure. At the start of a module, 1200 bytes of literal space is available. At the start of any procedure, at least 600 bytes of space will be available, since if the table is more than half full at the beginning of a procedure, it is flushed. To overcome the problem, split the offending procedure into several.

```
too many imported modules
```

There is a limit of 50 imported modules for any module.

```
bad code file - abandoned
```

The code file for the current module was bad. Recompile the module. This can occur if an `out of memory` error occurred during compilation. It can also occur if the compiler has failed to detect an error. If this occurs please tell us about it. It is possible that the linker will not detect that a code file is bad. In this case, the errors mentioned above with question marks in them may occur, or the linker may loop. However, these occurrences are rare.

```
extra com file
```

Your command line syntax is wrong. There can only be one name to the left of the equals sign. Perhaps you have entered more than one equals sign.

```
<name> not found
```

The module's relocatable file (`.REL`/`.SMR`/`.LMR`) could not be found. If the file does exist, check that you have set your search list correctly with `SETSEARC`.

```
circular refs - arbitrarily selecting <name>
```

There is a set of modules which produce a circular set of imports. For example, `A` imports `B` which imports `C` which imports `A`. This is detected by the linker when it is trying to determine an order for the execution of main program parts. One of the modules will be selected, and its name given.

```
<name> recompiled since <second name>
```

The first name is the name of a module imported by the second module. The definition module of the first module has been recompiled since the second module. This is a warning error only, the produced executable file may still work. However, it is quite possible that it will not. As a result of this error, the resulting program may mysteriously return to the operating system, it may call wrong procedures, or it may simply disappear, requiring a re-boot.

In other words, ignore the error at your own risk.

```
implementation module not compiled
```

The given module (the one being linked) has had its definition module recompiled without the implementation module being compiled since.

```
errors in compilation
```

The compilation produced errors. You must fix the errors and recompile before the module can be linked.

11 Compiler Limits

There are a number of limits on the use of the compiler. Provided these limits are not exceeded, quite large modules can be compiled. This is unlike some compilers, which use up some memory for every line compiled so that at most two or three hundred lines of code can be compiled. As these limits may change from time to time, check the `README.NOW` file for any alterations.

You cannot nest procedures, modules and `WITH` statements more than ten levels deep. This figure is *inclusive* not ten for each.

You cannot have more than 1200 bytes of literal text in a procedure, and should limit yourself to 600 bytes in a procedure. 600 bytes works out at about 10 long `WriteString` statements. There is a literal text buffer in the linker which is used to reduce the amount of memory used for literals. For example, if two literals are the same, then the same memory is used for both of them. This buffer is flushed whenever it is more than half full at the end of a procedure.

You cannot have more than 200 labels in a procedure. In fact the limit is a bit less than 200, because each procedure uses up one label for its entry point, and because a few labels are used by the code generator. All labels used in a procedure (except the one associated with the procedure's entry point) become available again at the end of the procedure. Labels are required as follows:

`WHILE`	1
`IF`	2 plus 1 for each `ELSIF`
`FOR`	2
`CASE`	1 plus 1 per alternative
`REPEAT`	1
`LOOP`	1

Only one label is used up per procedure as any labels used inside the procedure are made available again. That is, all the local labels only exist for the duration of the procedure.

You must not import more than 50 modules into any one module. You will be hard pressed to import 50 modules before you get `out of memory`.

12 A Comparison of Modula-2 and Pascal

This section describes many of the differences between Modula-2 and Pascal. This section is intended for the program who is experienced in Pascal and who wants a quick introduction to Modula-2.

Some subtle differences in the languages are not mentioned. If you are interested in such differences, see H.A. Muller's article 'Differences between Modula 2 and Pascal' (SigPlan Notices Vol 19 No 10, Oct. 84, pp32-39).

12.1 Lexical Differences

The lexical level of a programming language refers to the marks that you make on a piece of paper (or a terminal screen) to represent the program. At the lexical level, we are concerned with questions such as what are the key words, what characters are allowed in identifiers, and so on.

Modula-2 requires keywords to be entirely upper case. Case is significant in identifiers. In Pascal, case is ignored. In this Modula-2 compiler, up to 32 characters of an identifier are significant. In many Pascal compilers, only the first eight characters of an identifier are retained.

This means that, in many Pascal compilers, it is possible to have identifiers that look different, but are really the same since the first eight characters are the same. With the 32 character limit used by this compiler, this problem is far less likely to occur. On the other hand, in Pascal, you can use upper or lower case for identifiers and key words, and, in fact, it is considered good style to use primarily lower case. Modula-2 is sensitive to case, so that `Fred` and `fred` are different identifiers.

Pascal requires that character constants and strings be enclosed in single quotes. It allows the use of two consecutive quotes in a string to represent a single quote (so `''''` is a single quote character constant). Modula 2 allows either single or double quotes but does not allow the delimiting type of quote in the character string.

For example, the single quote would be written as "'" in Modula-2.

This compiler, however, has an extension which makes this possible. You must turn the extension on with a pseudo-comment (e.g. `(*$a^*)`). See section 3.3 of Part I for details.

Modula-2 has a facility to allow character constants to be specified in octal and, as an extension in this compiler, in hexadecimal. For example, `0ax`, `10c` are both the line feed character.

Pascal allows comments to be enclosed in `(*` and `*)` or in `{` and `}`. Pascal comments cannot be nested. Modula 2 only permits the use of `(*` and `*)` but allows comments to be nested. In Modula-2, `{` and `}` are used for sets.

12.2 Declaration

Declaration parts can be in any order and may be repeated in Modula-2. Pascal requires that the declarations be in the order `CONST`, `TYPE`, `VAR`, and none of them can be repeated.

In Modula-2, a variant part in a record declaration requires a separate `END`. In Pascal, the `END` for the record declaration also terminates the variant part. Modula-2 allows multiple variant parts.

Modula-2 uses the syntax `POINTER TO type` to declare a pointer type. Pascal uses `^type`.

In Modula-2, in an array declaration, the brackets (`[` and `]`) are part of the subrange declaration, not part of the array declaration. They are omitted if an identifier is used instead of an anonymous subrange and must be repeated if multiple index types are declared. For example:

```
ARRAY COLOUR OF INTEGER;
ARRAY [1..10],[1..10] OF REAL;
```

In Pascal, the brackets are part of the declaration and multiple dimensions are separated by commas within a single set of brackets.

Modula-2 permits constant expressions wherever a constant is required. Pascal only permits a simple constant or constant identifier.

Modula-2 does not use the keyword FUNCTION. Instead, a function is declared using the PROCEDURE keyword. In this case, the type is given after the parameter list, just as it is in Pascal.

In formal parameter lists, Modula-2 permits an open array declaration. An open array is an array with no index type, as in ARRAY OF INTEGER. The index type is determined when the procedure is called. This allows such procedures to be used with array variables of differing sizes.

12.3 Expressions

Pascal uses [and] around set constructs. Modula-2 uses { and }. In Pascal, all sets are the same size. Modula-2 allows sets of various sizes, since the type of a given set can always be determined from the set constant.

In Pascal, a character string of length one is of type CHAR. In Modula-2, it may be either CHAR or ARRAY[0..0] OF CHAR. This means that you can use a string constant containing exactly one character as either a character constant or a string constant.

Modula-2 allows a string of length zero. Pascal does not.

Modula-2 requires that parentheses be used in a parameterless function call. Omitting the parentheses gives the address of the function, not the value returned by a call of the function. Normally, this will produce a type mismatch when the program is compiled. Pascal requires that the parentheses be omitted.

There is no concept of a procedure variable in Pascal. In Modula-2, a procedure variable is used to contain the address of a procedure or function so that the procedure can be called indirectly.

12.4 Statements

All structured statements in Modula-2 may take more than one statement in their range of control and are terminated by a keyword, usually END. Pascal structured statements (except REPEAT) take only a single statement in the range of control, though this may be made into a compound statement with the use of BEGIN...END.

Modula-2 does not permit the use of BEGIN and END to produce compound statements. Nor is it necessary.

Modula-2 has an explicit RETURN statement which can be used to return from a procedure and must be used to return from a function. In the case of a function, the RETURN statement must include an expression giving a returned value. In Pascal, functions and procedures return by falling out the bottom of the procedure and the returned value is denoted by assigning it to the function name in an assignment statement. Modula-2 permits proper procedures (procedures which do not return a result) to return by falling through or by RETURN.

The Modula 2 IF statement allows multiple ELSIF parts. Pascal lacks this construct.

The Modula 2 CASE statement permits an ELSE part. Pascal lacks this construct. The case labels in Modula 2 may be subranges. Pascal requires simple constants. In Modula-2, BEGIN and END are not used in a case statement. Instead, the symbol | is used to terminate an alternative.

Modula-2 lacks a GOTO statement. Instead, a more general form of LOOP statement and a RETURN statement have been added. These two new constructs cover most uses of labels and GOTOs in Pascal and are easier to validate.

Modula-2 permits a step size in a FOR statement. A negative step is used instead of Pascal's DOWNTO option.

Pascal has a set of input-output facilities built in and allows automatic parameter substitution from the command line. Modula-2 requires the use of modules to perform IO and the programmer must handle parameter substitution.

12.5 Separate Compilation

Modula-2 supports separate compilation. Source code is divided into modules. Each module can have two parts, a definition part and an implementation part, or it may simply have the second of these parts. The definition part declares all objects which are defined in the implementation part and which can be accessed from outside the module. Anything declared in the implementation part is not visible outside the module. The definition part acts as a prefix to the implementation part.

The IMPORT statement is used to import objects from one module into another.

13 Revisions and Amendments

On November 21, 1983, a meeting was held with some implementors of Modula-2. Numerous features and facilities were proposed for addition or correction. The following were agreed upon.

This compiler implements the language as modified by this list. For this reason, if you are converting programs from an old compiler, you will notice some differences between what your old compiler accepts and what this compiler accepts.

This list is not a complete list of the changes and clarifications. Only changes that affect this compiler, and which may cause your existing programs to fail to compile are noted.

i) The types of a formal VAR parameter and that of its corresponding actual parameter must be identical, not merely compatible, except for formal parameters of type ADDRESS, which will accept any pointer as its actual parameter, and in the case of WORD, whose compatible types are implementation dependent.

ii) The types of expressions used for the starting and limiting values of a FOR loop must be compatible, not simply assignment compatible, with the control variable.

iii) The explicit export list in a definition module is discarded. All objects in a definition module are exported. This compiler ignores any explicit export list, giving a warning message.

iv) When the discriminant of a case selector in a record declaration is omitted, the colon must be included. For example:

```
TYPE T=RECORD
     CASE :BOOLEAN OF
          TRUE:a:REAL|
          FALSE:b:INTEGER|
          END;
```

Note also that, in case statements, an extra | is now permitted between the last alternative and the END or the ELSE. This is demonstrated in the example above. This change extends to CASE statements in the statement part as well.

v) The type PROCESS has been deleted. It is replaced by the type ADDRESS. In this implementation, the optional procedure NEWPROCESS and TRANSFER are to be found in the module Processes rather than in SYSTEM.

vi) There are two new standard functions MIN and MAX. These return the minimum and maximum values which the type given as a parameter can take.