GARDENS POINT MODULA
Users Guide
Reference Manual

April 1995

About this manual

The material in this manual was originally prepared by the following persons:

Michael Roggenkamp (editor)

John Hynd

John Gough

This 1992 update has been edited by John Gough, and contains details of the entirely new version of **gpmake**.

The manual was prepared camera-ready, using the LaTeX document processing system.

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

The date of this version of the manual is

February 1993 (*generic version*)

About gardens point modula

gardens point modula is a product of the programming language and systems group of the **Queensland University of Technology**.

Responsibilty for design and implementation of current versions is as follows:

John Gough (system architecture)

John Hynd (static semantics)

Diane Corney, Christina Cifuentes and Peter Kolb (the gpm-pc back-end, interpreter and bootstrap libraries)

John Chalk and Diane Corney (gpmake)

Michael Roggenkamp (some library modules)

John Hynd is project manager.

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

Contents

Introduction

gardens point modula

gardens point modula is a new implementation of Modula-2 for 32-bit *UNIX* machines. The implementation for the Hewlett-Packard HP9000/8*xx* has been available since October 1989, with the DECstation, **mips** and Silicon Graphics *IRIS* versions available in first quarter of 1990. The combination of Modula-2 and *RISC* architecture brings unprecedented power to the software developer.

gardens point modula provides a uniform Modula-2 programming environment in multi-vendor networks, and provides unrivaled portability between machines. The implementation is based on the emerging ISO draft standard for the Modula-2 language and its libraries, and is fully integrated with *UNIX*[1] standards.

gardens point modula provides the following features

- a fully type-checked, safe programming environment based on sound software engineering principles
- safe separate compilation based on pre-declared interfaces, data hiding and data abstraction
- comprehensive compile-time diagnostics, with explicit error messages, and optional warnings for obsolete syntax or dubious program constructs
- rigorous version checking of symbol and object files during compilation, and during load-module building
- seamless integration with standard *UNIX* tools such as **prof** and **dbx**, and safe interfaces to libraries such as **curses**
- extensive runtime checking is standard, with both command-line and embedded-pragma control of checks if desired
- high quality, fast code

The system consists of the compiler **gpm**, the load-builder **build**, libraries, and various utilities.

[1] *UNIX* is a registered trademark of AT & T

Introduction to the documentation set

The documentation for **gpm** consists of the following documents —

- **Language Reference Manual**
- **Library Definitions Reference Manual**
- **Version Release Notes** for each version
- **User Guide and Technical Reference Manual** (this manual)

How to use this manual

This manual is intended as a reference manual for writing application programs in Modula-2. It is not intended as a tutorial for beginning programmers, nor does it discuss systems programming or advanced techniques. This manual is divided into two parts: Part I, **User Introduction** comprises Chapters 1–6. Chapters 7–15 make up Part II, **Technical Reference**. All users should read Part I while users will choose from Part II the various features of the compiler that they wish to understand at a more technical level. The **Library Definition Parts** is essential reference material for all users, and is moved in this release to a separate manual. Chapter 6 contains handy syntax diagrams for Modula.

All of the material in chapters 1 – 15 applies to **gpm** as it is implemented on **all machines**. The material which is specific to particular implementations is in the appendices which follow chapter 15. Relegation of this material to the appendices does not imply that it is unimportant for most users. All of the material on profiling and use of the debuggers is machine dependent, and is thus found in these appendices.

Notational Convention

This manual uses the following notational conventions :

Convention	**Example of Convention**	**Description of Convention**
typewriter type	`Examples`	This typeface is use to simulate the appearance of screen output
bolding	**KEYWORD**	Bold letters indicate keywords or program names.
italics	*filename*	used for emphasis or to indicate a filename, a module name or a procedure name.
brackets	[*options*]	optional items listed between brackets

braces and vertical bars	{choiceA|choiceB}	you have a choice between two or more items
ellipsis	...	following an item indicate that more items having the same form may be entered

Part I

Users Guide

Chapter 1

Getting Started

1.1 The Program Development Cycle

The Program Development Cycle for an application program written in Modula-2 is summarized below :

1. Use a text editor to create or modify the source modules. Source modules can be organised in a variety of ways. It is assumed the normal convention of writing separate definition and implementation modules is used in the development cycle.

2. Use **gpm** to compile each of the modules of the program. If compilation errors are encountered in a module, you must go back to Step 1 and correct the errors before continuing. For each definition module (**.def**), **gpm** creates a symbol file (**.syx**). For each implementation module (**.mod**), **gpm** creates a reference file (**.rfx**) and an object file (**.o**[1]). Optional listings (**.lst**) can also be created during compilation.

3. Use **build** to create a single executable file

4. Debug your program if logical or run-time errors are encountered when the program is executed. It will be necessary to return to step 1 to correct one or more source modules.

1.2 Developing Programs

This section takes you through the steps involved in developing programs. Examples are shown for each step.

1.2.1 Writing and Editing Modula-2 Source Code

Modula-2 programs are created from one or more source files. Source files are text files that contain definition or implementation modules.

The following example, with apologies to B. Kernighan and D. Ritchie, is the first program to write when learning a new programming language. The definition and implementation modules have been combined into a single program module called *hello.mod*.

[1] In the case of **gpm-pc** the object file has the extention **.obj**. There are other obvious changes to the material in this chapter which apply to the pc version. For example, the DOS command `dir` must be substituted for the *UNIX* `ls` command

```
MODULE Hello;
  FROM InOut IMPORT WriteString, WriteLn;
  CONST str = "Hello world";
BEGIN
  WriteString(str); WriteLn;
END Hello.
```

1.2.2 Compiling Source Files

Source modules are compiled with the comand **gpm**. Its command line syntax is

```
gpm  [-options]  filename
```

Assuming you have the source file *hello.mod*, the simplest command line is

```
gpm hello.mod
```

The output would be an object file, *hello.o*, and a reference file, *hello.rfx*.
To compile the same source file with the maximum amount of information being produced, use the folllowing command line :

```
gpm -glV  hello.mod
```

The version date and time of the compiler are displayed, all messages are verbose, a list file, *hello.lst*, is generated and the code is compiled with markers for the symbolic debugger (either **dbx** or **adb**). Compiler options are discussed in detail in Chapter 2.

1.2.3 Building an Executable File

The command line syntax for the load-builder is

```
build  [-options]  basename
```

Having compiled *hello.mod* as above, an executable file is produced by the command :

```
build hello
```

The program can now be executed with the command `hello`

1.2.4 Use of Library Modules

A common complaint from programmers who learn Modula-2 after some other high-level programming language is the apparent lack of general purpose input/output facilities, for example the *read* and *write* procedures of Pascal. The decision not to include such facilities in the language means that Modula-2 is an *extensible language* and in order to use it effectively a programmer needs to know how to access the library modules, what facilities are provided by a library module and also the means of providing new features when desired.

Library modules consist of two parts — the *DEFINITION* part, which specifies the services provided, and the *IMPLEMENTATION* part , which specifies how those services are provided.

The **.def** files for library modules may be found in the directory of the environment path **$M2SYM**. If this path consists of a single directory, a directory listing of these files may be obtained by the command `ls -l $M2SYM/*.def`

Chapter 2

Using gpm

2.1 Compiling a program module

A program module is the single simplest compilation unit containing all declarations and program statements with perhaps an importation from one or more modules. The example program, *hello.mod* of Chapter 1 is a program module.

2.2 Compiling a definition module

A definition module (**.def**) is a separate compilation unit defining the interface between a module and its environment. Its contains all the information that the compiler needs to verify that another module is correctly using the facilities provided by this module. The command

```
gpm myfile.def
```

will produce a symbol file, with extension **.syx**. A definition module must be compiled before the corresponding implementation module. The definition module should also be compiled before any program or module that imports objects from that module.

2.3 Compiling an implementation module

An implementation module (**.mod**) contains the procedure bodies and initialization statements together with any other hidden procedure or data declarations. The command

```
gpm myfile.mod
```

will produce an object file, with extension **.o**, and a reference file, with extension **.rfx**. Since an implementation module depends on its own definition module and those other definition modules from which it imports some facility, an implementation module cannot be compiled until all such definition modules have been compiled. During compilation, symbol files for imported modules are accessed, looking first in the current directory and then on the directories of the path **$M2SYM**. All module names are moved to lower case before any extension is appended. If necessary, names are truncated to 80 characters in length, not counting the extension.

2.4 Using gpm's options

gpm provides many options to control the operation of the compiler and the format of messages produced during compilation. The syntax for options flags follows closely the style used by *UNIX* commands.

gpm -*options ... filename ...*

Options are described in the following sub-section. Options may occur in any order, and may be given separately or in a single group. A detailed description of each option is given in the Chapter *Compiler Options* in the technical reference section.

2.4.1 Compiler Option Flags by Functionality

Listing and message control

–d	suppress console warning messages
–l	produce listing file
–v	produce verbose messages and listings
–V	version date is displayed, module key values are traced, and verbose messages and listings are produced
–X	more detailed information is provided for each error

Runtime error checking

–a	suppress runtime assertion tests
–i	generate code without array index checks
–r	generate code without value range checks
–s	runtime stack tests are turned off
–t	runtime arithmetic overflow checks off

Compiler control flags

–f	force compiler to produce reference and object files with names based on the filename rather than the module name
–g	compile with extra information for the debugger program (*xdb* or *dbx*)
–n	compilation checks only, no object code is produced
–I	run in interactive mode, with option of jumping to the editor
–Oc	optimise for compactness and speed
–Of	optimise for maximum speed
–p	compile with procedure call-count code for the runtime profiler
–S	intermediate code is persistent

2.4.2 Using the Interactive Option

One of the most useful options when modifying programs is **–I**. This allows an easy alternation between **gpm** and **vi** or any other editor.

When an error occurs, and the interactive option is in force, the user is informed of the error, and is given the *verbose* version of the diagnostic message following up to four lines of context. The user is then asked to choose between continuing to the next error (if there is one), requesting more information about the error, abandoning compilation, or jumping straight into the editor[1].

```
$ gpm -I foo.mod
254      (* do a swap *)
255      temp := a[i];
256      a[i] := a[j];
257      a[j] := temp,
**** ............... ^ syntax error 105
**** 105 Expected semicolon ****
<enter> to continue,'m' for more info,'q' to quit,'v' to edit :
```

If the user enters 'v', **gpm** terminates and **vi** starts, with the cursor already on the offending line. For most errors this means that the file is already positioned for the user to correct the error with a minimum of effort.

After the error is corrected and **vi** is terminated, **gpm** automatically resumes. In the example, the comma is changed into a semicolon, and the user types 'ZZ'

```
gpm: recompiling <foo.mod>
$
```

gpm announces that it is recompiling. If there are further errors detected, the alternation between **gpm** and **vi** may be repeated.

When the interactive option is used, **gpm** only pauses and prompts the user when it detects *errors*. It does not bother the user with *warning* messages. In the absence of errors, the use of the '**–I**' option is similar in effect to the use of the '**–d**' option.

If the **m**ore response is selected, the compiler will print out any further information on the particular error. The messages are based on the information in the chapter *Interpreting error messages* of this technical reference manual. In the case of errors which have to do with type mis-matches, the two particular types are listed. Here is an example —

```
C:\GPM\WRK> gpm -I compare.mod

  12     WriteString("> ");
  13     ReadString(vst);
  14     val := Compare(vst,str);
  15     CASE val OF
  16       | friday : WriteString("equal");
 ****   ......^ SemanticError #  207
 **** 207 Expression is not compatible with declared type ****
<newline> to proceed,"m" for more info,"v" go to editor,"q" to quit: m
```

If the **m**ore option is selected, the following extra information is obtained on the screen —

[1]The particular editor may be chosen by an environment variable, and current versions allow either **e** for editor or **v** for vi to be used synonymously

```
---- More info. ----
Modula enforces strict agreement between types of expressions and
the context in which they are used. This error occurs if a label
of a CASE statement branch does not match the selector type, an
element in a set constructor does not match the set type, a bound
of a subrange does not match the host type or the other bound, a
record variant label does not match the tag type, or an array
index does not match the index type.
--------------------------------------------------------------------
The expected type is <StdStrings.CompareResult>
while the actual type is <Compare.DaysOfWeek>
--------------------------------------------------------------------
<newline> to continue,"m" for more info,"v" go to editor,"q" to quit: _
```

Note that the "more info" explanation is also obtainable on the screen by using the **–X** option, or in the listing file by using **–lX**. However, the *type* information is only available to the interactive mode, as the information is volatile, and is lost if compilation proceeds.

2.4.3 Inline Compiler Options

A number of the forementioned compiler options can be inserted in the **IMPLEMENTATION** part of the source program. These inline options must appear within the comment symbols (∗ ∗) .

$C+	produce **c**ompact code, even at the price of small speed loss
$F+	produce **f**ast, even at the price of larger memory size
$I-	array **I**ndex tests are turned off
$R-	**R**ange checking of assigned values and actual value parameters is suppressed
$S+	**S**tack overflow checking is turned on

Chapter 3

Using build

3.1 Building an Executable File

The load-builder program, **build** accepts the name of the base (main) module (with the name moved to lower case). It opens **.rfx** files as needed to calculate which other modules need to be linked, and the order in which they should be initialized. This program produces an executable file with no extension.

The **.syx**, **.o** and **.rfx** files produced by **gpm** take their name from the module name, not from the original **.mod** file name if that is different. Sensible users will avoid confusion by making sure that module and file name correspond. If necessary, it is possible to override this behaviour (see **–f** compiler option in Chapter 2). All output file names are transformed to lower case characters, and truncated if necessary to 80 characters in length. All searches for **.syx** and **.rfx** files use the same convention.

The syntax of the build command is

build [-options] *base*

base is the name of the base file, with no extension. The '**–v**' option produces a verbose listing on the screen. Other options are listed in the following section. The options for **build** follow the same conventions as the compiler options. That is, options may appear in any number of groups, in any order.

3.2 Builder Option Flags

–D	"**D**ebug" — the file is linked with the runtime stack unwinder
–q	"**q**uery" — the builder prompts for the names of reference files which it cannot find. Only the file base-name is required, and querying may be aborted by entering a blank line
–S	"a**S**sembler" — the initialization-call-chain code file and the linker script are persistent as files "`modbase.c`" and "`modbase`" respectively
–v	**v**erbose messages are displayed on the screen so that progress may be monitored, and the origin of any error messages determined
–V	builder **V**ersion date and time are displayed, and screen messages are verbose
–L*dirname*	the linker searches the directory *dirname* for library files, before searching the default library path

To build a profiling version of a program, **build** is invoked under the name **bldprf**. The option and filename conventions for bldprf are identical to those of **build**. Information on using the profiling facilities are given in the Chapter *Using the Profiling Tools*.

3.3 Running your program

Once all modules have been compiled successfully and the **build** utility used to create an executable file, the program is run by the command `base` where *base* is the name of the base file. A number of possible runtime errors can be detected when the program is executed. Refer to the Chapter *Errors and Error Messages* for more details.

Version specific details

The –**D** option is supported only on DECstation currently, and is the default for **gpm-pc**. The –**L***dirname* option is not supported under *MS-DOS*. The Apollo Domain version has a number of specific options which are detailed in the release notes for that version.

Chapter 4

Programming in the Large

4.1 Using gardens point modula to solve problems

The problem-solving process must start with a clear specification of what is required; from this a solution can be devised.

A typical structured (top-down) design will identify several major modules which make up the solution – these modules will be largely independent of each other (have few connections) but will be internally cohesive (share the same working data and detailed logic). Such modules are naturally expressed as Modula-2 modules. Thus the result of the high-level design is the partitioning of the problem and its solution into a number of modules, with the interfaces between those modules expressed by Modula *DEFINITION MODULE*s, and the main program written in terms of calls to the facilities defined in those definition modules.

Each of the modules then represents an independent subproblem, which is solved by writing the *IMPLEMENTATION MODULE* which provides the facilities advertised by its definition module – the definition module specifies what is to be done, and the implementation module does it, independent of any calling (or client) programs. Of course, the further refinement of the implementation may well identify lower-level facilities which are similarly defined and implemented as separate modules.

As experience with structured problem-solving is gained, it will commonly be the case that a module used in a previous solution will be useful in the current problem. Thus the design will not be strictly top-down – a measure of bottom-up design will be added, by recognising when previously-written modules from a library are appropriate. As well as reducing the design and coding effort, this approach produces more reliable code: if a previously-written module has been thoroughly tested (or, better still, proved correct) it can be re-used with confidence, rather than introduce new errors in a new solution. When modules are likely to be re-usable in this way, there is a greater incentive to design, code, test and verify them well.

Consider now a typical small problem solution using **gpm**. An application program such as a cross-reference generator must maintain a table of word descriptors, where each word descriptor comprises the word itself and a sequence of reference line numbers. This is clearly an application of a standard table abstraction, and the program can be written in terms of calls to standard table facilities such as "insert an entry", "look up an entry", "display all entries", etc. A definition of such a table abstraction constitutes a *DEFINITION MODULE*

Table; the implementation may be in terms of any appropriate data structure, but typically a balanced tree or hash table will be chosen for efficiency combined with flexibility.

Thus we now have a solution comprising three logical units: the definition of the table abstraction, the cross-reference program which uses it, and the implementation as, say, a balanced tree. In Modula terms, these are separately-compiled modules, each stored in a separate file:

```
file table.def:        DEFINITION MODULE Table;
                       (* ..... *)
                       TYPE Table;
                            ItemType = RECORD
                                         key : KeyType;
                                         ...
                                       END;
                            KeyType = Word;
                       PROCEDURE Insert (VAR t:Table; item:ItemType);
                       PROCEDURE Lookup (t:Table; key : KeyType;
                                         VAR found : BOOLEAN;
                                         VAR item:ItemType);
                       ...
                       END Table.

file crossref.mod:
                       MODULE CrossReference;
                       (* ..... *)
                       FROM InOut IMPORT Write, ... ;
                       FROM Table IMPORT Insert, ... ;
                       ...
                       BEGIN
                         ...
                         Insert (words, thisWord);
                         ...
                       END CrossReference.

file table.mod:        IMPLEMENTATION MODULE Table;
                       (* ..... *)
                       TYPE Table = POINTER TO TreeNode;
                            TreeNode = ...
                       ...
                       END Table.
```

Having produced these files, compilation and linkage using gpm proceeds as follows : [1]

1. *table.def* must be compiled first, since both other compilation units depend on its definitions (*crossref.mod* uses them, *table.mod* must supply a matching implementation):

[1] The sample outputs are for the gpm version of Jun 21 1989; there may be differences in detail in the output produced by later versions, especially in the –V (super Verbose) output used.

```
$gpm -IV table.def
HP-Precision Architecture Version of Wed Jun 21 19:55:14 1989
Opening "table.def" as input
Creating symbol file "table.syx"
$
```

and the result is the creation of the symbol file *table.syx*, which holds the compiler-readable equivalent of *table.def*.

2. Either *table.mod* or *crossref.mod* can now be compiled; each imports the table definitions (explicitly in the case of *crossref.mod*, implicitly in the case of *table.mod*), so **gpm** reads the data from *table.syx* to check the matching correctness of the cross-reference program use and the tree implementation:

```
$gpm -V crossref.mod
HP-Precision Architecture Version of Wed Jun 21 19:56:04 1989
Opening "crossref.mod" as input
... Importing <InOut> from /usr/local/m2sym/inout.syx
      -- mod <SYSTEM> key = 0
      -- mod <InOut> key = 1608020411
... Importing <Table> from table.syx
      -- mod <Table> key = 410000993
... Importing <Ascii> from /usr/local/m2sym/ascii.syx
      -- mod <SYSTEM> key = 0
      -- mod <Ascii> key = 3360978574
Output name is "crossreference"
..... Header file /usr/local/m2sym/m2rts.h
$
```

producing *crossreference.o* and *crossreference.rfx*. (Note the longer output filenames based on the module name.) The file *crossreference.o* is the machine code version of the cross-reference program, and *crossreference.rfx* is the reference file (nothing to do with the cross-reference of our sample application !) which notes that the code in *crossreference.o* cannot work until it is combined with the implementation code of each of the modules it imported (*InOut*, *Table* and *Ascii*).

```
$gpm -V table.mod
HP-Precision Architecture Version of Wed Jun 21 19:57:21 1989
Opening "table.mod" as input
... Importing <Storage> from /usr/local/m2sym/storage.syx
..... Header file /usr/local/m2sym/storage.h
..... using object library "<storage.o>"
      -- mod <SYSTEM> key = 0
      -- mod <Storage> key = 1497782521
... Importing <InOut> from /usr/local/m2sym/inout.syx
      -- mod <SYSTEM> key = 0
      -- mod <InOut> key = 1608020411
```

```
... Importing <Ascii> from /usr/local/m2sym/ascii.syx
      -- mod <SYSTEM> key = 0
      -- mod <Ascii> key = 3360978574
... Importing <Table> from table.syx
      -- mod <Table> key = 410000993
Output name is "table"
..... Header file /usr/local/m2sym/m2rts.h
$
```

producing *table.o* and *table.rfx* (in this simple case, *table.rfx* includes references to *Storage*, *InOut* and *Ascii*).

3. Now we must build an executable program from the cross-reference program's machine code, that of the table implementation, and that of the library module *InOut*. This phase is performed by the **build** utility:

```
$build -V crossreference
Build version of Sat Jun 24 09:42:56 1989
Reading <CrossReference> key = 0
  Importing <InOut> key =  1608020411
  Importing <Table> key = 410000993
Reading <InOut> key =  1608020411
  Importing <Files> key = 3143116421
Reading <Table> key = 410000993
  Importing library file <<storage.o>>
  Importing <InOut> key =  1608020411
Reading <Files> key = 3143116421
$
```

Given the 'base name' *crossreference*, the build program reads *crossreference.rfx* to find the modules directly imported by *crossreference*; in turn, their **.rfx** files will lead to any lower-level modules needed. Having found all such needed modules in their machine-code (**.o**) forms, **build** constructs a small C program which will invoke *crossref*, and linker commands which will link it to all the modules found via **.rfx**'s (in this case, *crossreference* itself, *table*, and *inout*). The result of this linkage is the executable file *crossreference*.

4.2 Consistency checks between modules

4.2.1 Symbol-file key values

Whenever a definition part is compiled by **gpm** the system evaluates a *key-value* for the symbol file. This key, sometimes also called a *magic number*, is unique to the symbol file. If the definition file is changed in a way which changes the symbol file, the key value will be different. The key-values of all the symbol files used in a particular compilation are also recorded in the reference file.

The system uses these key values to ensure that programs only use consistent versions of modules. Some checks are carried out during compilation, while others are carried out during load-building.

4.2.2 Compile-time key-value checks

gpm records the key values of every symbol file which it imports. The symbol files also contain the key values of any symbol files which the definition-part file itself imported. Thus **gpm** is able to check when it meets the same symbol file directly and indirectly that it has a consistent version.

If **gpm** finds symbol files with inconsistent key-values it issues an error 300 message. In difficult cases a complete trace of the directly and indirectly imported key-values may be obtained by use of the super-verbose –V option.

4.2.3 Build-time key-value checks

The **build** program checks the information included in .rfx files to verify that all modules importing from a common definition module in fact used the same version of that definition file. This guarantees that the separate compilations of program and implementation modules are nevertheless dependent on the same definition information, and so will work together correctly. If this were not the case, the cross-reference program might have been compiled using information in *table.syx* (derived from compilation of *table.def*), then *table.def* could have been changed and recompiled, and a changed *table.mod* recompiled (matching the new definitions in *table.syx*); clearly, *crossreference.o* and *table.o* would not work together. If **build** detects any such error in the dependency checks, it outputs a diagnostic and aborts the build process:

```
$build -V crossreference
Build version of Sat Jun 24 09:42:56 1989
Reading <CrossReference> key = 0
  Importing <InOut> key = 1608020411
  Importing <Table> key = 410000993
Reading <InOut> key = 1608020411
  Importing <Files> key = 3143116421
Reading <Table> key = 11511020
** Inconsistent key for module <Table> **
  Importing library file <<storage.o>>
  Importing <InOut> key = 1608020411
Reading <Files> key = 3143116421
*** File creation unsuccessful ***
$
```

You must then recall what changes have been made to determine what recompilations are needed, and in what order they should be performed. Clearly, in general, new or changed **.def** modules should be compiled first, since they are the common basis for later compilations; within the **.def**'s, lower-level facilities imported by other **.def**'s must be compiled first. Then, application programs such as *crossreference* and the various implementations may be compiled in any order (as long as all the **.def**'s they import have been compiled). Finally, **build** produces a new working version. When very complex dependencies exist between modules it is useful to allow the utility **gpscript** to analyse the dependencies and compute the compilation order, or to use **gpmake** to perform an optimized partial recompilation.

4.3 File names – and gpm

Commonly, file names are chosen to match module names – thus a service module such as *Table* resides in the pair of files *table.def* and *table.mod*, while an application such as *CrossReference* might reside in *crossreference.mod*.

However, **gpm** will normally use the module name as the basis of the symbol, object and reference files its creates, regardless of the source file name (the source file name is, of course, relevant – it must be the argument to the **gpm** command). Further, in forming output file names from the module name, **gpm** will convert all alphabetic characters to lower case and truncate the name to 80 characters, should that be necessary. Thus, the module name *CrossReference* led to files *crossreference.rfx* and *crossreference.o*. Subsequent searches when other modules reference *.syx* files, or when the build utility uses *.rfx* files, use the same lower case convention.[2]

As a side-effect of the module to file name manipulations, common Modula identifier conventions such as capitalising the first character of module, type and procedure names, and the first character of any concatenated word (to make multi-word identifiers readable without resorting to underscores), will not carry through to file names. This is consistent with the general Unix practice of lower-case file names (except for files which should be specially noticeable, such as **README.NOW**).

Some Unix sites, however, do use mixed-case file names, and in this environment **gpm** can be directed to produce corresponding mixed-case names by setting the environment variable *GPNAMES* to `"Mixed"`. In this case, module *CrossReference* would lead to files *CrossReference.rfx* and *CrossReference.o*, and a subsequent **build** would produce an executable *CrossReference*. For further details, see Section 7.2 Environment Variables, and the library module *GpFiles*.

As another way of controlling file name generation, the compiler option **-f** will use the file name as the basis for files it produces, ignoring the module name, and will not map to lower case. Note that this option cannot be used with program modules. Thus, had we chosen the longer source file name *mytable.mod*, `gpm -f mytable.mod` would produce *mytable.o* and *mytable.rfx*, while a source file called *MyTable.mod* would produce *MyTable.o* and *MyTable.rfx*.

Another use for the **-f** option is a case where there are two implementations of some facility. Consider again the *Table* example. Clearly, there should be only one definition module – corresponding to the one concept of what a *Table* abstraction does. But if we wish to have available both a balanced tree implementation and a hash table implementation, we could choose to have two implementation files:

```
file baltreetable.mod: IMPLEMENTATION MODULE Table;
```

[2]Early versions of **gpm** truncated filenames to 8 characters plus extension. For compatability with these versions, both the compiler and the builder do a final check for the shorter version of the name before declaring a file missing.

```
                        (* Implement Table using a balanced tree *)
                        ...
                        END Table.

 file hashtable.mod:    IMPLEMENTATION MODULE Table;
                        (* Implement Table using a hash table *)
                        ...
                        END Table.
```

Now the command, **gpm -f baltreetable.mod** will produce files *baltreetable.o* and *baltreetable.rfx*, while the command, **gpm -f hashtable.mod** will produce *hashtable.o* and *hashtable.rfx*.

In both cases, the entry point names will still be based on the module name, which matched the single definition – *InsertTable*, etc. In order for the build phase to succeed, one of the two implementations must be chosen. This can be done in two ways:

(1) copy the appropriate files to *table* files:

```
$cp baltreetable.o table.o
$cp baltreetable.rfx table.rfx
$build crossreference
$
```

(2) use the **-q** option of **build** to prompt for the missing *table* files :

```
$build -q crossreference
Build version of Sat Jun 24 09:42:56 1989
Reading <CrossReference> key = 0
  Importing <InOut> key = 1608020411
  Importing <Table> key = 410000993
Reading <InOut> key = 1608020411
  Importing <Files> key = 3143116421
<Table>** Ref file not found ...
Filename for <Table> : baltreetable   (* users response *)
Reading <Table> key = 410000993
  Importing library file <<storage.o>>
  Importing <InOut> key = 1608020411
Reading <Files> key = 3143116421
$
```

Of course, an alternative to this whole process would be to compile the implementations without the **-f** option. In that case, each would produce *table.o* and *table.rfx* directly, overwriting any previous version. To switch between implementations you could either rename the previous version before recompiling, or let it be overwritten and recompile again if necessary to return to the alternative:

```
$gpm baltreetable.mod           (* => table.o and table.rfx *)
$build crossreference           (* with balanced tree table *)
```

```
$

$mv table.o baltreetable.o        (* if want to save tree version *)
$mv table.rfx baltreetable.rfx
$

$gpm hashtree.mod                 (* => table.o and table.rfx *)
$build crossreference             (* with hash table *)
$
```

4.4 File names – and the build phase

The build utility invokes the standard Unix linking loader **ld**. Modern Unix systems do not have any sensible limit to the length of external symbols, and **gpm** produces symbols which may extend up to 31 characters in length.[3] These external/entry-point names correspond to Modula procedure names imported by client programs (external references) and exported by definition/implementation modules (entry points). Since the same procedure name might be exported by two modules, the external/entry names generated by **gpm** must include the module name.

A compromise must be made between the unlimited module and procedure names of Modula and this 31 character limit. **gpm** produces external names by taking the module name (shortened to the first ten characters if necessary) followed by the symbol name (shortened to 20 characters if necessary). These two parts are separated by an underscore character. Thus procedure *Insert* of module *Table* gives rise to an entry point name `Table_Insert`, while *WriteString* of *InOut* becomes `InOut_WriteString`, and *Compare* from *StdStrings* becomes `StdStrings_Compare`.

Fortunately, most of this is invisible to you; it will only become apparent when you are using the debugger (**xdb** or **dbx**) to investigate a run-time error. Then, the names in the stack backtrace (the chain of procedures leading to the one in which the error was detected) will all be of this form.

4.5 Maintaining complex programs

Large programs will comprise many modules, with possibly complex export/import dependencies. Keeping track of dependencies may thus be quite difficult (though, the better the design, the simpler will be the interactions). Some cases will be simple – (1) a change to an implementation module requires only that that module be recompiled and the whole program be re-built; (2) if a definition module has few dependencies on it, only the definition module, its implementation, and the few dependencies need be recompiled, and the program re-built.

Beyond these simple cases, two courses are open:

(1) Create shell scripts which invoke **gpm** and **build** to compile all modules in the correct order and build the executable file. This may be split up into a **makedefs**, followed by a **makemods**, followed by **build**. This may recompile more modules than is required, but is simple and fairly fast.

[3]Early versions of **gpm** formed character long linker names which were only 14 characters long

(2) Use the supplied utility **gpmake**. This program automatically analyses the source code of all the modules to extract the dependencies. **gpmake** also calculates the order of compilation for definition modules, where these import from each other. It compiles all inconsistent non-library modules in the correct order. See the Chapter *Using The gpmake tool* in the technical reference manual for more details.

4.6 Other utilities

As well as **gpmake**, which has already been mentioned, there are several other utilities which are helpful in the maintainence of Modula programs.

4.6.1 The cross reference generator gpxrf

The utility **gpxrf** produces crossreference listings for modula programs. The default operation of the utility produces a listing of user defined identifiers in case-sensitive lexicographic order.

The program accepts the following options —

–p includes pervasive identifiers in the listing

–f lists identifiers in ascending order of frequency of use

A full description of the mode of use is given in the *Technical Reference Manual.*

4.6.2 The definition extractor grepdef

The utility **grepdef** extracts definitions from definition files on the symbolic path $M2SYM. It is useful for discovering which modules define particular identifiers, and the exact spelling of these. The program is invoked by the command

`grepdef` *RegExp*

where *RegExp* is a regular expression in the style expected by *grep.* All lines from files with names ending in `.def` which match the regular expression are printed. The order of directory search exactly matches that used by the compiler. The current directory is searched first, and the directories on the path $M2SYM are then searched in order.

4.7 Temporary files

Both **gpm** and **build** use various temporary files, which are normally removed on successful completion. In abnormal situations, including a user interrupt, and when appropriate options are selected, these files will be apparent.

gpm compiles your Modula program or implementation code into a C file *modulename.c*, plus the reference file *modulename.rfx.* The C code is then compiled by **cc** to give *modulename.o*, and the C source *modulename.c* is removed. The compiler option **-S** suppresses the C compilation phase and leaves the file *modulename.c* in the current directory. This option is largely for debugging purposes; the C code is not particularly interesting, and is hard to read due to the lack of comments and liberal use of compiler-constructed internal names. Nevertheless, it has recognisable similarity to the Modula source, and those versed in C may use it to localise problems.

build creates temporary files with a names like *bld28077.c* and *bld28077*. The first is the initialization call-chain code, while the second is the linker script. Again, an abnormal event may leave such files in the directory **/tmp**.

A run-time error will cause a memory image to be dumped to the file *core* which is used by the postmortem debugger. Since this may be a large file, it is good practice to remove it when the fault has been corrected.

General housekeeping such as this may be simplified by a *makefile* entry *clean* which includes actions such as `rm core`; a cleanup is then invoked by `make clean`.

Chapter 5

Compiler Diagnostics : Summary

Here are the error messages exactly as they appear on the screen and in listing files. Detailed description of the circumstances under which each of these may arise is given in the chapter *Interpreting compiler diagnostics* in the Technical Reference Manual.

Lexical Errors

```
  1 Line ends inside literal string
  2 Illegal character in input file
  3 Input file ends inside a comment
  4 Invalid exponent in REAL constant
  5 Illegal character in numeric constant
  6 Floating-point error during constant evaluation
  7 Number too long
  8 Character constant too large (377B is maximum)
  9 Illegal use of underscore in identifier
```

Syntax Errors

```
100 Invalid symbols precede start of module
101 No identifier at end of module
102 No fullstop at end of module
103 Expected END symbol
104 Expected module END symbol
105 Expected semicolon
106 Expected declarations
107 Expected equals sign
108 Expected identifier
109 Expected IMPORT symbol
110 Expected comma
111 Expected ')' symbol
112 Expected '..' symbol
113 Error in qualified identifier
```

```
114 Expected parameters
115 Expected ']' symbol
116 Expected OF symbol
117 Expected colon
118 Formal parameter bad
119 Expected '{' symbol
120 Error in expression
121 Expected '(' symbol
122 Expected '}' symbol
123 Expected '|' symbol
124 Expected EXPORT symbol
125 Expected selectors
126 Expected addops
127 Expected mulops
128 Error in statement
129 Expected DO symbol
130 Expected UNTIL symbol
131 Expected ':=' symbol
132 Expected TO symbol
133 Expected THEN symbol
134 Expected start of type
135 Expected start of factor
136 Expected BEGIN
137 Premature exit: too few ENDs in  block
138 Expected END identifier;
139 Resynchronizing here
140 Foreign import must be "IMPORT IMPLEMENTATION FROM litstring"
```

Semantic Errors

```
200 Identifier at block end does not match
201 Symbol file missing
202 Identifier is not exported from module
203 Identifer already known in this scope
204 Identifier not known in this scope
205 Qualified identifier is not a type name
206 Type is not an ordinal type
207 Expression is not compatible with declared type
208 Identifier is not a constant
209 Maximum of range is less than minimum
210 Implementation limit exceeded for set base type
211 Target of forward reference not declared
212 Type ident not expected here
213 Function HIGH cannot be used in a constant expression
214 Parameter is of wrong type
```

215 Range of type exceeded
216 Too many parameters
217 Conversion not implemented
218 Not of numeric type
219 Operation invalid on constant
220 Type incompatible operands
221 Not of Boolean type
222 Record field name is not unique
223 Opaque type only allowed in definition part
224 Opaque type not elaborated
225 Exported procedure not declared
226 (Implementation restriction) Too many formals of same type
227 Invalid elaboration of opaque type (must be a pointer)
228 Invalid elaboration of procedure header
229 Function return type not as defined
230 Exported object not declared
231 Too many constants in enumeration
232 Designator is not a record type
233 Fieldname not known for this type
234 Attempted field selection not on a record structure
235 Designator is not a variable
236 Attempted pointer dereference not on a pointer type
237 Attempted array index not on an array type
238 BY expression not within INTEGER value range
239 Control variable not found in local scope
240 Control variable must not be a formal parameter
241 Control variable must not be imported or exported
242 Selectors not permitted on constant
243 Selectors not permitted on procedure name
244 Standard procs are not valid as proc-values
245 Function name not known in this scope
246 Designator is not a function
247 Designator is not a set type name
248 Too few parameters
249 Designator is not a procedure name
250 Designator is not a procedure variable name
251 Missing function return expression
252 Proper procedure cannot return a value
253 Actual value parameter not assignment compatible with formal
254 Actual variable parameter type not identical to formal
255 Actual variable parameter must be a variable
256 Actual parameter corresponding to open array formal not an array
257 Incompatible open array element type
258 Expression not assignment-compatible with variable
259 Return value not assignment-compatible with function type
260 Designator is not a function variable name

```
261 Selectors not permitted on set type name
262 HIGH may only be applied to open array parameters
263 Expression is not of type CHAR
264 Name of qualifying module clashes in outer scope
265 Enumeration constant name clashes in this scope
266 Name clashes with an enumeration constant name
267 Duplicate case selector in this range
268 Operand not of signed numeric type
269 Operand(s) not of Boolean type
270 Operand(s) not of numeric type
271 Operand(s) not of whole number type
272 Operand may not be compared
273 Proper inclusion operator not defined for sets
274 This type may only be compared for (in)equality
275 Right operand or first parameter not of set type
276 Exported enumeration constant clashes in outer scope
277 Procedure in !LIBRARY module calls non-library procedure
278 EXIT not within a LOOP
279 FOR loop control variable may not be modified
280 Name is not a module name
281 Expected proper procedure, not function
282 ALLOCATE not known in this scope
283 DEALLOCATE not known in this scope
284 Not a valid substitution for NEW or DISPOSE
285 Type ranges do not overlap at all
286 Selectors not permitted on type identifier
287 Nested procedures are not valid as proc-values
288 Implementation restriction: case range too large
289 Duplicate identifier in export list of module
290 Actuals passed to amorphous formals must be simple
291 No literals except sets and strings allowed here
292 Values cast to structured types must be simple
293 Value is too large to cast to unstructured type
294 Actual parameter must be a pointer type
295 Right operand must be greater than zero
296 FOR loop control variable is threatened in uplevel access
297 FOR loop control variable is threatened
298 Feature not implemented -- read latest release notes
299 Multi-dimensional open array parameters not implemented yet
300 Incompatible keys for symbol files
301 Wrong name in symbol file
302 Linker name is not unique
303 Fatal circular import through this module
304 Target object has zero storage size
305 Header file for !FOREIGN symbol file not found
306 Library name has bad format in header file
```

```
307 Expression cannot be aligned with specified type
308 Ident was already uplevel referenced in this scope
309 Procedure declared FORWARD was not elaborated
310 Array exceeds machine size limit
311 Parameter name is repeated
312 Expression must be a designator
313 Constructor has too few elements
314 Constructor has too many elements
315 Ranges not allowed in record or array constructors
316 Replicators only allowed for array constructors
317 Repetition count must be positive
318 Illegal assignment of INTERFACE proc with open array, see manual
319 Open array parameter may only be accessed element by element
320 Maximum nesting depth for procedure declarations has been exceeded
321 RETRY is not inside an EXCEPT clause
322 Forward IMPORT not elaborated
323 Declaration must precede use in a declaration
324 Expression must be compatible with control variable
```

Warnings

```
495 Name or function will change next release
496 Array is very large
497 Last type has zero storage size
498 Case statement has very low density
499 Variant tags are ignored in this implementation
500 Symbols follow module end
501 Obsolete syntax, colon is compulsory
502 Obsolete syntax, export list is ignored
503 Invalid option selection character (I,R,F,C only)
504 Too many levels of option restoration
505 Invalid option operator (+, -, = are valid)
506 Obsolete syntax, use SYSTEM.CAST for type transfers
507 Procedure is not called, assigned, or exported
508 No EXIT from this LOOP
509 Priority not implemented, ignored
```

Chapter 6

Syntax Diagrams for Modula-2

Introduction to the syntax diagrams

The syntax of Modula[1] is divided, for convenience, into two parts. First there are the **lexical** rules which describe how the symbols of the language are built up from characters of the implementation character set. Then there are the **syntactic** rules, which describe the ways in which the symbols of the language may be placed together to form grammatically correct programs. Of course there are also rules which have to do with type checking and so on, but these are not usually thought of as being part of the syntax of the language, and are not treated in this chapter.

The symbols which make up the alphabet of Modula consist of a number of keywords such as **FROM** and **AND**, some special symbols such as ':=' and '<>', and a small number of *lexical categories*. Lexical categories are symbols which have some substructure. They are the identifiers, numbers of various kinds, and literal strings. All of the other symbols have a fixed representation.

Modula is a free format language. The formatting of symbols onto separate lines and the placement of space between symbols has no significance to the compiler at all. All space characters, linebreaks, control characters and comments are treated uniformly as *whitespace*.

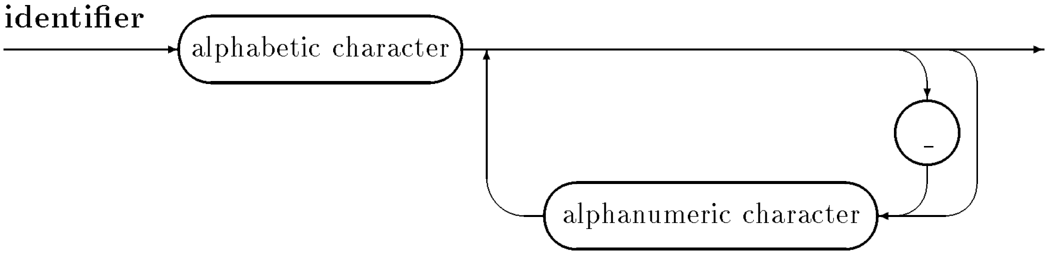
As a general rule, symbols must be separated by *whitespace* only when that is necessary to avoid ambiguity. This is the case between identifiers and keywords, between keywords and numbers, and between identifiers and numbers. In all other cases whitespace is optional, and should be used freely to assist human beings to read the program code.

Comments begin with a marker consisting of the character pair "(*" and terminate with a matching marker consisting of the pair "*)". Comments may be nested to any depth, and do not end until every opening comment marker has been matched with a closing marker.

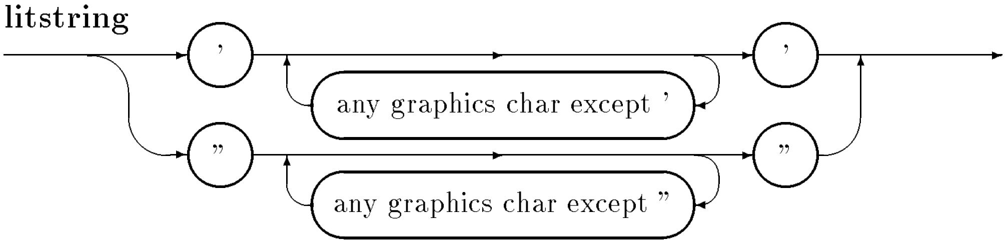
[1] This edition of the **gpm** manual includes some of the new syntax proposed by the ISO. The use of *FORWARD* as a keyword for compatability with compilers with single-pass restrictions is shown, but multi-dimensional open arrays, and absolute address expressions are omitted. It is believed that these diagrams accurately correspond to the April 1990 release of **gpm**.

Lexical Categories

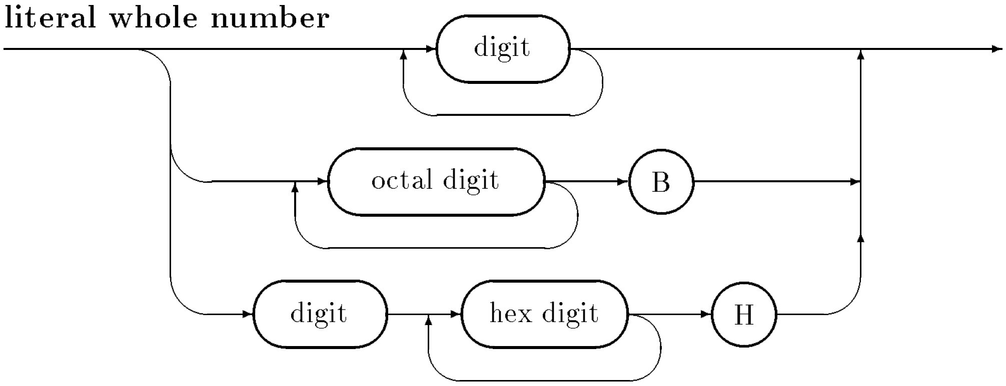
identifier



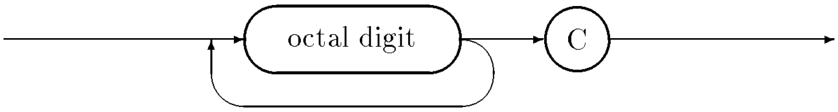
litstring



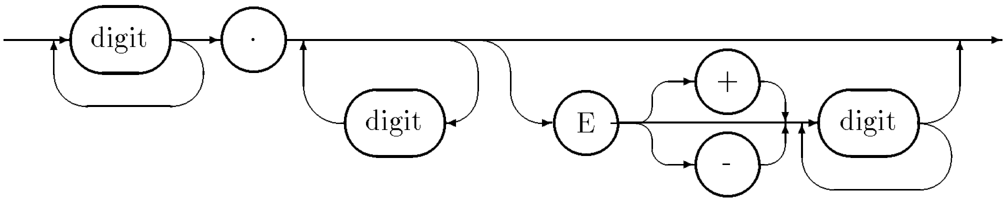
literal whole number



CHAR-**valued number**



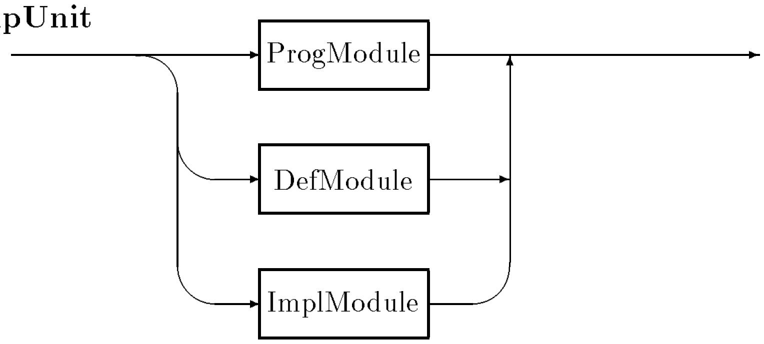
REAL-**valued number**



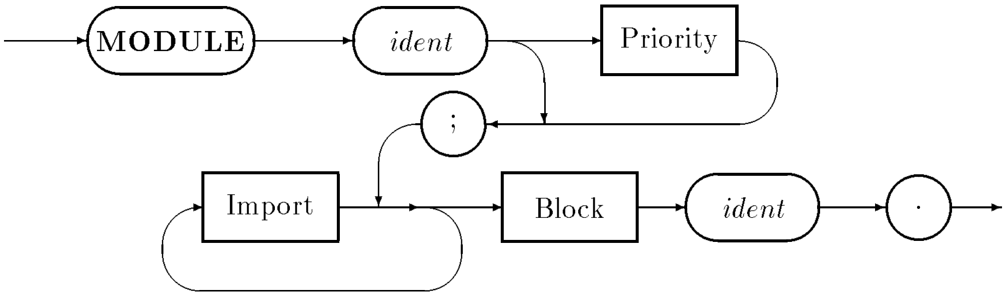
Syntactic Categories

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

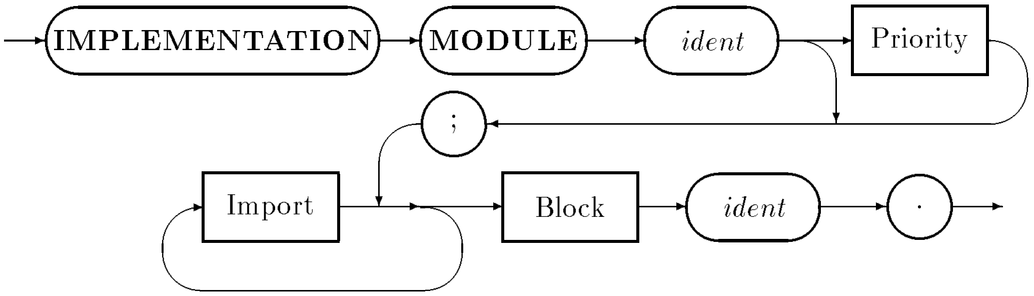
CompUnit



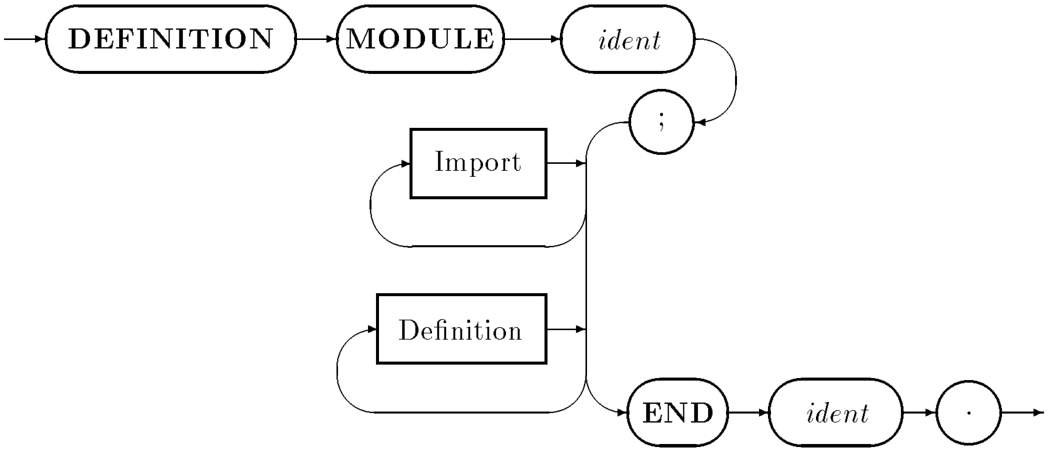
ProgModule



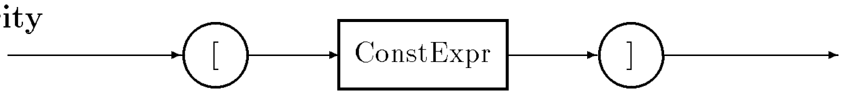
ImplModule



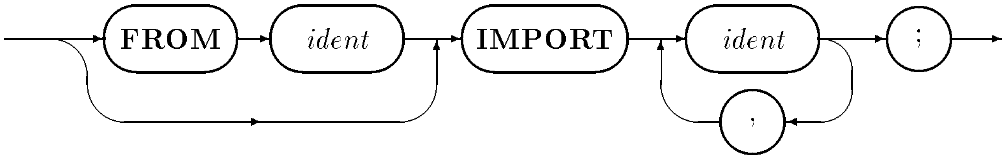
DefModule



priority

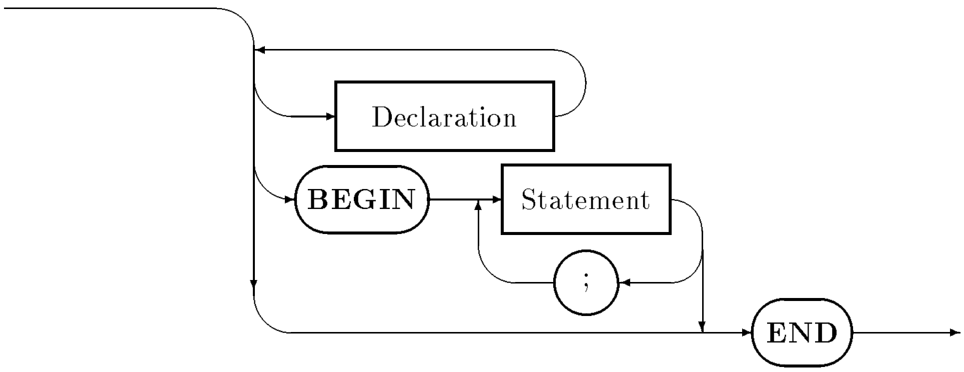


Import

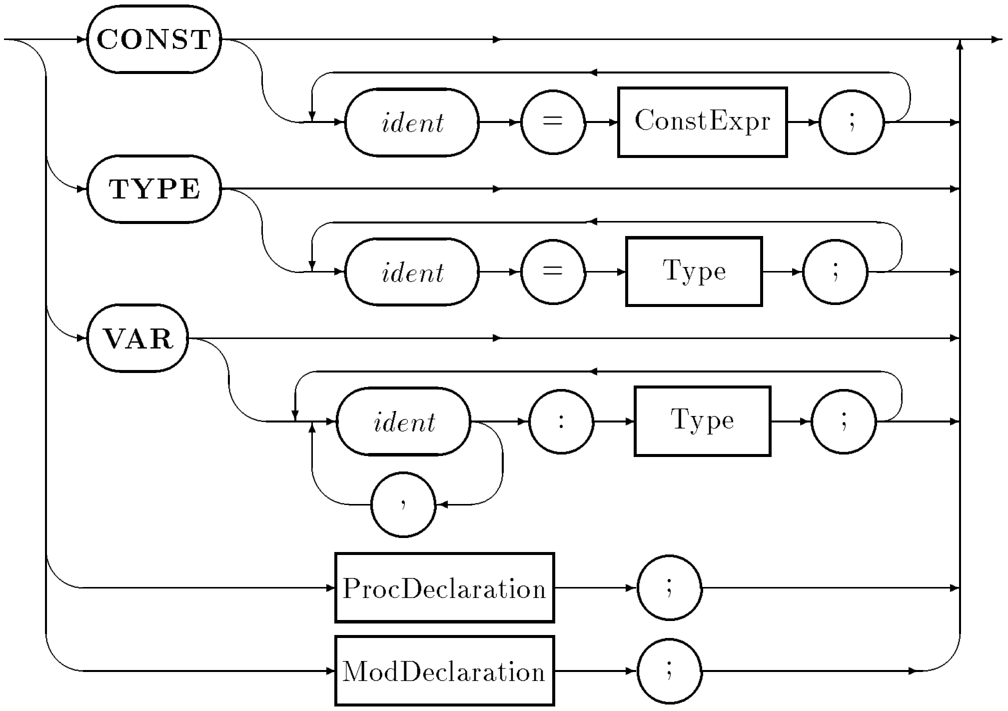


Export

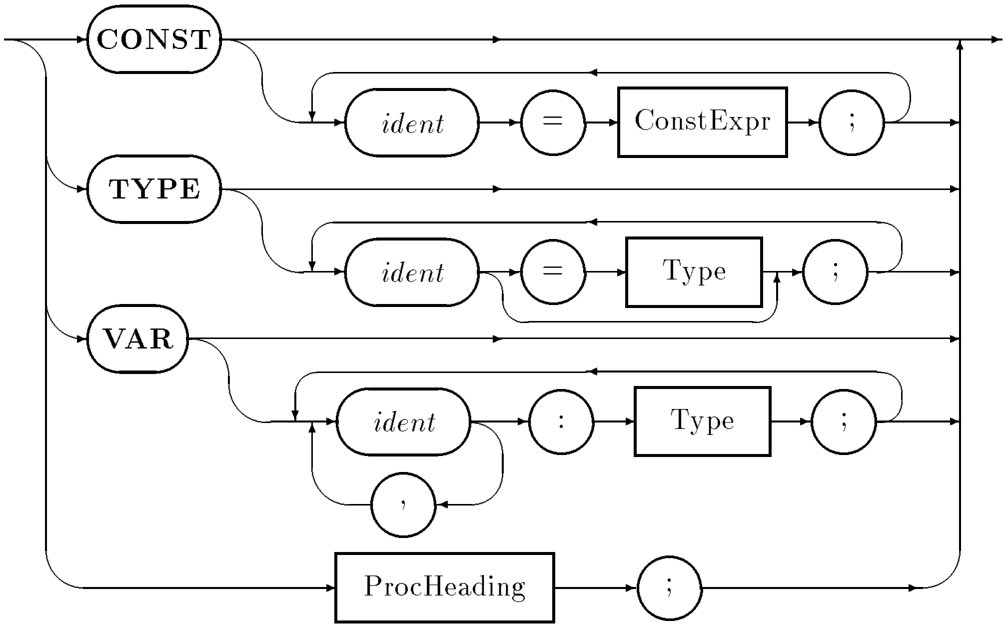
Block



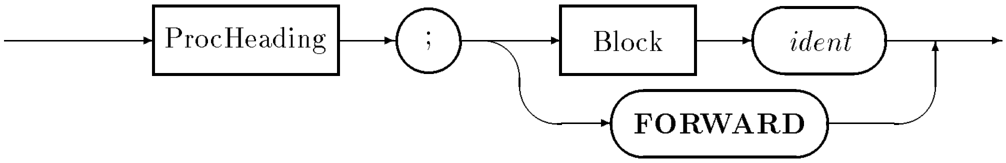
Declaration



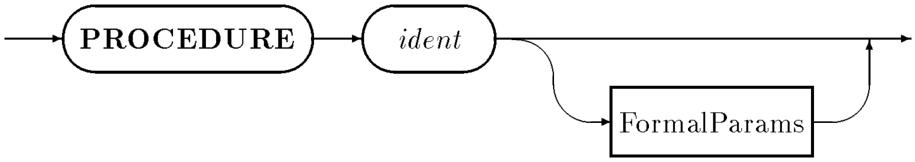
Definition



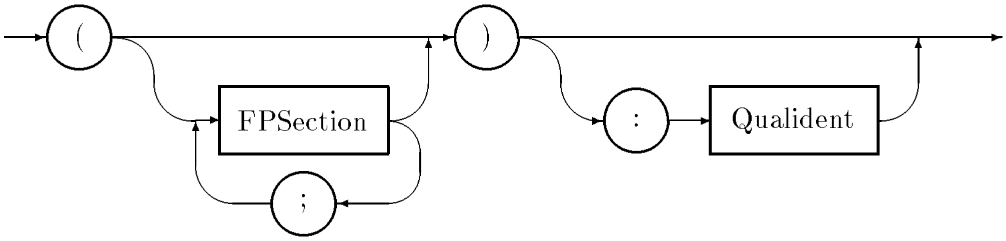
ProcDeclaration



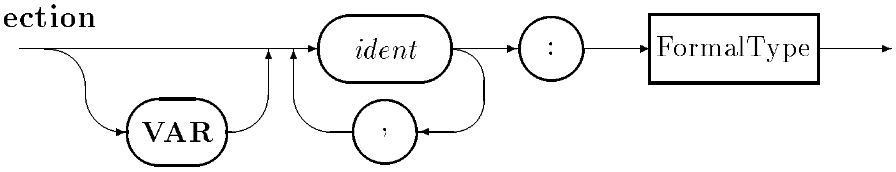
ProcHeading



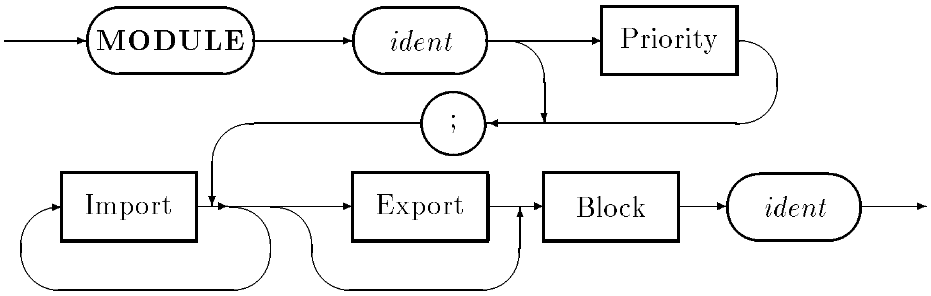
FormalParams



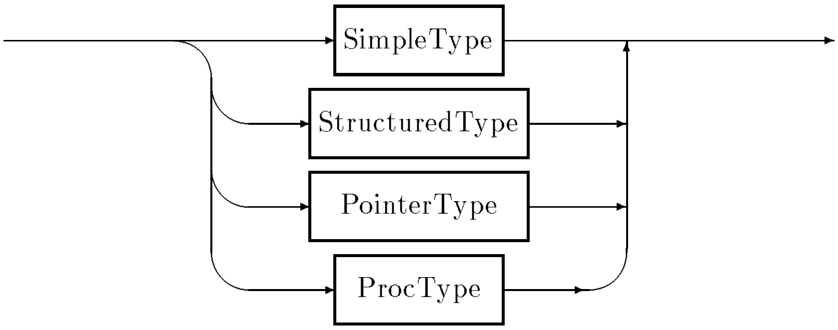
FPSection



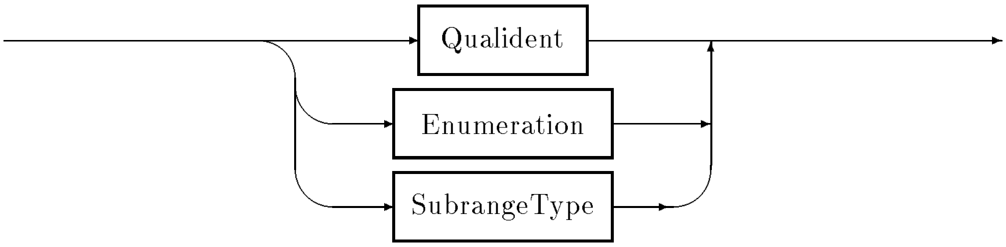
ModDeclaration



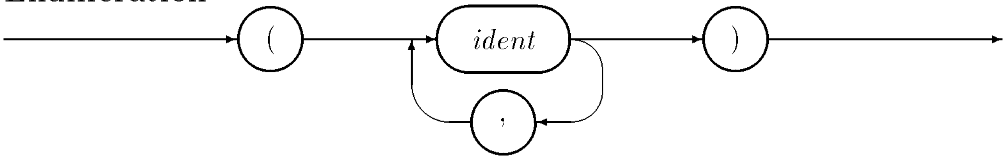
Type



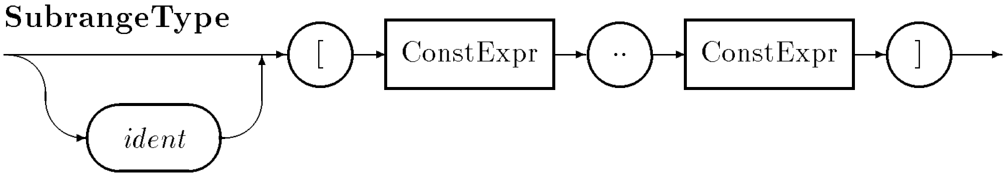
SimpleType



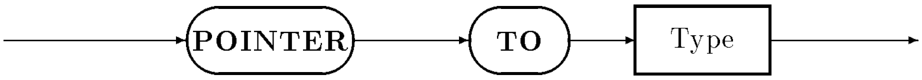
Enumeration



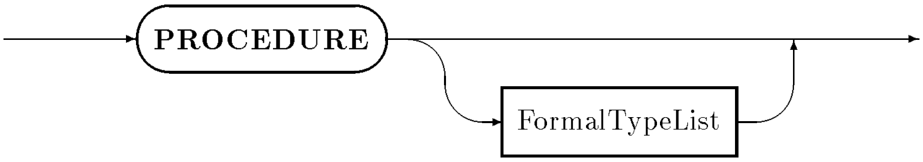
SubrangeType



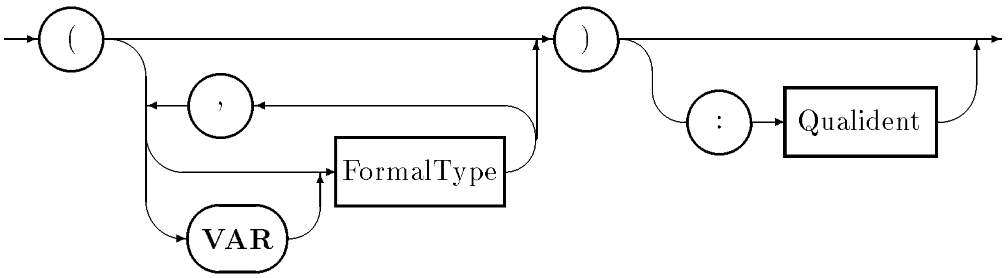
PointerType



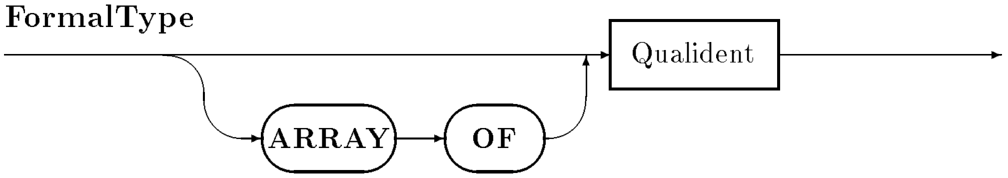
ProcType



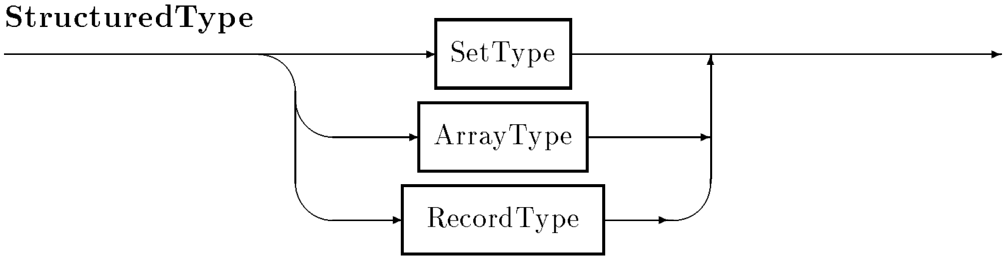
FormalTypeList



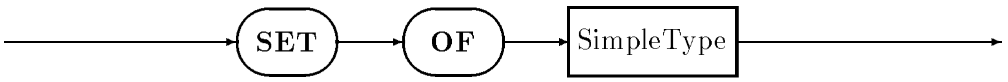
FormalType



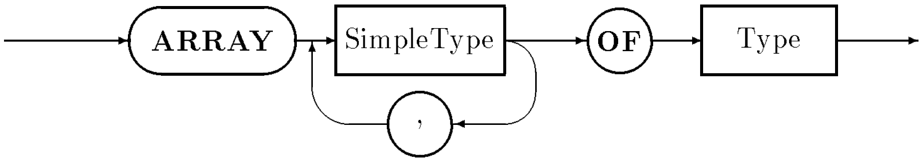
StructuredType



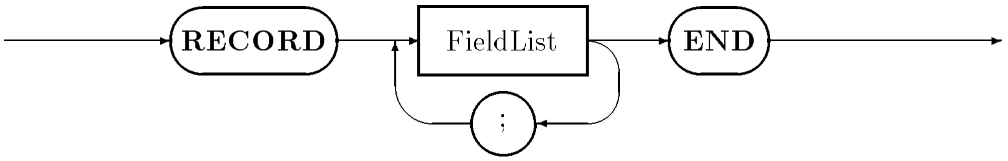
SetType



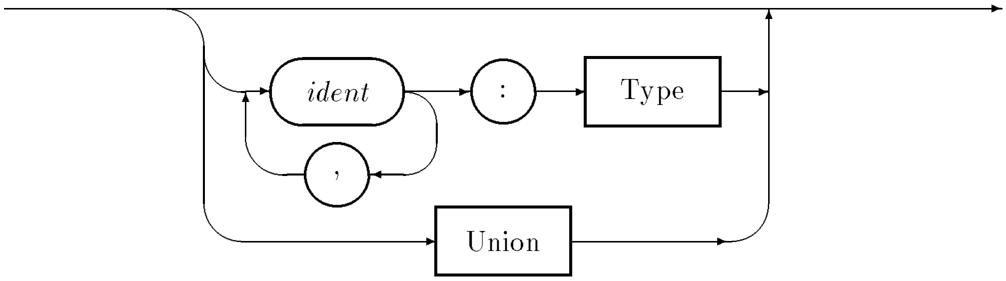
ArrayType



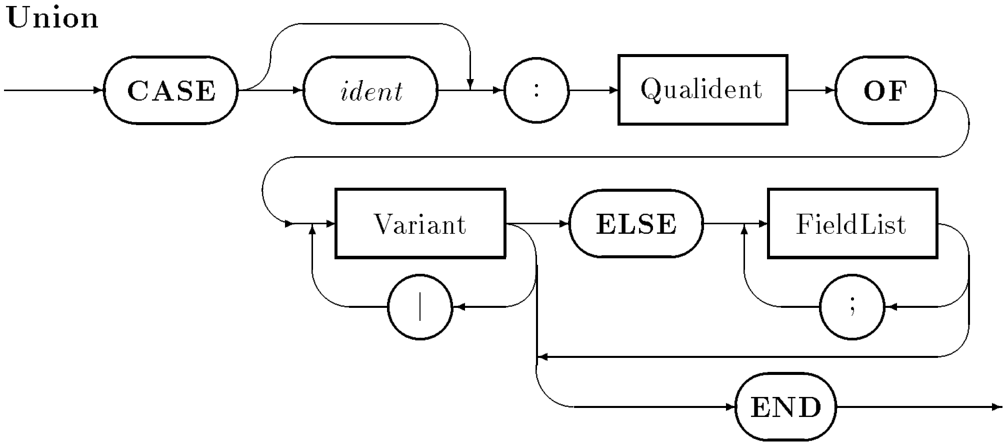
RecordType



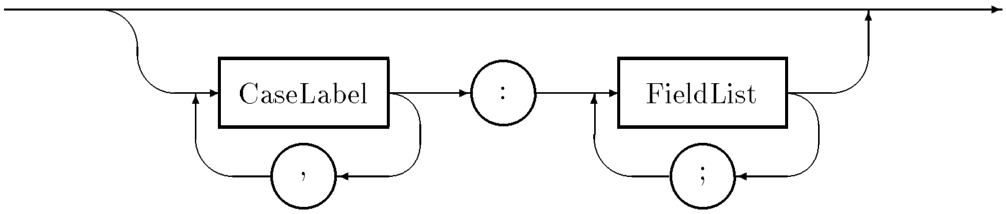
FieldList



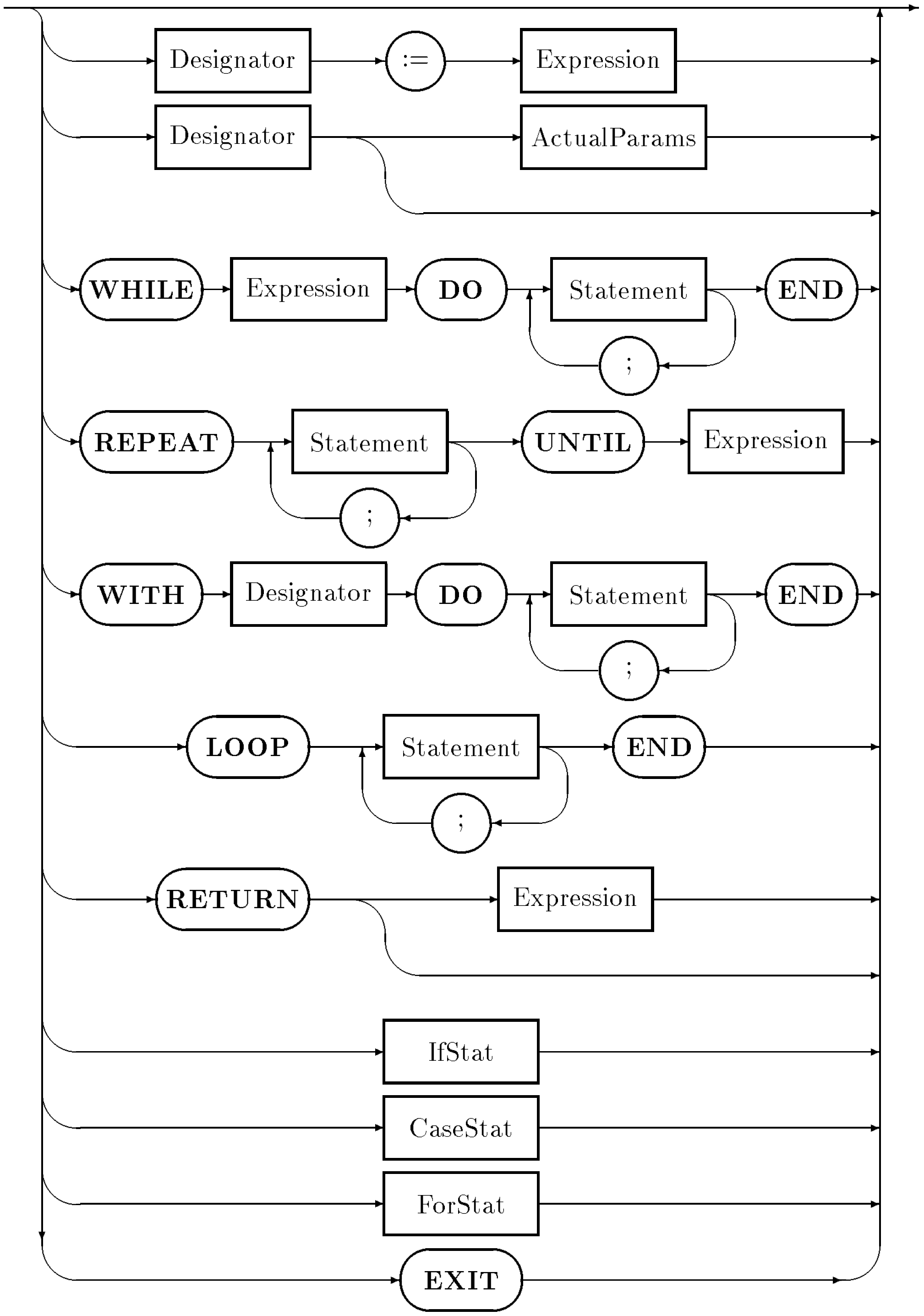
Union



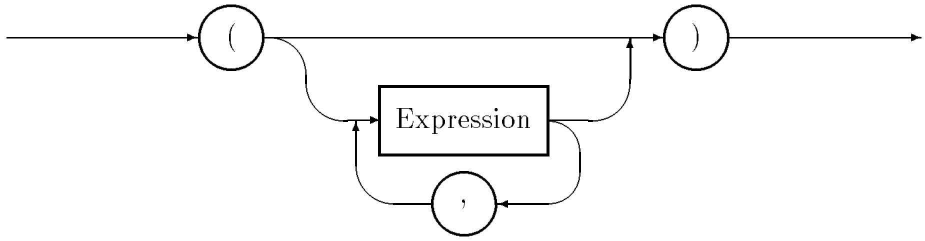
Variant



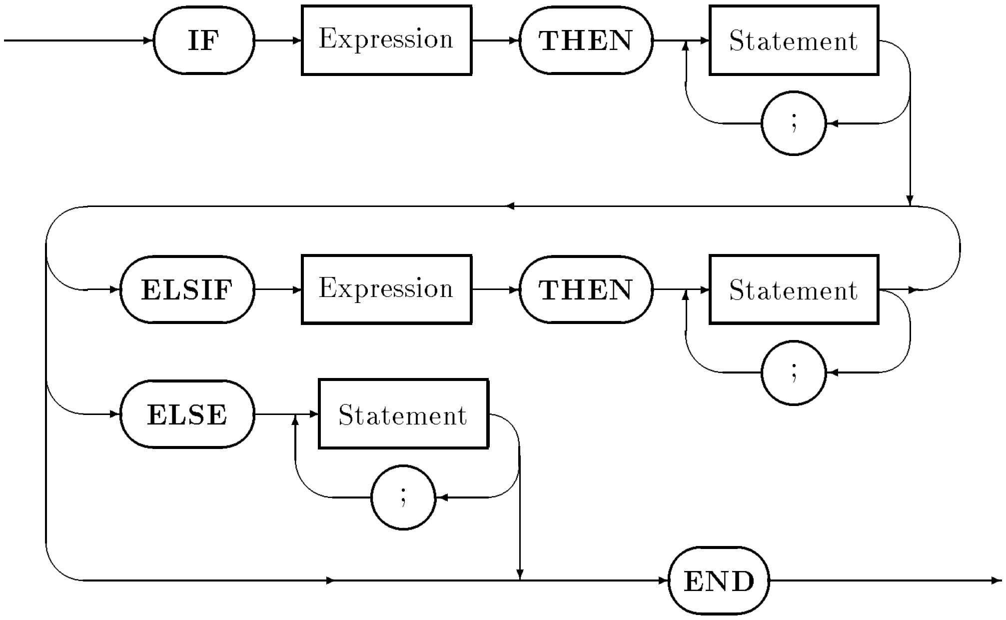
Statement


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

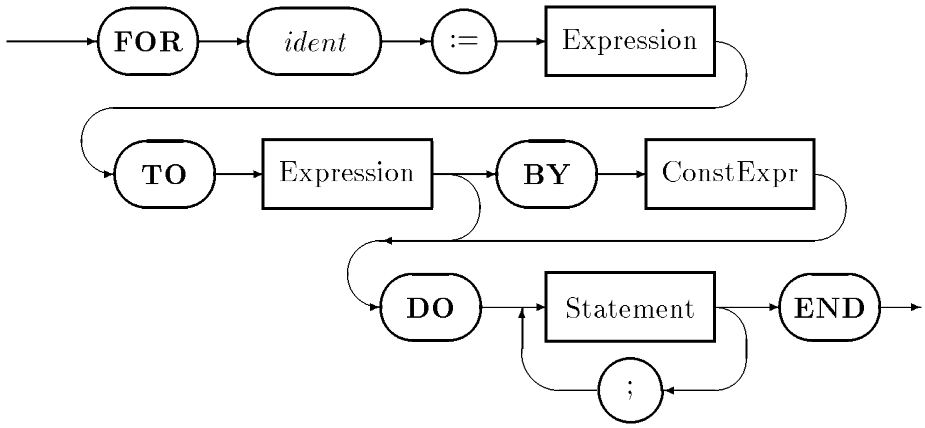
ActualParams



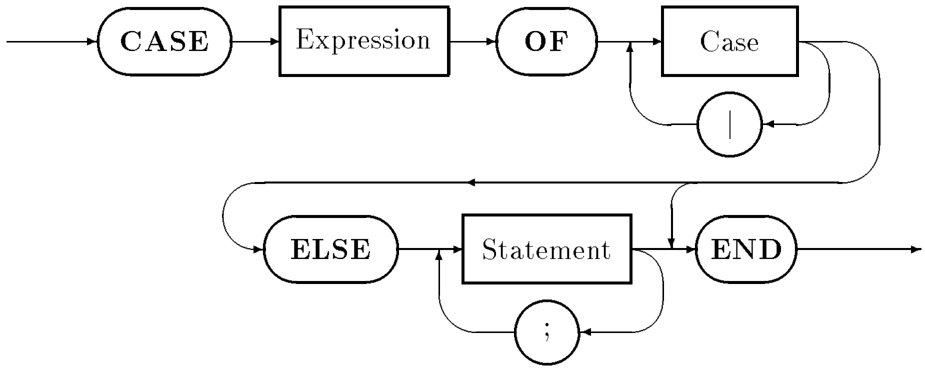
IfStat



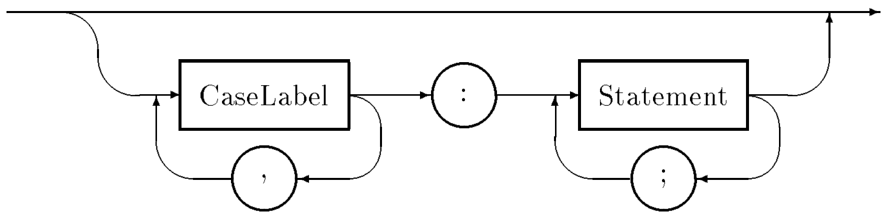
ForStat



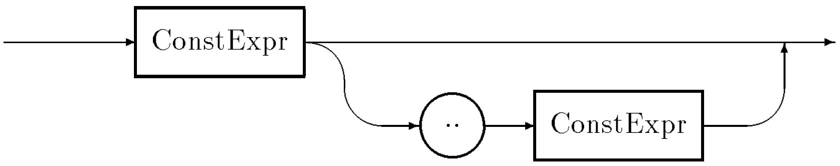
CaseStat



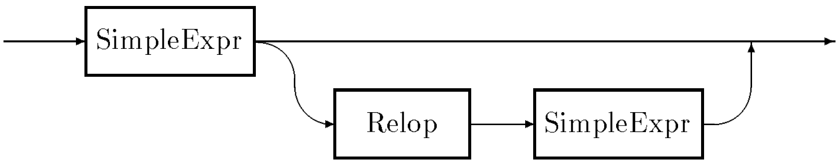
Case



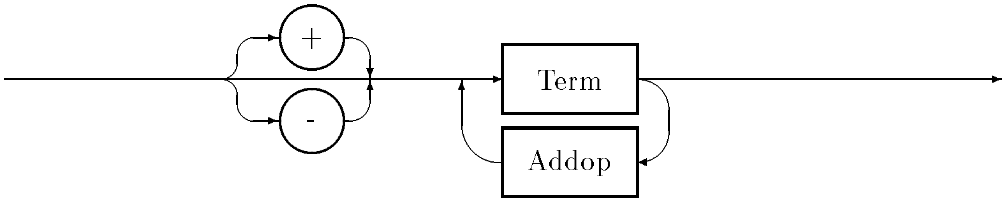
CaseLabel



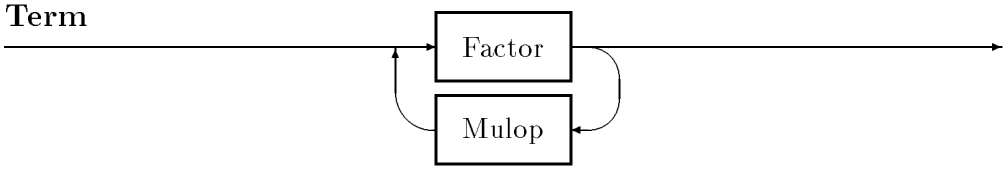
Expression



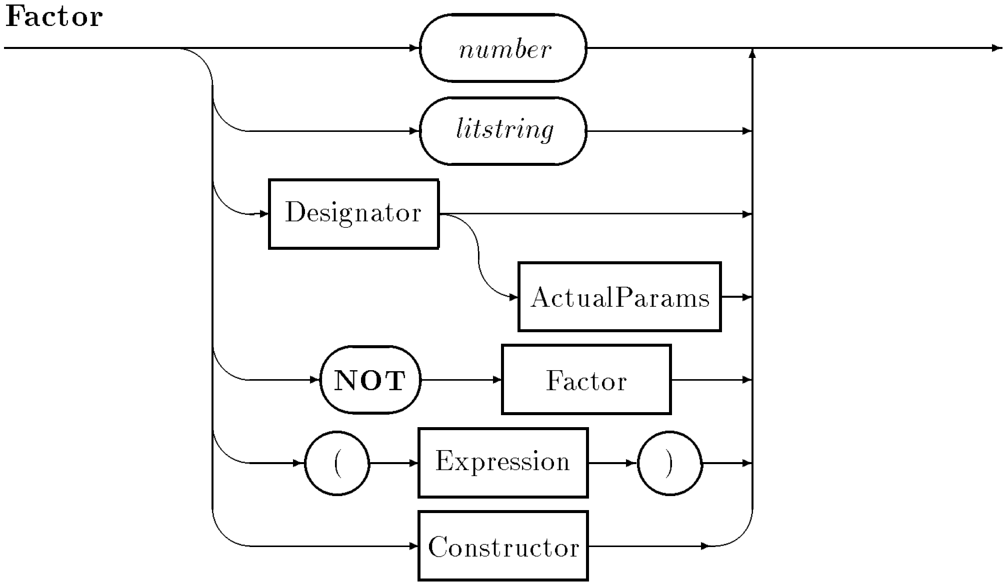
SimpleExpr



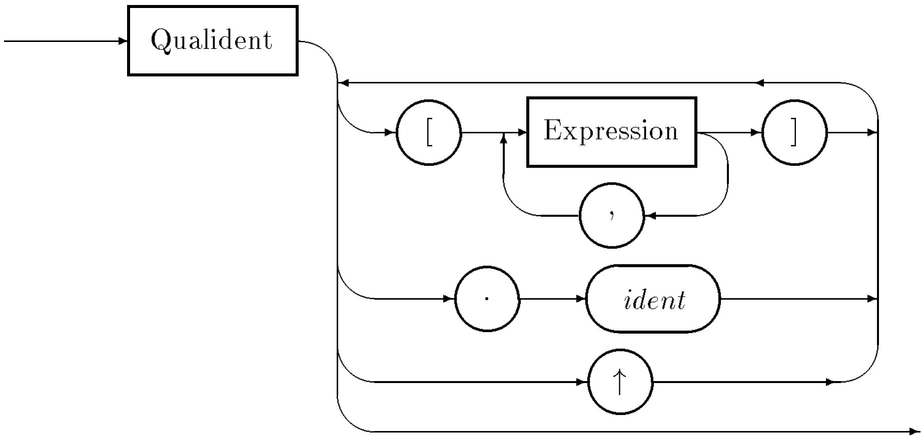
Term



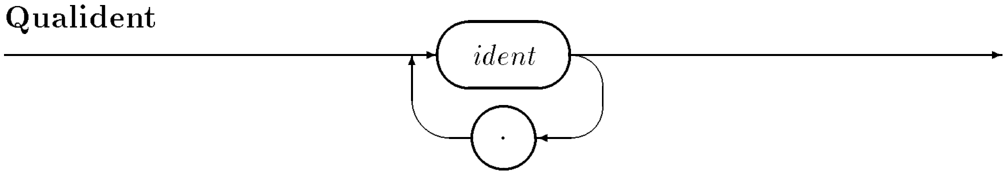
Factor



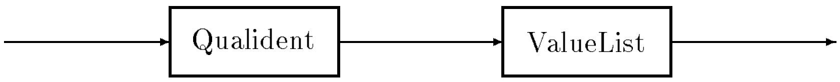
Designator



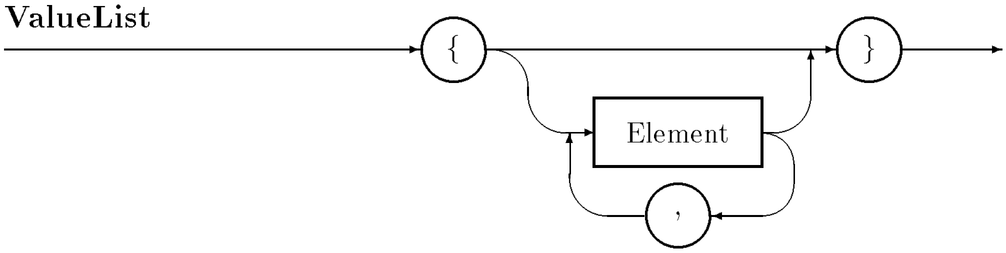
Qualident



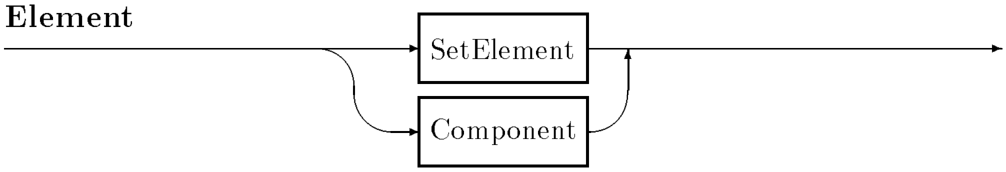
Constructor



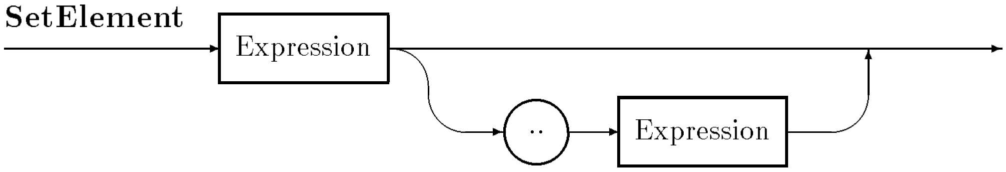
ValueList



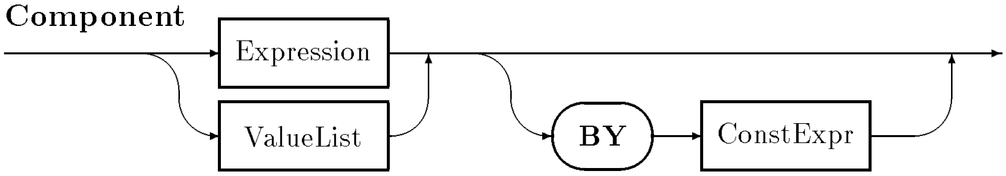
Element



SetElement

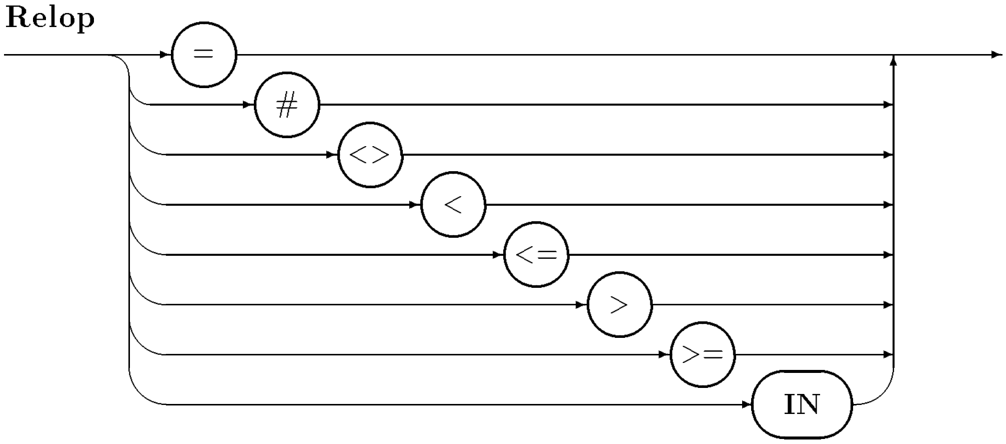


Component

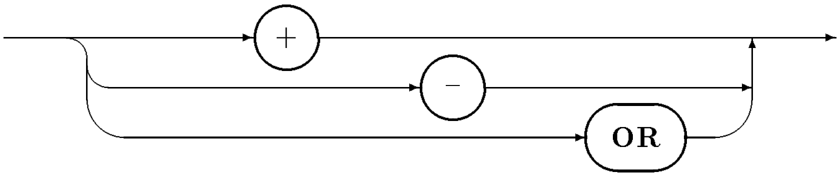


ConstExpr

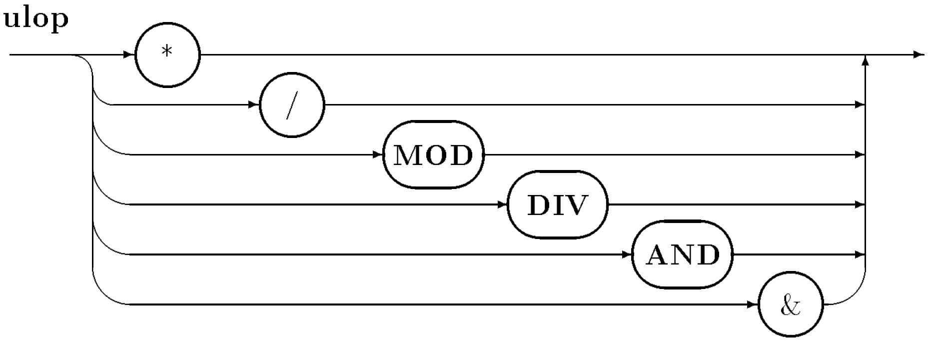
Relop



Addop



Mulop



Part II

Technical Reference

Chapter 7

The Compiler Environment

7.1 Overview of the System

Gardens Point Modula is a Modula-2 programming environment which brings the advantages of Modula to the *UNIX* environment. The close interworking of Modula programs and the standard tools and libraries of the *UNIX* environment is a goal of the development. **gpm** will provide a uniform programming environment over all of the hardware platforms on which it is available. The developers of the system are firmly committed to implement all aspects of the emerging ISO draft standard for Modula, and to provide portability at the source code level between implementations.

The software consists of the following components

- the compiler **gpm**
- the load-builder **build**
- a profiling load-builder **bldprf**
- an order of compilation analysis tool **gpmake**
- a cross-reference generator **gpxrf**
- standard libraries
- special libraries

7.1.1 The Compiler

The **gpm** compiler is written entirely in Modula, and has a target-independent front-end which performs all syntactic and static semantic checks on the source program. The compiler builds a complete representation of the compilation unit in computer memory and performs all analysis on this internal representation.

After performing a number of transformations on the code of the source program, **gpm** emits its intermediate output code. The preliminary release of the **gpm** compiler on each new target architecture emits output using language *C* as an assembly language. This assembly output is further compiled by the standard *C* compiler **cc**. All of this is transparent to the user, and the whole two-stage compilation process is invoked with a single command. In some of the following, reference is made to the native code versions currently in test form.

Internally, the program **gpm** consists of a small driver program which parses the command line arguments, *fork*s and then *exec*s **gpm2**, the compiler proper. When **gpm2** terminates it passes back a result code to the driver. In response to this exit code, **gpm** does one of three things. If the source code was a definition part, or errors were detected, **gpm** terminates. Alternatively, if the compiler was running in interactive mode and the user responded 'v', **gpm** chains to **vi**. Finally, in the event of a successful compilation **gpm** *fork*s again and this time *exec*s **cc**. When **cc** terminates, the driver program removes the intermediate code file, unless the **−S** (assembler) option was specified. In the case of multiple source-file arguments on the command line, **gpm2** is forked repeatedly to compile each of the specified files.

The **gpm** driver is written entirely in Modula, and the source code is included in the distribution. It is a good example of the way in which programs may use the underlying *UNIX* system calls to manipulate processes.

It must be stressed that **gpm** is a complete *compiler*, and deals correctly with *all* of the constructs of Modula, and not just a *translator* for those constructs which have an equivalent in *C*. There are a number of immediate consequences of this high level assembler approach. The level of possible integration between Modula programs and the *C* libraries is greatly enhanced, and the portability of the compiler itself ensured. However there are a number of other considerations.

Compilation Speed

Many vendor supplied *C* compilers are rather slow by modern standards, but most produce quite good quality code. The **gpm** front end is exceptionally fast, despite the very comprehensive checks and transformations which it carries out. Early implementations indicate that it so much faster than **cc** that compilation speed is essentially dominated by the second stage of the process.

As an example, on the *HP9000/840* the **gpm** front end produces its assembly level output at between 30000 and 60000 lines per minute, depending on the number of imported symbol files. The overall compilation speed, when chaining to **cc**, is 5000 to 7000 lines per minute. Faster processors are correspondingly faster.

Object Code Quality

The final object code quality depends on the quality of **cc** which, as noted above, is generally quite good. Early measurements show that the quality of the final object code is comparable with that of *C*, and in many cases the speed of applications exceeds that of implementations of the same algorithms directly in *C*.

The reason for this rather surprising phenomenon is the front-end transformations which **gpm** carries out on the source representation. The *C* which **gpm** produces is often quite different to that which a human programmer would produce for the same purpose.

Generally however, sensible programmers will compile Modula code into object code which has runtime checking enabled. In this case, the speed will be less than that of the equivalent, unsafe *C* version, but not by a very great factor. **gpm** performs extensive analysis of the source code, and removes a large number of runtime checks which it is able to prove are unnecessary. When used in this way, the compiler produces code which is much faster than comparable Pascal compilers.

7.1.2 The Load-builder

The load-builder program **build** does three things. It performs an analysis of the compiled code of the modules, so as to produce a linker script to build the final executable file. It analyses the dependencies between the modules so as to determine which of the modules need initialization. Those which do are initialized exactly once, and in the order required by the importation graph. Finally, **build** checks that all modules have been compiled with consistent versions of the symbolic interface definitions.

Internally **build**, like **gpm**, consists of a small driver program. This driver parses command-line arguments, *fork*s and *exec*s the load builder proper **build2**. This program performs its analysis, and emits a short code file containing the initialization call chain, and a linker script. When **build** regains control it chains to the standard *UNIX* linker **ld**. Finally, the script and code file are deleted.

build does its analysis by reading the *reference file* of each module. First, the reference file of the base module is read, and names and cryptographic checksums of all imported modules recorded. The reference files of all imports are likewise read, and the imports of *those* modules. Diagnostics are issued if inconsistent checksums are detected.

As a result of this structure, the **gpm** system is able to use the standard *UNIX* linker, without either losing the safety of Modula's strict version consistency checks, or involving the user in the manual construction of long linker scripts.

7.1.3 The Profiling Load-builder

The profiling version of the load builder **bldprf** actually uses the same executable file as **build**. The directory entry **bldprf** is linked to the same program using the command *ln*. When **build** begins, it fetches its first argument to determine the version of the task which it is being asked to perform.

7.1.4 The gpmake Utility

The utility **gpmake** is an extremely clever recompilation analyser. The program reads all the source files associated with a particular program which are accessible in the current directory. The program determines the module dependencies, and determines an order of recompilation which is consistent with these dependencies. It also determines which of these recompilations, if any, are required.

As a result of this analysis, **gpmake** issues commands to compile all necessary modules of the program in the correct order. As an alternative, the program issues a script for a complete recompilation. In this case the user may edit different compile-time options into different lines of the script file, should that be required. In either case, library modules which are not in the current directory are not recompiled.

7.1.5 The Cross-reference Generator

The utility **gpxrf** produces a cross-reference listing of all occurrences of identifiers in the compilation unit. The listing is produced either in lexicographic ("alphabetical") order, or in order of frequency of use.

7.1.6 The Standard Libraries

gpm will make all of the proposed ISO standard libraries available in due course. Interim versions also have a number of old-style libraries, as described in Wirth's *Programming in Modula-2*.

7.1.7 The Special Libraries

There are a number of non-standard libraries supplied. The chief purpose of these is to provide access to *UNIX* system calls and other facilities.

7.2 Environment Variables

The compiler uses eight environment variables. Three of these are used in the file search strategy, two are used to set compile-time table sizes, a fifth is used to define a build-time buffer size, a sixth sets the runtime stack-test limit, and the final variable chooses the interactive editor.

The variable M2SYM

When looking for symbol files, or for the header files of *foreign* modules, **gpm** looks first in the current directory, then in each directory on the path defined by the environment variable *M2SYM*. The value *$M2SYM* is a sequence of one or more path-names, separated by colon characters ':'. The compiler uses the same path when searching for the file **m2errlst.dat** which contains the table of verbose error diagnostics.

In many cases, the path *$M2SYM* will consist of a single directory. It may also be convenient to place the human readable definition files in the same directory.

Environment Variables and gpmake

gpmake depends upon finding the definition files of library modules, the **.def** files, on the path *$M2SYM*.

The variable M2LIB

When looking for reference files, **build** looks first in the current directory, and then in the directories of the path defined by the environment variable *M2LIB*. The value *$M2LIB* is a sequence of one or more path-names, separated by colon characters ':'.

The build program depends on the fact that each reference file and its associated object file will be in the same directory. If this rule is broken, then the linker will complain that it cannot find one of the object files. It is common to have *$M2LIB* consist of a single directory, and to place the object files and the associated reference files of all library modules in this directory.

The variable GPNAMES

The compiler, load-builder, and all associated tools (**gpmake**, etc) use the file lookup strategy defined by the library module *GpFiles*. The environment variable *GPNAMES* controls both the order of lookup of filenames in lower-case-only, mixed-case and lower-case-truncated-to-DOS-length, and also the use of mixed case names for files created by the gpm programs. If *GPNAMES* is undefined, or has the value `"lower"`, the module name is converted to lower case and this is used as the first choice for lookup and as the base for output file names (**.syx**, **.rfx**, **.o**). If *GPNAMES* is `"Mixed"`, the module name as found is used for both purposes. On lookup, if the first choice file name is not found, the other alternative is tried, followed if necessary by a file name truncated to DOS length (and always lower case). This strategy provides backward compatibility with earlier versions of gpm which always generated lower-case names, and provides generous lookup matching. However, if mixed case in output file names is required, *GPNAMES* must be set to `"Mixed"`.

The variable M2HASH

This variable sets the number of entries in the compiler's hash table used for fast identifier lookup. It must exceed (preferably by a factor of 2, for speed) the total number of distinct identifiers in the compilation unit and the definition modules it imports. The default value is 5987, but any value up to 65521 may be selected. The size must be a prime, but if *$M2HASH* is not prime the compiler will use the next lower prime (the upper limit is the largest prime less than 2^{16}); a small saving at each invocation of the compiler is achieved by choosing a prime value.

If the number of identifiers exceeds the hash table size, the compiler halts with the message "Warning: hash table near full".

The variable M2STRING

All the identifiers (and also literal strings and set bitmaps) are stored in a string table; the size of this table is set to *$M2STRING*. This must clearly exceed the total length of identifiers, but unlike the hash table no speed penalty is incurred if the spare space is small. The default value is 64512, corresponding to an average identifier size of about 10 characters for the default limit of 5987 identifiers. Clearly, *$M2STRING* should be increased in step with *$M2HASH*, with a 10:1 ratio a good guide. In special cases (eg long identifiers as in XWindows interfaces), a larger ratio may be appropriate.

If the total string length exceeds the table size, the compiler halts with the message "String table overflow".

To assist with tuning *$M2HASH* and *$M2STRING*, note that the compiler's super-verbose output (-V option) includes a usage summary for both tables. Note too that large settings may cause the C compiler to fail with table overflow; if so, check man cc to see if your compiler has options to expand its tables.

In the case of **gpm-pc** the defaults are smaller, to suit the restricted memory.

The variable SOAPSIZE

gpm uses a separate stack for local copies of value-mode open array parameters. The default size of this stack is 4096 bytes. This size has proven to be more than adequate for all normal programs. It is possible to increase this allocation at link time.

The value of this variable is checked by **build** during the linking phase. The value is thus determined at *build* time, rather than at *compile* time. **gpm-pc** uses extensible stack frames, and does not use the soap mechanism.

The variable M2STACK

This variable sets the limit against which the stack overflow tests are made. Remember however that stack testing is *off* by default. The variable states the size of the stack in bytes. If the variable is not set, the stack defaults to 64K bytes. The symbol *M2STACK* is a decimal numeric string, specifying the stack size in bytes.

The value of this variable is checked by **build** during the linking phase. The value is thus determined at *build* time, rather than at *compile* time.

The variable GPMEDITOR

This variable sets the editor which the interactive mode of **gpm** chains to when errors are detected, and tells **gpm** how to start the editor on the correct line.

The environment variable *GPMEDITOR* is of the form *edName* $\{arg \mid \%\}$, where as ususal the braces denote optional repetition, and the vertical bar denotes choice. In the case of the editor *vi* the variable might have been defined to the Bourne shell by the command

```
GPMEDITOR="vi +# %"
```

In this format *edName* is the name of the chosen editor, such as *vi* or *emacs*. The remaining arguments may appear in any order, with the following macro substitutions — the percent sign % is replaced by the name of the file to edit, while any embedded # sign denotes the line number on which the editor is to be started. Thus with the above definition, if the program *foo.mod* has an error on line 23, interactive mode will invoke the editor *vi* with command string `"vi +23 foo.mod"`. As a second example, if the environment string was `"uemacs -g# %"` the same program would have invoked the editor *uemacs* with the command string `"uemacs -g23 foo.mod"`.

In the event that the variable *GPMEDITOR* is undefined, the default editor *vi* is invoked.

Chapter 8

Command-line Options

The compiler and load-builder program both accept a number of options from the command line. In each case the syntax is quite simple. If any options are to be passed, they must be in option strings which follow after the command (**gpm build** or **bldprf** as the case may be), but precede any filenames.

Option strings start with a minus sign '–' and options may appear in one or more groups. The order and grouping of options is never important. Thus '`gpm -S -Of -i -r -I foo*.mod`', '`gpm -IirOfS foo*.mod`' and '`gpm -SOfirI foo*.mod`' all have a precisely equivalent effect. The programs will complain if too few arguments are passed to them.

Options are listed here in functional groups, and then in a single alphabetical sequence. The alphabetical listing has more detailed explanations of the effect of each option.

8.1 Compiler Options

8.1.1 Flags grouped by function

Listing and message control

–d	suppress console warning messages
–l	produce listing file
–v	produce verbose messages and listings
–V	version date is displayed, symbol module key-values are traced, and verbose messages and listings are produced
–X	a detailed explanation of each error is given on screen and in the listing (if selected)

Runtime error checking

–a	suppress runtime assertion checks
–i	generate code without array index checks
–r	generate code without value range checks
–s	runtime stack tests are turned off
–t	runtime arithmetic overflow checks off

Compiler control flags

–f	force compiler to produce reference and object files with names based on the filename rather than the module name
–g	compile with extra information for the debugger program (either *xdb* or *dbx*)
–n	compilation checks only, no object code is produced
–I	run in interactive mode, with option of jumping to *vi*
–Oc	optimise for compactness and speed
–Of	optimise for maximum speed
–p	compile with procedure call-count code for the runtime profiler
–S	intermediate code is persistent

8.1.2 Flags listed alphabetically

–a "**a**ssertion-off" — assertion tests in the source code, invoked by the system procedure *Assert*, are ignored by the compiler and generate no code

–d "**d**angerous" — warning messages to the screen and list file (if selected) are suppressed. However, if a program has both errors *and* warnings both will go to the screen and listings in spite of the **–d** option

–f "**f**ilename" — force the compiler to name reference and object files based on the source filename rather than the module name. This option is illegal when applied to program modules. It is intended to allow several implementations of the same module to exist in the same directory, and to be selectively linked using the query option **–q** of **build**

–g intermediate code is compiled with markers for the debuggers (*xdb* or *dbx*). This option is not required to simply obtain a postmortem procedure call-chain listing. With **gpm-pc** this flag leaves line number marks in the d-code output so that line numbers will appear in the stack unwind

–I "**I**nteractive" — the compiler halts on errors, displays the verbose version of the diagnostic, and prompts the user to continue, quit, obtain more information or jump into the chosen editor

–i array **i**ndex tests are turned off (same as (* $I– *) in the source file)

–l a **l**ist file *name*.**lst** is generated from input file **name.def** or **name.mod**

–n "**n**o-code" — the compiler performs syntactic and semantic checking, but no object code is produced. This allows a speedy check of modified code, before a complete recompilation is attempted

–Oc **O**ptimise so as to produce **c**ompact code, even at the price of small speed loss (same as (* $C+ *) in the source file).

–Of **O**ptimise so as to produce **f**ast code, even at the price of larger memory size. This flag has a similar effect to the (* $F+ *) in the source file, but affects code generation strategy as well as front-end program transformations. Use of this flag is thus preferred to the use of the program switch statement

–p compile with **p**rofiling code so that **bldprf** and *prof* can produce procedure call-counts as well as time statistics[1]. See the relevant chapter of the technical reference manual for more on the profiling facilities

–r **r**ange checking of assigned values and actual value parameters is suppressed (same as (* $R– *) in the source file)

–S "aSsembler" — the *C* language intermediate code is persistent in the file **./***name***.c**, no object code is produced. This flag is only of use for debugging mixed language programs

–s **s**tack overflow checking is suppressed (same as (* $S– *) in the source file). *Stack overflow checking is currently turned* **off** *as the default. It may be explicitly turned* **on** *by the use of the pragmas in the code*

–t arithmetic overflow tests are suppressed (same as (* $T– *) in the source file). *Arithmetic overflow checking is not used in C-producing versions, but is standard on all native code versions*

–v **v**erbose screen messages are produced, and verbose listings also, if **–l** is also specified

–V "super-**V**erbose" — the **V**ersion date and time of the compiler are displayed, the key-values of all directly and indirectly imported symbol modules are traced, and verbose messages and listings are produced

–X e**X**plain — a detailed explanation is given for each error which is detected

8.2 Builder Option Flags

–D "**D**ebug" — the program is linked with the runtime stack unwinder library so that debugging information is available if the program crashes (DECstation only)

–L*name* the specified directory *name* is searched for libraries

–q "**q**uery" — the builder prompts for the names of reference files which it cannot find. Only the base-name of the file is required, and querying may be aborted by entering a blank line

–S "aSsembler" — the initialization-call-chain code file and the linker script are persistent as files "`modbase.c`" and "`modbase`" respectively

–v **v**erbose messages are displayed on the screen so that progress may be monitored, and the origin of any error messages determined

–V builder **V**ersion date and time are displayed, and screen messages are verbose

To build a profiling version of a program, build is invoked under the name **bldprf**. The option and filename conventions for **bldprf** are identical to those of **build**. Information on using the profiling facilities are given in the chapter titled **Using the Profiling Tools**.

[1] If any modules of a program are compiled with this option, the program must be built with **bldprf**

Chapter 9

Implementation Specifics

This chapter includes details of **gpm** which are specific to this implementation. This includes pragmas, special features, implementation restrictions and extensions. There are also details of the primitive types and their representation, and the way in which **gpm** builds structured types from these basic building blocks.

9.1 Pragmas and Compiler Switches

9.1.1 Source code switches

The compiler recognizes a number of switches in source code, which allow various options to be turned on and off. These are introduced in the traditional way for Modula, by means of a *pseudo-comment*. All of these switches are in the form of Modula comments in which the first non-whitespace character of the comment is a dollar sign $. These allow the switching on and off of runtime checks and optimizations, and the restoration of previous values.

The precise format of the compiler switches is described by the following extended-*BNF*[1]

comment	→	'(*' [**switch**{','**switch**}] any string '*)'.
switch	→	'$' **optionChar commandChar**.
optionChar	→	'R'| 'I'| 'F'| 'T'| 'C'.
commandChar	→	'+'| '–'| '='.

Whitespace is allowed between switches and between the opening comment delimiter and the first switch, but the dollar sign and the option character must be adjacent. The meaning of the command characters is '+' *activates* the option, '–' *deactivates* the option, while '=' *restores* the previous value of the same option. There is a limit to the number of options values which can be stacked for later restoration. This limit is currently 8.

The options which may be controlled in this way are

$I controls the emission of index bounds checks on array accesses

$R controls the emission of range checks on assignments, passing of value parameters, and the incrementing and decrementing of values of ordinal types

[1] It should be noted that the next release will offer a new format for option switches, using the format proposed by WG-13: <* *switch-id* *>. Of course **gpm** will continue to support the current format, for backward compatability

$F controls the application of optimizations to the compilation process. This switch controls those transformations of the program which will increase speed of execution, possibly at the expense of code size

$C controls the application of optimizations to the compilation process. This switch controls those transformations of the program which will decrease the code size of the program, even at the expense of a slight decrease in speed

$T controls the emission of arithmetic overflow tests in native code versions.

Interaction with command-line arguments

All of these options are able to be set globally from the command line. The interaction between command line option arguments and the compiler switches is governed by the following principles.

> The initial values of the *I* and *R* options are set from the command line. Both tests are **on** by default but may be switched off by the **–i** and **–r** options. The switches in the source program can then change these initial values at will.
>
> The initial values of the *F* and *C* options are set from the command line. Both optimizations are **off** by default, but may be turned on by the **–Of** and **–Oc** options. The switches in the source program may change these values at will, but such changes only control transformations in the compiler front end. The use of either option from the command line causes a higher level of optimization to be applied to the code generation in the back end, irrespective of any changes introduced in the source program by the use of switches.

9.1.2 Stack overflow testing

Programs which do not use the coroutines library, so-called *single-stack programs* have little need to perform stack overflow testing. Typically, several hundred megabytes of virtual memory are available for expansion of the stack segment of such programs, although it is usual for *UNIX*'s process size limit to be exceeded well before this. Programs which use the coroutines library have a separate stack for each coroutine, suggesting the prudent use of stack overflow testing. The facilities provided for this are also available for single stack programs, although the default continues to be for stack overflow testing to be disabled except in **gpm-pc** which has stack overflow testing as the default.

In some environments it may be useful to perform stack overflow testing even on single stack programs. For example, in a student environment, it may be useful to reduce the size of the core dumps which are produced when student programs recurse in a berserk fashion.

The facilities for stack overflow testing consist of the following items.

- Stack testing in turned on with the pragma `(* $S+ *)`
- Stack size limits are set by a environment variable *M2STACK*
- A new runtime message signals stack overflow

Setting the stack size limit

The load builder program **build** queries its environment for the variable *M2STACK*. The value, if set, is inserted in the tiny program which contains the initialization call-chain for the final, executable program. The value in the tiny program is used in the initalization of the runtime support system.

If the variable *M2STACK* is not defined, the runtime system allocates a default size of 65536 bytes (16000 bytes in **gpm-pc**) to the stack. If this default is unsuitable a larger value should be placed in the user profile. Note that this value is effective only for the main process. The stack size of every other coroutine is determined separately by the third parameter to the the call of *NEWPROCESS* which created it.

Miscellaneous information

Stack overflow testing is performed by a simple inline statement. The cost of this test, in terms of time taken, is approximately that of a simple assignment statement to a global variable. The cost is thus similar to the inline *Assert* "procedure".

If stack testing is turned on by use of the pragma `(* $S+ *)` and the stack exceeds the set limit a runtime error is raised. In the absence of exception handling, the program is aborted with the following message being sent to the *StdError* stream.

```
**** m2rts: stack limit has been exceeded ****
```

9.1.3 Pragmas in the definition part

There is a facility built into the compiler which allows special information to be given about modules, which is used in the program transformations which the **gpm** front-end performs. The use of these facilities in the supplied libraries adds significantly to the quality of the code generated for user programs, even although the user programs themselves do not use the pragmas. **All of these facilities may be safely ignored by the applications programmer.**

gpm recognizes a number of *context sensitive marker* symbols immediately prior to the keyword *DEFINITION*. These are the pragmas !*LIBRARY* and !*SYSTEM*, and the special markers *FOREIGN* and *INTERFACE*.

!LIBRARY! definitions

There are a number of program transformations which are only safe to perform if it is known that particular procedures cannot be recursive, or cannot modify their *actual* parameters along indirect paths. The !*LIBRARY*[2] pragma gives the compiler a useful hint, by promising that the facilities of the module will be well-behaved in this respect.

In particular these modules promise that calls to procedures exported by the module cannot be involved in indirect recursion with code outside of the library system. When the compiler meets the implementation of such a module the code is rigorously checked to see that it keeps this promise. The code of these modules can only call external procedures which have made the same promise (and have been similarly checked). The load-builder **build** also

[2]The use of the closing exclamation point is optional

checks that the pattern of importation within the library modules is strictly heirarchical (that is, there are no cycles in the *imports* graph which involve *!LIBRARY* modules).

!SYSTEM! definitions

This marker signals to the compiler that no implementation is required for the module. Usually this means that the facilities of the module are built into the compiler, or into the runtime system. These modules have symbol files in the usual way. Although the compiler knows about the implementation, it must read the symbol file before it will allow use of the module's exports in a user program.

Users may mark their own modules in this way, provided the module does no more than define types and constants. See the standard library *Ascii* for an example of this.

FOREIGN definitions

This marker signals to the compiler that the facilities of the module are implemented outside of the Modula environment. Such modules have definition and symbol files in the usual way, but do not have reference files associated with their object files. Instead each of these modules has an optional non-standard *IMPORT* statement which marks the symbol file so that programs using the module know to link the object file resulting from the foreign implementation.

As an example, the supplied library module *Terminal* has the following definition file

```
FOREIGN DEFINITION MODULE Terminal;
  IMPORT IMPLEMENTATION FROM "terminal.o";
  ...
```

The *FOREIGN* marker tells the compiler that no reference file will be produced, the import statement ensures the programs using the module will automatically include the file `terminal.o` in the linker script. The object file may be derived by compiling from any available language, and will define external symbols such as `Terminal_WriteLn`, `Terminal_WriteString` and so on.

INTERFACE definitions

This marker signals to the compiler that the facilities of the module are implemented in C, and that the names of the symbols should not be changed by the normal process of concatenating module and symbol names. Such modules are used to provide an interface to C-language libraries where the external names are already determined. Once again, a non-standard import statement is used to signal to the linker that particular object files should be linked, or particular libraries searched. For example, consider the following interface fragment —

```
INTERFACE DEFINITION MODULE GammaFunc;
  IMPORT IMPLEMENTATION FROM "-lm";
  VAR signgam : INTEGER;
  PROCEDURE gamma(x : REAL) : REAL;
END Gamma.
```

The *INTERFACE* marker tells the compiler that the function procedure really does have name *gamma*, and not the otherwise expected *GammaFunc_gamma*. The import statement ensures that the linker will search the math library **–lm**.

The use of these modules is essential to the integration of Modula with the standard *UNIX* facilities. All necessary details are given, for those applications which require to generate foreign or interface definitions, in the chapter *Interfacing to Other Languages*.

9.2 Omissions and Limitations

9.2.1 Omitted constructs

The compiler currently does not support the following constructs

- multi-dimensional open array parameters are not supported yet. Support for these will be in the next release
- runtime value constructors other than sets are not supported. ISO WG-13 has proposed such constructs, and we will support these as soon as the proposal is firm. The current release permits record and array constructors only for constant values. This limited implementation is most valuable.
- function procedures currently cannot return array values, they may however return records. This limitation will be removed in the next release
- the use of module priority has no relevance to programs running under control of a multi-user operating system such as *UNIX*. Priority specifications are allowed, but produce a compiler warning message that the specification has been ignored
- the optional variant tag specification parameters to *TSIZE* and *NEW* and *DISPOSE* are permitted, but ignored. A compile-time warning is issued
- **gpm** does not yet permit sets of ranges which have lower bounds less than zero. Set base types currently must be either: *CHAR* or its subranges, enumerations or subranges of enumerations, or subranges of [0 .. 255]

9.2.2 Included constructs

There are a number of new constructs which have entered the Modula language recently, or are proposed by ISO WG-13. With the omissions noted above, **gpm** implements all of the current syntax and semantics. In some cases the compiler accepts obsolete syntax also, and flags such usages with a warning only. The old form of tag-less variant record is accepted and flagged, and the inclusion of explicit export lists in definition parts is permitted but ignored. Old style type-casts which use the *type transfer function* are allowed but attract a warning message. Use of type transfer functions is discouraged, as the new system function *CAST* fulfills the same purpose with greater safety and enhanced functionality.

The following new or changed constructs are permitted, and have semantics as specified by the ISO proposals. Further details on some of these constructs may be found in the release notes

- literal character strings may be assigned to arrays of *CHAR* even where the lower bound of the array is not zero
- a new pervasive identifier *LENGTH* may be used to return the length of character strings. Since this is a built-in procedure, it may be used in declarations
- the empty string is compatible with the type *CHAR* as well as with other arrays of *CHAR*. As a character, the empty string has a value equal to the string terminator (*Ascii.nul* in this case).
- declarations of procedure types may include forward references to types in their formal parameter lists. This is analogous to the familiar way in which pointers are declared which are bound to target types which have not yet been declared. A function procedure type may therefore be defined which returns its own type
- *FORWARD* is now a reserved word. Although **gpm** does not require forward declarations of mutually recursive procedures, it recognizes the keyword.
- the new system functions *INCADR, DECADR* and *DIFADR* for "portable" address arithmetic are supported, as are the system functions *SHIFT* and *ROTATE* for the type *BITSET*
- *NEW* and *DISPOSE* are back in the language. These standard procedures map into calls to the procedures of name *ALLOCATE* and *DEALLOCATE* which are visible in the current lexical scope. The conformance of these substitution procedures with the expected formal procedure type is checked
- function procedure *SIZE* allows either a type-name or a variable name as actual parameter. The function is *pervasive* (rather than in *SYSTEM*), and returns a value which is compatible with either *INTEGER* or *CARDINAL*
- complete checks on threats to *FOR* loop control variables are implemented, including threats from uplevel access within nested procedure bodies
- procedure constants may be declared as a mechanism for procedure renaming. When used in definition parts, it allows export of a renamed version of a procedure imported from another module, thereby allowing strictly layered software designs to avoid the call overhead of an encapsulating procedure
- constant value constructors are permitted for arrays (including multi-dimensional arrays), and for *fixed* format records (that is, those which do not have any variant part). These constructors have a syntax which is an easy extension of the the form for set constructors, which is shown in the syntax diagrams in the *User Guide*. This construct is useful for initializing arrays and records, and for setting up constant tables. More detail is given later in this chapter.

9.2.3 Compiler limits

Number of identifiers

The default maximum number of identifiers which **gpm** will allow is 5987; this may be increased via the *M2HASH* environment variable as described in Section 7.2. This number includes the names of all declared and imported objects, including such things as enumeration constant names and record field names. All of these names are held in a *string table* during compilation, along with literal strings and the bit-patterns of set constants. The default space allocated to the string table is $63K$-bytes, and may be increased via the *M2STRING* environment variable.

Memory use during compilation

gpm builds a complete abstract syntax tree of the compilation unit during compilation, and builds descriptors for all declared and imported types. All of these are built in heap memory, and have no significant limitation in size.

There is no limit to the size of code which **gpm** can compile. In theory, with very, very large programs **gpm** will start to exercise the demand paging of the host machine, and the compilation will therefore slow down somewhat. This phenomenon has never been observed in practice however.

A rough rule of thumb is that **gpm** will allocate about 3 bytes of heap space for every byte of source code processed. For compilation units which import a large number of symbol files this figure has been known to increase by a factor of almost two.

Other limits

There is a limit to the complexity of addressing expression which **gpm** allows. A designator such as `a^.b[x]^.c[y]^.d^.e ...` will overflow an internal buffer if the number of dereference operators `^` exceeds about 12. Clarity, and good programming practice would suggest a somewhat lower limit.

The **build** program has no limit to the number of modules which it will handle.

In a procedure declaration, the maximum length of a sequence of formal parameters which may share the same mention of a formal typename is 15. The explanation for error 226 in the chapter *Interpreting compiler diagnostics* explains how to evade this restriction.

9.2.4 Symbol file keys (magic numbers)

All Modula implementations ensure version consistency by attaching a *magic number* to each symbol file. If two modules of the same program each import some third module, then it can be checked that both modules used the same version of the interface definition which the symbol file represents. With **gpm** the reference file associated with each object file contains a list of the magic numbers of every module on which that module depends. These magic numbers are carefully checked by **build**.

Unlike many Modula systems, **gpm** does not use magic numbers based on date stamps for ensuring consistency of modules when building. Instead it uses a cryptographic checksum based on the total information in the symbol file.

The consequences of this deliberate choice are as follows. Definition files may be recompiled at will. The accidental recompilation of a low-level file will not necessitate the recompilation of any dependent modules, unless there has been some real change in the interface which the definition part specifies. It is possible to edit a definition part file so as to include new or expanded comments, and this will not affect the symbol file, and hence the magic number. It is even possible to change the names of the formal parameters of exported procedures, and since these are not recorded in the symbol file the magic number will not be affected. However, the slightest change to the number of parameters, their types or modes will be instantly detected.

9.2.5 Miscellaneous Information

Using functions in constant declarations

Compile-time constants may be defined using any expression which is able to be evaluated at compile time. These expressions may use use all of the built-in expression operators and *some* of the standard functions. The functions which are permitted are *ABS, CAP, CHR, FLOAT, SFLOAT, LFLOAT, LENGTH, MAX, MIN, ODD, ORD, SIZE, TRUNC* and *VAL*. Needless to say, in all cases the actual parameters to these functions must be constant expressions.

The use of the *VAL* function perhaps requires some explanation. The function has many legitimate uses in constant expressions, but can introduce unwanted restrictions if used unneccesarily. For example, literal numeric constants such as 11 have an internal type *ZZ* which is compatible with both signed and unsigned types. If the declaration

```
CONST eleven = VAL(INTEGER,11);
```

is used, then the constant *eleven* will no longer be expression compatible with the unsigned types. In this example, unless there is a reason to deliberately restrict the compatibility, it is best to simply say

```
CONST eleven = 11;
```

Amorphous open array parameters

Objects which are passed to amorphous open arrays (arrays of *BYTE* or *WORD*) must obey two simple rules. They must be at least as stringently aligned as the formal element type, and they must possess a runtime address.

The first restriction prevents arrays of characters from being passed to an open *ARRAY OF WORD* formal parameter. The second prevents the result of an expression evaluation (other than large sets) from being passed to open arrays. The exception for large sets arises from the fortuitous circumstance that large set expression evaluations require the allocation of memory temporaries. Use of the exception is likely to be non-portable to non **gpm** implementations. The second restriction also rules out literals and constants other than sets and strings.

Casts and coercions

There are restrictions on the casts which **gpm** can perform, which are hardware originated. These restrictions are closely allied to those for amorphous open arrays, discussed above.

In summary **gpm** will allow casts between word-sized and smaller values in a free fashion, including expression evaluations. Neither the source nor the destination type may be an array type however. For types of arbitrary size, casts are possible between quite differently sized objects as proposed by ISO WG-13. However, the destination type must be at least as stringently aligned as the source type, and the value to be cast must have a runtime address.

It is possible, for example, to cast a short string literal value to a very large *CHAR* array type. This would enable indexing into (a copy of) the entire constant data section of the runtime representation of the program. Sensible users will treat this new facility with considerable caution.

9.2.6 Constant value constructors

ISO WG-13 has proposed value constructors for arrays and records, with a similar syntax to set constructors. **gpm** has a partial implementation of this proposal in the current release. It is not entirely clear, at the time of writing whether constructors evaluated at runtime will finally be approved. Nevertheless, constant constructors are *very* useful for many purposes, and are supplied for that reason.

The extended-*BNF* syntax fragment for constructors is as follows –

constructor	→	**typename valueList**.
valueList	→	'{'**element** {',' **element**}'}'.
element	→	**setElement** | **component**.
setElement	→	**expression** ['..' **expression**].
component	→	**constExpression** [BY **repeatCount**]
	|	**valueList** [BY **repeatCount**].
repeatCount	→	**constExpression**.

The replicator clause only applies to array constructors, even although it is sometimes meaningful for records which have repeated fields of the same type. Semantic restrictions include the fact that any repeat counts must be non-negative, whole-number constants, and the total number of elements (taking into account any repetition of elements) must exactly match the number of elements in the structure. Values must be assignment compatible with the component type to which the equivalent assignment is being made. Thus a valid record initialization might be –

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

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

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

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

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

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

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

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

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

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

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

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

9.3 Size and alignment of data items

Details of the size and alignment of Modula variables may be needed in some instances, for example,

1 in restricted memory situations, where the required variable space may exceed that available;

2 in low-level applications, or those which link to other language environments, where it is essential to know what the memory layout is.

The correspondence between **gpm**'s types and emitted C types is given below, for the HP 9000/8xx version, together with the resulting size and alignment constraints as specified in the **HP C Programmer's Guide**. Note that all gpm versions have the same *size* for all these types, but the alignments may vary. In particular, most Motorola 680x0 processor versions, and **gpm-pc** never align fields more strictly than on an even address boundary; the Sony NEWS/M68K uses 4-byte alignment.

gpm TYPE	C type	Size (bytes)	Align	Alias
CHAR	unsigned char	1	1	(typdef unsigned char un_chr)
BOOLEAN	char	1	1	(typedef char BOOL)
BYTE	unsigned char	1	1	
INTEGER	int	4	4	
CARDINAL	unsigned	4	4	
WORD	int	4	4	
REAL	double	8	8	
LONGREAL	double	8	8	
SHORTREAL	float	4	4	
BITSET	unsigned	4	4	(typedef unsigned BITSET)
PROC	void (*)()	4	4	(typedef void (*PROC)())
ADDRESS	char *	4	4	(typedef char *ADDRESS)
enumeration	unsigned char	1	1	
subrange	*note 4*			
set	unsigned []	4n	4	(* see note 1 below *)
record				(* see note 2 below *)
array				(* see note 3 below *)
POINTER	*targetType* *	4	4	

Notes

1 Set size is the number of whole words needed to hold a bitmap of the possible members of the set. Thus a SET OF [0..n] requires 1 word for n up to 31, 2 words for $32 \leq n \leq 63$, and so on up to 8 words for $224 \leq n \leq 255$, the latter being the implementation limit for set size.

2 A record translates into a C struct; thus

```
VAR
  record : RECORD
             int  : INTEGER;
             char : CHAR;
           END;
  ...
  record.int
```

translates to:

```
static struct t_BA {
        int _int;
```

```
        un_chr _char;
} record;
...
record._int
```

Note that in this case the record type is anonymous, and so the struct has the synthesised name `t_BA`. If it were declared as the named type *Thing* the resulting typedef would be:

```
typedef struct tThing {
        int _int;
        un_chr _char;
} Thing;
```

A variant part translates to a C union, with each variant a struct; the entire union is a field within the struct corresponding to the enclosing record:

```
VAR
  variant : RECORD CASE b : BOOLEAN OF
                   | TRUE : trueForm : INTEGER;
                   | FALSE : falseForm : BITSET;
                   END;
            END;
  ...
  variant.trueForm
```

translates to :

```
static struct t_AA {
        BOOL b;
        union {
                struct {
                        int trueForm;
                } _a;
                struct {
                        BITSET falseForm;
                } _b;
        } _0;
} variant;
...
variant._0._a.trueForm
```

Given this structure, the size and alignment of a record may be calculated:

(1) The size is the sum of the sizes of the fields, with extra 'padding' where necessary to satisfy alignment requirements

(2) the alignment is the strictest alignment required by any field

In detail:

(1) Knowing the size of each field, they can be assigned sequential offsets within the record; thus for example:

```
Ex1 = RECORD
        card : CARDINAL;
        int : INTEGER;
        ch : CHAR;
      END;
```

would have field 'card' in bytes 0-3 inclusive, 'int' in 4-7, and 'ch' in 8. However, the record must end on a boundary corresponding to the strictest alignment within it, so that bytes 9-11 comprise a pad, and the size is 12 bytes. If the field order were different:

```
Ex2 = RECORD
        card : CARDINAL;
        ch : CHAR;
        int : INTEGER;
      END;
```

then 'card' would still occupy bytes 0-3, 'ch' byte 4, bytes 5-7 would be a pad, and 'int' would occupy bytes 8-11, for the same size of 12

(2) Since the offsets have been calculated and pads inserted on the assumption that the record starts at address 0, its actual address must be such that the alignments guaranteed by this assumed start address are still satisfied – this will be the case if the alignment requirement of the entire record is the strictest alignment of any field. Thus, both Ex1 and Ex2 have 4-byte aligned *INTEGER* and *CARDINAL* fields; any variable of either type must be aligned at a 4-byte boundary to guarantee 4-byte alignment of the *CARDINAL* and *INTEGER* fields

The rules clearly apply recursively: if a field of a record is itself a structure, its size and alignment are calculated and then used in calculating its offset within, and the size and alignment of, the parent record.

3 Thanks to the requirement mentioned in note **2** that records finish on a multiple of their alignment, arrays are quite simple. An array of any type never needs any further padding between elements, and so the array size is just the element size multiplied by the number of elements. For arrays with any number of dimensions, just multiply the ultimate element size by the product of the index type cardinalities. The array alignment requirement is the same as that of its elements.

Note that some special cases cause minor variations :

(1) Open array parameters acquire an extra following *CARDINAL* parameter which holds the *HIGH* value.

(2) Empty records parts have a dummy field inserted, since the C compiler treats an empty struct or union as an error. The inserted field is named `char _99_ /*dummy */` Empty variant parts (that is, all variants empty) are simply omitted. The usefulness of empty records or variant parts is, of course, very limited !

These rules apply to all versions, except that the alignment of the underlying primitive types varies from machine to machine. In particular, current versions based on the Motorola 68K processor never require more than *even*-byte alignment. On the pc, **gpm-pc** also uses even byte alignment for primitive types larger than one byte. The *SPARC* architecture only requires quad-byte alignment for reals, in current models. The *MIPS* processor versions use the same alignment as does the the HP precision architecture, with octo-byte alignment for reals.

The important attribute is that **gpm** should use the same alignment rules as the usual C compiler on the same machine, so that the interface to foreign libraries is safe.

9.3.1 Subranges

Subranges only take up as much space as is required to contain their extremal values. They are compatible with their *host type* (that is, the type of which they are a subrange), and are automatically widened and narrowed as necessary. In particular, subranges which fit within the bounds [-32768 .. 32767] will occupy two bytes only, and will be the same as C-language `short int`. Similarly, unsigned ranges which fit within the limits [0 .. 65535] will occupy only two bytes, and will be the same as C's `short unsigned`.

In the case of even shorter subranges, only one byte will be occupied. Thus a subrange [-128 .. 127] will be the same as C's (signed) `char`, while a subrange [0 .. 255] will be the same as C's `unsigned char`.

9.3.2 Miscellaneous notes

Variable declarations in a given scope are usually emitted in a different order in the C-intermediate code than the order in the original Modula declarations.

Modula identifiers which clash with C reserved words are systematically renamed – e.g. `VAR char : CHAR` leads to `char _char`.

9.4 How gpm passes parameters and results

9.4.1 Parameter passing

The parameter passing conventions of **gpm** are designed so that, whenever possible, they conform to the same rules as for language C. In particular, scalars (numeric and ordinal types) and records are passed by value or by reference according to whether they are value of variable mode. Arrays are always *passed* by reference. In the case of value arrays the called procedure is responsible for making the local copy.

Open arrays are always passed by reference, and have a second parameter automatically included which contains the *HIGH* value.[3] Procedures with value mode open arrays make a local copy of the actual parameter, obtaining the necessary buffer space from the *stack for open array parameters* (*SOAP*) space.

Formal parameter	Modula type	type of C parameter(s)	Comment
value scalars	p : S	S p;	S is a scalar type
VAR scalars	VAR p : S	S *p;	S is a scalar type
value records	p : R	R p;	R is a record type
VAR records	VAR p : R	R *p;	R is a record type
value arrays	p : A	E *p;	E is elem-type of A callee makes copy
VAR arrays	VAR p : A	E *p;	E is elem-type of A
open value arrays	p : ARRAY OF E	E *p; unsigned h;	callee makes copy h is HIGH value
open VAR arrays	VAR p : ARRAY OF E	E *p; unsigned h;	h is HIGH value
sets			*see the note*

Note

Sets which are one word in size are treated as scalars. Multi-word sets are treated exactly as for (fixed size) arrays.

9.4.2 Function results

Functions returning scalar types and one word sets pass their results in a register in all current **gpm** implementations. Functions returning records do so in the same way as C does on the same machine.

Currently versions of **gpm** do not allow for functions to return arrays or multi-word sets. Nevertheless the result passsing mechanisms have been decided and prototyped. This information is given here for future reference, and because future releases will use an uniform mechanism for arrays and for records.

Future releases of **gpm** will return structured types in the following way, irrespective of the mechanism used by language C for records on the same machine.

- a synthetic first parameter will be prepended to the parameter list which will point to the destination location
- the called function will copy the result to the designated location

[3] Note however the deliberate exception to this rule in the case of *INTERFACE* definition modules whose sole purpose is to interface with C libraries using *exactly* the C-language conventions. This exception is treated in detail in the chapter *Interfacing to other languages*.

- the function will return a pointer to the destination location exactly as if it were declared as returning a pointer to the actual result type

Thus, a function procedure with the following declaration

```
PROCEDURE ArrayValue(params...) : ArrayType;
BEGIN
  ...
  RETURN a;
END ArrayValue;
```

will be treated internally as if it were declared in this way —

```
TYPE ArrPtr = POINTER TO ArrayType;

PROCEDURE ArrayValue(dst : ArrPtr;
                        params...) : ArrPtr;
BEGIN
  ...
  dst^ := a;   (* copy result to destination *)
  RETURN dst;  (* return pointer to result   *)
END ArrayValue;
```

9.5 How gpm forms linker names

gpm produces linker names which are at most 31 characters long, and which depend on both the identifier and module name of the named object. This is necessary since Modula understands qualified names as a mechanism for resolving clashes of names, but the standard *UNIX* linker **ld** only understands a single level of names.

The linker name is formed by concatenating two strings formed from the module identifier, truncated if necessary to 10 characters, and the object name, truncated if necessary to 20 characters. A *lowline* (underscore character) is placed between the two parts for ease of readability. **gpm** preserves the case of characters in identifiers. If follows that linker names will reach the maximum length of 31 characters only if the module identifier is at least ten characters long, **and** the object identifier is at least 20 characters long.

As an example, if a module *Terminal* exports an object with identifier *GetKeyStroke* an importing module may refer to the object as *GetKeyStroke* or as *Terminal.GetKeyStroke*, depending on whether qualified or unqualified import is used. In either case, the object is known to the linker as `Terminal_GetKeyStroke`.

A special case is the synthetic names which are formed for the initialization entry point for module bodies. The entry point of the main module is `Start`*module name*. The entry point for the implementation part bodies of imported modules are called `Init`*module name*.

gpm can detect if two objects known to the compiler during the same compilation accidentally generate the same linker name. However, it is possible that such conflicts will only be detected during the build phase if the two objects are not both visible in any single module of the program. Such linker name conflicts do not arise if sensible naming policies are adopted, but it is possible to deliberately choose names so as to demonstrate the error message.

Chapter 10

Using the gpmake Tool

10.1 Overview of gpmake

Almost all substantial programs involve multiple modules which are spread over a number of separate files. Working out the dependencies between them can be time consuming, particularly if changes are made to lower level definition module files. The *IMPORT* statement of Modula-2 enables a utility to calculate the dependencies automatically, thus saving time, and reducing the chance of errors.

Separate, but not independent compilation in Modula-2 through the *IMPORT* statement makes explicit the files from which other program objects are required, within the initial file.

The related programs

The make utility comes in 2 flavours, **gpmake** and **gpscript**. The version **gpmake** provides automatic recompilation of all files whose modification times or module keys indicate that they are not up to date. The version **gpscript** provides an executable script file. If this script is executed it causes the compilation of *all* the non-library modules of the program.

Those interested in a more technical overview of the program will find details of the algorithm in the section *Smart recompilation*, later in this chapter.

gpmake

All that is required is the command **gpmake** *name*, where *name* is the name of a main (program) module, probably in a file named *file*.**mod**, and the compiler will be automatically invoked with commands for recompilation of all non-library files which are missing or in an inconsistent state. Following successful recompilation of the inconsistent files, the **build** program is automatically invoked to create the executable file.

gpscript

All that is required is the command **gpscript** *name*, where *name* is the name of a main (program) module, probably in a file named *file*.**mod**, and a script file will be generated with all commands for recompilation of all non-library files in the correct order. The script file is created using the same name conventions as the compiler, with the name *name*.**mak**. This file is an executable script file in the *UNIX* environment. The script file may be executed by means of the command `sh` *filename*`.mak`.

In **gpm-pc** the make file is named with extension `.bat` to fit in with *DOS* conventions. If there is already an executable file with the same base name, a warning is issued, since versions of *MS-DOS* prior to 5.0 cannot execute a `.bat` file if there is a `.exe` file on the path with the same base name. In *DOS* version 5 it is possible to simply issue the command *filename*`.bat`, in earlier versions the executable file must be deleted.

10.1.1 Invoking the program

The program **gpscript** will be used in most of the following examples, but the comments will apply equally in most cases to **gpmake**. The syntax for use is **gpscript** [*options*] *name* The given name must correspond to the name of the base source file in one of the following ways —

- the given name includes a "." character, and the name exactly matches the filename
- the source file has a name formed by moving the given name to lower case, and adding the suffix `.mod`
- the source file has a name formed by moving the given name to lower case, shortening the base name to eight characters
- the source file has a name formed by moving the given name to lower case, shortening the base name to eight characters, and adding the suffix `.mod`

Suppose, for example that the source of module *Graphbuild* has been saved in file *graphbui.mod*. The commands *gpscript Graphbuild*, or *gpscript graphbuild*, or *gpscript graphbui.mod* would all be effective in finding this file.

Options, if any are specified, are passed unchanged to the compiler.

Example

To use the well known dhrystone benchmark program as an example, suppose we wish to create a script file containing all the commands to "make" an executable file for the program. Before showing an example, however, it may be instructive to look at the import structure of the program. The modules involved are

- Dhry (main)
- Dhry1 (definition and implementation)
- Dhry2 (definition and implementation)
- Dhry3 (definition and implementation)

The relevant part of the code for each is :

```
MODULE Dhry;
FROM Terminal IMPORT Write, WriteString, WriteLn;
FROM Dhry2 IMPORT Proc0;
...

DEFINITION MODULE Dhry2;
FROM Dhry1 IMPORT RecordPointer;
...

DEFINITION MODULE Dhry1;
(* no imports *)
...

DEFINITION MODULE Dhry3;
FROM Dhry1 IMPORT ArrayDim1Int, . . . etc;
...
```

This structure dictates the order in which the definition modules must be compiled. Subsequently, the main module and all implementation modules may be compiled in any order.

If we now issue the command `gpscript -irOf dhry` the file **dhry.mak** is produced with the following contents

```
# script for the compilation of module [Dhry]
gpm -irOf dhry1.def
gpm -irOf dhry3.def
gpm -irOf dhry2.def
gpm -irOf dhry.mod
gpm -irOf dhry2.mod
gpm -irOf dhry1.mod
gpm -irOf dhry3.mod
build dhry
```

Note that the option **–irOf** has been passed to *every* call of **gpm** in the script file. This is a general rule.

It can be seen that **dhry1.def** must be compiled first, because both **dhry2.def** and **dhry3.def** import from it.

Should a command be issued without any arguments, a usage message is given.

10.1.2 Search Strategy

Starting with the base file, **gpscript** tries to find a definition module and an implementation module for each imported module. It then tries to do the same for each of the imported modules until all modules required by the program are located. Search is made for module source files only, using the name conventions of the compiler.

If the imported module is *SYSTEM*, no files are required because the objects of *SYSTEM* are known to the compiler.

The definition module file is first searched for in the current directory. If this file is not found, a search is made for the symbol file in the current directory, and on the path *$M2SYM*. It is an error if the symbol file cannot be found.

Whether the definition module is found or not, a search is made for the implementation module file in the current directory. It is common to have a customized local implementation of a module the interface of which is defined in a library in another directory. However, it makes no sense to have a local definition of a module which is implemented in another directory, and the programs reject this as an error.

If a definition module is found in a library area, then it is presumed that the implementation is a custom version of a system library module.

Whenever a source file is found in the current directory, the file is opened and the source is parsed. First, it is verified that the module name is actually correct, as specified in the importing module. Then the module import list is parsed to see which other modules need to be processed.

In the case that a definition is found to be a *FOREIGN* or *INTERFACE* definition then these tools are unable to perform any checks on the correctness or otherwise of the implementation. A warning is issued that manual recompilation of the implementation of any such files may be necessary.

An often encountered case occurs when an implementation file is not found in the current directory, and neither is the definition, but then subsequently a definition is found on the path *$M2SYM*. This is the usual case for a module supplied in the system library, and no recompilation should be necessary which involves these modules. It should be noted here that care will be needed by persons engaged in development of library modules, and it is advised to have the environment carefully crafted, perhaps to have all library files available locally while working, and the environment variables set accordingly.

10.2 Smart recompilation

gpmake offers "smart recompilation" of all of the files in a program which are inconsistent with the latest versions of the available source code files. This utility differs from, and improves on, traditional "make" programs in two ways —

- the module dependencies are automatically extracted from the source code, and thus do not depend on the accuracy of programmer supplied dependency lists
- if some module is recompiled, dependent modules will be recompiled only if it is found that the cryptographic key of the recompiled module is changed

Conditions for recompilation

It is possible to use **gpmake** without being concerned about the algorithm by which it performs its analysis. Nevertheless, an understanding of the behaviour of the program is helpful if a user wishes to predict the consequences of a particular change to a module.

The program begins by constructing the importation graph of the nominated program. This is not necessarily a complete graph, since the program does not explore any dependencies in the libraries. Any modules not in the current directory are thus *leaves* of the graph fragment.

The next step is to topologically sort the nodes of the graph. This operation sequences the decisions on recompilation in such a way that every module has the decision made only after the decision has been made for all modules on which it depends. A definition module file is compiled if and only if one or more of the following conditions is true —

- the symbol file *file*.`syx` is missing
- the symbol file is present but the creation date is earlier than that of the corresponding source file, but not more recent than the time at which **gpmake** was invoked[1]
- the symbol file is present but has key values which are inconsistent with the keys of its imported symbol files

For program or implementation modules the corresponding conditions are —

- the reference file *file*.`rfx` is missing
- the object file *file*.`o` is missing
- the reference file is present but has key values which are inconsistent with the keys of its imported symbol files
- the reference file is present but the creation date is earlier than that of the corresponding source file, but not more recent than the time at which **gpmake** was invoked[1]
- all files are present, but the time stamps on the reference and object files differ by more than thirty seconds, but the reference file is not more recent than the time at which **gpmake** was invoked[1]

The domino stopper effect

The implementation of the strategy used here has an immediately useful effect. Suppose, in the *dhrystone* example, that the file `dhry1.syx` was deleted from the file system, or that the file `dhry1.def` was editted in some trivial way which did not change the meaning of the definition. When **gpmake** is executed, the definition file must be recompiled.

```
$ gpmake -irOf dhry.mod
## compiling dhry1.def
## building dhry
Circular imports, initialization order is
   <Dhry3> (empty body)
   <Dhry2> (empty body)
$
```

Note that even although every single module in the program depends directly or indirectly on the recompiled module, only `dhry1.def` is recompiled. Although the first domino has fallen, the others do not follow. This effect arises because the new version of `dhry1.syx` turns out

[1] This extra condition ensures that files are only recompiled once. Without this test, repeated recompilation could be caused by, for example, a source file with its date erroneously set to some future time, or an exceptionally slow recompilation.

to have exactly the key value which the consistent modules are expecting. No other modules meet the conditions for recompilation, and no unneccesary work is performed.

Suppose, contrary to the above assumption, that the interface in the file `dhry1.def` had actually been modified in some non-trivial way. It that case the symbol file of the recompiled module would have a different key, and all the other modules would be recompiled. In terms of the conditions given above, the third condition would be met, in each case.

10.2.1 Summary of messages

The programs produce the following messages —

`#gpmake: Usage: gpmake [-adfgIilOcOfprtvV] BaseModFileName`

This is the normal usage message which indicates the correct command to invoke the program.

`#gpmake : Cannot complete unless .obj produced, so -S, -n are illegal`

The recompilation cannot complete unless an object file is produced. So it is illegal to specify an option which asks for an object file to *not* be produced. This prevents the compile-and-test cycle from looping.

`#gpmake : bad exit code from` *process-name*

One of the subprocesses of **gpmake** sent back an error return. This usually happens if a compilation results in an error exit.

`#gpmake : can't exec program` *process-name*

This results from a failure to spawn a subprocess, and is usually caused by executable files such as **gpm** not being on the path or not being executable by the user.

`#gpmake : can't open temporary file: /tmp/gpm`*PID*

The temporary file could not be opened, check for locked files in the /tmp directory.

`#gpmake : can't find` *process-name*

A subprocess of **gpmake** could not be found on the executable path.

`## compiling` *file-name* `...`

This is the normal trace message of **gpmake** it is not an error message.

`## building` *file-name* `...`

This is the normal trace message of **gpmake** it is not an error message.

`#gpmake : invalid source file format`

The current file had invalid syntax, and maybe is not a source file.

`#gpmake : not a base module`

The current input file is not a main module. Remember that implementation modules cannot be the base of a make operation.

#gpmake: Foreign implementation of <*ModuleName*> may need recompilation.

A foreign definition module was found. **gpmake** cannot check on the consistency of the object file for this module, since it cannot deduce the name of the source file.

#gpmake: searching for <*Module1*>, found <*Module2*> in file *filename*

The actual module name found in the file was not the same as the name used in the importing module. This is usually a spelling error. Check for upper-case – lower-case problems.

`#gpmake : invalid symbol file format`
The current symbol file format has invalid syntax. This is a serious error which can only arise due to corruption of the symbol file.

`#gpmake : local .def file must have a local .mod`
If a definition module is found in the current directory, then the corresponding implementation must be found also. The reverse situation, a local implementation with the definition on the library path, is normal and passes without comment.

`#gpscript: WARNING:- delete` *filename*`.exe before executing` *filename*`.bat`
This message appears in DOS versions only. It warns that the file *filename*`.exe` must be deleted before the batch file *filename*`.bat` may be executed.

`#gpmake: "DEFINITION" or "IMPLEMENTATION" not found`
gpmake assumes minimally-correct syntax in the parts of the source files it must scan. This error message occurs if the expected module keyword is not found.

`**** m2rts: assert error: Attempt to read past <EOF> ****`
Another response to syntax errors which prevent **gpmake** from extracting dependency information from the source files. This error message occurs if the scanner encounters end-of-file before the required parts are found - typically due to an unclosed comment.

10.2.2 The rule for forming file names

File names are taken from the module identifier, truncated if necessary to 80 characters and moved to all lower case. The extensions `.def, .mod, .syx, .rfx`, or `.o` are appended, as appropriate. In the case of source files, with extensions `.def` or `.mod`, if the file is not found the module base name is truncated to eight characters and the usual extension added. This gives compatability with files which have been transferred from *DOS*.

10.2.3 Files

UNIX files

The executable files used by **gpmake** are — `gpmake, decider, graphbuild, gpm, gpm2, build, build2`. All of these must be on the path. *graphbuild* and *decider* are subprocesses of **gpmake**. The first creates the importation graph and writes it to a file in the current directory. The second reads the file, and decides incrementally whether each file requires compilation.

Two temporary files are created. These are `/tmp/gpm`*PID* and `./`*filename*`.mak`, both are deleted automatically on completion.

gpscript creates a text file named *filename*`.mak` in the current directory.

DOS files

The executable files used by **gpmake** are — `gpmake.exe, decider.exe, graphbuild.exe, gpm.exe, gpm2.exe, build.exe, build2.exe`. All of these must be on the path. *graph-*

build and *decider* are subprocesses of **gpmake**. The first creates the importation graph and writes it to a file in the current directory. The second reads the file, and decides incrementally whether each file requires compilation.

Two temporary files are created. These are `modbase` and *filename*`.mak`, both are deleted automatically on completion.

gpscript creates a text file named *filename*`.bat` in the current directory.

Chapter 11

The Cross-reference utility gpxrf

gpxrf is a simple utility for obtaining cross reference listings of Modula programs. The program has two command line options, and always sends its output to the standard output stream.

Using gpxrf

The program is invoked from the command line with the following syntax

gpxrf [−*options*] **filename**

Because the output always goes to the standard output, it is common to use **gpxrf** in combination with the *UNIX* shell commands for redirection or piping to other tools.

The default output lists all of the non-pervasive identifiers used in the nominated file, sorted into lexicographic order, each followed by a list of those line-numbers on which it occurs. The ordering is case sensitive, placing *Foo* before *foo*, and *bar* before *BAT*.

Consider the following program

```
MODULE Hello;
FROM Terminal IMPORT WriteString, WriteLn;
  VAR str : ARRAY [0 .. 6] OF CHAR;
BEGIN
  str := "hello, "; WriteString(str);
  str := "world";   WriteString(str);
  WriteLn;
END Hello.
```

The command `gpxrf hello.mod` produces the following output

```
GPM Cross reference listing for file <hello.mod>

Hello                        1    8
str                          3    5    6
Terminal                     2
WriteLn                      2    7
WriteString                  2    5    6
```

```
String usage is 513 bytes
Entries = 78
```

Identifiers are padded to a uniform length of 25 characters, or truncated if necessary. Note that the identifier *str* occurs twice on line 5 and twice on line 6, but these numbers occur only once in the output. If the number of occurrences of an identifier is large, **gpxrf** wraps lines so the block of line numbers will fit on an 80 column screen.

gpxrf also indicates how much of **gpm**'s string-table memory the file used up. In this example only 513 bytes were used, mainly by pervasive identifiers which were not used in the example program.

Command line options

The option **−p** causes **gpxrf** to include **pervasive** identifiers in its output. The main body of the output for the previous example program, using this option is

```
CHAR                     3
Hello                    1    8
str                      3    5    6
Terminal                 2
WriteLn                  2    7
WriteString              2    5    6
```

The option **−f** lists the identifiers in **frequency** of occurrence in the nominated file. If several identifiers occur the same number of times, the group is sorted on the lexicographic order. For the same example program, the main body of the output is

```
Terminal                 2
Hello                    1    8
WriteLn                  2    7
WriteString              2    5    6
str                      3    5    6
```

Note in this case that *str* is listed last on the basis of its five occurrences, even although it only occurs on three different lines.

Chapter 12

Errors and Error Messages

12.1 Errors Detected at Build Time

The *build* program, as its name implies, builds a complete program out of the separately compiled parts in the library and the user modules. In doing so it performs a number of checks for consistency, and consequently detects certain errors.

Every symbol file contains a magic number called the *key*. If two versions of a symbol file have different keys then it is certain that they contain different information.

When an implementation module is compiled, the reference file contains the key values of every symbol file which that module imports. This allows the build program to rigorously check that if two modules both import the same module, then they both access the same version.

If the build program detects inconsistent keys, it issues an error message, and the final message

```
**** File Creation Unsuccessful ****
```

A detailed trace of the build process will show the key values for every module in every reference file, and will help pinpoint the module(s) causing the problem.

Build also produces a warning message if modules are involved in circular imports. If your program causes such a message to be displayed, you will have to check that no module on the circular path is relying on initialized data of any later member of the cycle.

```
$ build dhry
Circular imports, initialization order is
   <Dhry3> (empty body)
   <Dhry2> (empty body)
$
```

In this example, the *dhrystone* program has a circularity between two of its four modules. In this case the modules involved in the circularity *both* have no initialization (empty body) and so the message may be ignored without further consideration.

In general circular imports should be avoided if at all possible, in order to remove the need for manual checking of data structure dependency.

A special case is that of circular imports which involve a module which has been declared with the *!LIBRARY* pragma. In this case the circularity is fatal, since the compiler relies on *!LIBRARY* modules not being involved in cross-module recursion.

If the base module filename passed to **build** is not a program module (for example if it is an implementation module) **build** issues an error message stating this fact. Similarly, if any module imported by the base module is not an implememtation module, an appropriate error message is issued. This last diagnostic can only occur if a separately compiled module has a valid definition part, but the keyword *IMPLEMENTATION* is missing from the implementation part. The same diagnostic occurs if the input base file name is inadvertently typed with an extension.

12.1.1 Summary of build messages

The **build** program emits several other messages. These are, in alphabetical order:

** Bad filename <*file*> in header file **
This error is **fatal**. The library filename pragma in a foreign header file was badly formed. The first character which is not a legal filename character must be '>'.

`** Bad reference file syntax **`
The current reference file has incorrect syntax. This implies that the file has been corrupted in some way. This error is not only **fatal**, but causes *build* to immediately abort execution.

`** Base file is not a program module **`
This error is **fatal**. The filename given on the command line does not correspond to a reference file belonging to a program module. Remember that the base name must not be the name of an implementation/definition part pair. This message also arises if you type in `modname.mod` instead of `modname` without a filename extension.

** Module <*module*> was compiled with the -p (profiling) flag **
This error is **fatal**. Recompile the named module without the flag.

`** Base file not found **`
This error is **fatal**. The command line file name could not be found. It must be in the current directory. Other files may be anywhere on the library path.

`Build: illegal option`
This is a **warning** only. An illegal command line argument has been passed to **build**.

`Build: too many libraries`
This error is **fatal**. **gpm** currently allows a maximum of 16 library object files to be included via the header file mechanism. This is not a limitation on the number of header files, only on the number of different libraries which may be utilized.

`** Can't create output file **`
This error is **fatal**. The attempt to create the intermediate code file *name.c* failed. Check file permissions. The error usually arises when the –S flag has been used previously, and has left a protected file `modname.c` in the current directory. The error is very unlikely to occur in other cases, as the synthetic name for the output file is the unique /tmp/bld*pid*.c, where *pid* is the process identifier. All such files are removed at the end of the building process.

`** Can't create shell file **`

This error is **fatal**. The attempt to create the linker script file *name* failed. Check file permissions. The error usually arises when the –S flag has been used, and a protected file `modname` already exists.

`** Can't find runtime sytem file **`
This error is **fatal**. The file `m2rts.o` could not be found on the library path $M2LIB.

`** Can't find file <`*file*`> on library path **`
This error is **fatal**. A file named in a header file could not be found either in the current directory or the library path $M2LIB.

`** FATAL CIRCULAR IMPORT ERROR **`
This error is **fatal**. A circular import exists involving a module which has been declared with the !*LIBRARY* pragma. Such circularity does not occur in the supplied libraries.

`** Imported module <`*module*`> is not an implementation **`
This error is **fatal**. The module which was imported has a symbol file, but the reference file belongs to a program module, rather than an implementation. Check whether you have left off the word *IMPLEMENTATION*.

`** Inconsistent key for module <`*module1*`> in reffile of <`*module2*`> **`
This error is **fatal**. While reading the reference file of *module2*, a key value was found for *module1* which is different to the value found in a previous reference file. This implies that two modules of the program have been compiled using two different versions of *module1*.

12.2 Errors Detected at Compile Time

The compiler detects about two hundred different errors during compilation. These are divided for convenience into several categories. Detailed explanations for these errors are given in the following chapter.

12.2.1 Lexical Errors

These are errors caused by badly formed tokens in the input file, such as illegal characters, numbers, badly formed literal strings and comments. These errors have error numbers less than 100.

12.2.2 Syntax Errors

These errors are caused by incorrect syntax in the input file. Such things as missing semicolons, keywords or arithmetic operators fall into this category. In most cases the error message indicates the identity of the expected symbol which was not found. These errors have error numbers between 100 and 199.

12.2.3 Semantic Errors

These errors are caused by the failure of various so-called *static semantic checks* in the compiler. These checks include such things as compatibility of types, correct declaration of objects, matching of identifiers and so on. This group is easily the largest group, being allocated over one hundred separate error messages numbered between 200 and 450.

12.2.4 Warnings

The compiler produces a number of friendly warning messages! These occur when certain non-fatal errors are discovered. For example, if a program has a procedure which is not called, exported or assigned as a procedure variable, then **gpm** will warn you of this probable error. Similarly, *LOOP* statements without at least one *EXIT* or *RETURN* attract a warning.

In order to provide maximum compatibility with previous definitions of Modula, **gpm** accepts several syntactic constructs which are technically illegal but conform to the definitions in *Programming in Modula-2.* These constructs are flagged with an obsolete syntax warning message.

Warnings have "error" numbers greater than 450. Their reporting may be suppressed by use of the **-d** (dangerous) compiler flag.

12.2.5 When are Errors Detected?

The compiler produces its error messages in two separate phases. The first phase is a single pass over the source text; the second is a traversal of the internal abstract syntax representation. If errors are found during the first phase, the further checking of the second phase is not attempted. In even more extreme circumstances, if symbol files are missing, or relate to the wrong module, even the first phase is cut short.

It is useful to know which errors are detected in which phase, since it explains why previously unreported errors may appear after an unrelated error is removed. The rules are as follows

- all lexical and syntactic errors are detected in the first phase, during parsing of the source file
- semantic errors in declarations are detected in the first phase, since declaration analysis is interleaved with parsing
- semantic checks on FOR loop headers are performed in the first phase, for no particular reason
- all other semantic checks on statement sequences are produced in the second phase, the semantic analysis traversal
- global semantic errors and warnings, such as that caused by the failure to elaborate an opaque type, are produced after the completion of the second phase.

More detailed explanations of the various errors are included in the next chapter.

12.2.6 Position of the Error Marker

Most errors are reported with a message which "points" to the error

```
                    thing := 5;
**** ...... ^ semantic error 204 ****
**** 204 Identifier not known in this scope ****
```

In general, the marker points to the start of the token which is in error, as in the example. However, there are examples where the marker points to the following token:

```
 CONST Foo = 1.0 / 0.0;
**** ............... ^ semantic error 215 ****
**** 215 Range of type exceeded ****
```

In some cases the error marker may even point to the next line. This is common for missing punctuation errors, since **gpm** cannot tell a punctuation symbol is missing until it actually finds something other than whitespace (blanks, newlines and comments).

There are also a small number of errors where the error has no position since the error is caused by the fact that some expected feature is missing. Failure to elaborate an opaque type is in this category. Such an error gives the following style of message, at the last legal position that the declaration could have been placed.

```
BEGIN (* module body *)
**** Error 224 with identifier <Foo> ****
**** 224 Opaque type not elaborated ****
```

In this case type *Foo* was declared in a definition part but was not found among the declarations of the implementation.

This style of error message is crucial also when declaration conflicts arise between the constant identifiers of imported enumerations, since the offending identifiers do not occur in the import lists themselves.

12.2.7 Other compiler messages

gpm produces a small number of error messages which are related to the compiling environment, rather than to the source file. These are as follows, in alphabetical order:

`gpm2: Assert error in module <M2`*xxxx*`> at line` *nnnn*
This error should never occur. It indicates that the compiler has detected an internal error. All such occurrences should be reported to your support organization.

`Bad option -`*k*
This is a **warning** only. The option *k* was not recognized. Check this manual for a list of allowed options.

`gpm2: Can't open input file`
This error is **fatal**. The input file specified on the command line could not be found in the current directory. Remember that the source file *must* be in the current directory, and the full name of the file must be given.

```
gpm2: Can't create symbol file
gpm2: Can't create reference file
gpm2: Can't create object file
gpm2: Can't create tmp file
```

These errors are all **fatal**. Usually these are caused by an existing file which is write protected.

`gpm2: Can't open list file`
This is a **warning** only. Compilation proceeds, but the requested listing is not made.

`gpm2: Can't open error list file`

This is a **warning** only. **gpm** cannot find or cannot open the file `m2errlst.dat`, which is required for verbose error messages or for the interactive option. Compilation proceeds, but without the verbose version of the error diagnostics.

```
gpm2: String table overflow
gpm2: Hash table near full
```

These errors are **fatal**, they imply that an internal compiler table limit has been exceeded. The default limits of the compiler are sufficiently large that they should rarely be exceeded for programs partitioned into moderately sized modules as suggested by good software engineering practice. If circumstances justify larger name spaces, the table sizes can be increased by the environment variables *M2HASH* and *M2STRING* as described in section 7.2. An alternative is to partition the module to decrease the number of identifiers or the number of strings.

`Expected` *n* `found` *m*
`Bad` *object* `in SYM`

The current symbol file has incorrect syntax. These imply that the file has been corrupted in some way. These errors are not only **fatal**, but causes *gpm* to immediately abort execution.

Since the compiler is implemented in Modula, an error in the compiler itself may lead to any of the runtime errors described in the next section. The only expected such error is a storage error in the PC version, where memory exhaustion is likely for large compilation units (depending on other resident code). As noted above, subdivision of modules is the cure.

12.3 Errors Detected at Runtime

Runtime errors are errors which are detected when a program is executed. Some errors are detected by the compiler during compilation while others can only be detected when the program is executed.

There is a module **Exceptions** which allows the program to regain control after the occurrence of a runtime error. In the absence of the use of this module runtime errors cause the program to terminate with the production of a core dump.

The core dump exists as a file named **core** in the current directory. An analysis of this file can provide information on the nature of the event. This process is called *Postmortem debugging*. Depending of the way in which the program was compiled, more or less information may be obtained from the core-dump file.

The higher levels of information which can be obtained require the program to be compiled with special option flags (see the *Options* chapter for a list of option flags). However, useful information can be obtained without invoking the special options. In particular a trace of the procedure-call chain is most useful, because it points to the action which led to the event and tells the user in which procedure the error occurred; this is called *unwinding the stack*. The names of the procedures on this stack are just the (Modula) procedure names. In the case of exported procedures the name is formed from the first 10 characters of the module name followed by the first 20 characters of the procedure name.

For further information on this point refer to section **How gpm forms linker names** in the **Implementation Specifics** chapter.

On the HP9000 the debuggers are called “xdb” and “adb”. On **mips**-based machines the usual debugger is “dbx”, and is used for postmortem and for interactive runtime debugging.

One of the appendices for this manual describes how to use the standard debugger supplied with your system to debug programs compiled using **gpm**.

Every program compiled by **gardens point modula** is linked to a module of useful routines called the runtime system. The functions and procedures of this module, which is called **m2rts**, performs such things as error checking and the catching of errors. This module writes out error messages which indicate the nature of any error, and then causes a core dump to be produced.

12.3.1 Range Check Errors

Range errors occur whenever an attempt is made to assign an illegal value to a variable of ordinal type. Ordinal types include characters, enumerations and the whole number types, and subranges of these. The assignment might be an assignment statement, an actual parameter substitution, or an *INC* or *DEC* procedure call.

Different format messages are produced for various kinds of tests — examples are

```
**** m2rts: range error: 25 > 12 ****
```

an attempt was made to assign the value 25 to a variable the upper bound of which was 12.

```
**** m2rts: range error: 25 not in [3..17] ****
```

an attempt was made to assign the value 25 to a variable for which the legal range was [3..17].

```
**** m2rts: range error: -3 < 0 ****
```

an attempt was made to assign the value -3 to a variable with lower bound 0 (probably an unsigned type).

```
**** m2rts: MOD by op < 0 ****
```

Modula semantics demand that *MOD* (unlike *DIV* and the newly proposed whole number operators *REM* and '/') is only defined for positive right hand operands. Values less than zero report this error, values of exactly zero give the separate divide-by-zero message.

12.3.2 Index Bounds Check Errors

Whenever an array index is calculated, it is compared with the upper bound of array indices. Internally, **gpm** uses zero-based indices. If, for example, you declare an array with an index type which is [-5 .. 5], the array will have eleven elements numbered (internally) as 0 to 10. The index bound is thus 10 in this case, and all index expressions will be compared against the limit 10. If an index exceeds the bound, a message is produced with the following format

```
**** m2rts: index error: 11 >10 ****
```

In this case an attempt was made to select element 11 of an array with last element 10, corresponding to an original, unnormalized index value (in this example) of 6.

Some more recent versions of **gpm** give the error message in terms of the user-defined index bounds. In this case the error is quite unambiguous, and for the previous example would read

```
**** m2rts: index error: 6 not in [-5 .. 5] ****
```

12.3.3 Case Selector Errors

If a case statement does not have an ELSE part, and the case selection expression evaluates to a value not specified, a case selector error occurs. The message has the following format:

```
**** m2rts: case selector error: 5 ****
```

in this example the selector expression had ordinal value 5. Note that if the selector type was an enumeration of, say, type *WeekDays*, this would imply that the value *friday* had been selected.

12.3.4 Memory and Bus Errors

If an attempt is made to access an illegal or non-existent memory location, a memory error or a bus error will occur. The most likely cause of this error is an attempt to dereference the NIL value in programs which use pointers, or the use of an uninitialized pointer variable. However, there are other possibilities, such as a call to an uninitialized procedure variable.

There are two message formats used to report these errors

```
**** m2rts: bus error ****
```

when a non-existent memory location is addressed, and

```
**** m2rts: memory error ****
```

when an illegal memory location is addressed. Note that *NIL* pointer references cause the second message to be emitted, uninitialized pointers may cause either.

12.3.5 Divide by Zero Error

An attempt to divide by zero (either by DIV, MOD, REM or '/') provokes the following message

```
**** m2rts: divide by zero error ****
```

12.3.6 Floating Point Errors

If the evaluation of some real-valued expression results in a *not-a-number symbol* a floating point trap occurs, with the following message:

```
**** m2rts: real number error ****
```

12.3.7 Storage Errors

If the supplied version of the module **Storage** is unable to allocate any further heap space to the program, the following message is produced

```
**** m2rts: storage error has occurred ****
```

12.3.8 Soap Errors

The compiler allocates a small amount of space as a **stack** for value-mode **open array parameters** (hence *soap*). This space is adequate for all normal programs. However, if you call many nested procedures with huge value-mode open arrays you may get the message

```
**** m2rts: out of soap space ****
```

If it is absolutely necessary to increase the soap space, then declaring an environment variable *SOAPSIZE=nnnn* where *nnnn* is a decimal number will cause the amount to be varied at build time. The default size of the soap space is 4096 bytes, and **gpm** will ignore any user attempt to allocate less than this amount.

If the **Exceptions** module has been imported, and errors are caught by the user, then both storage and soap errors raise the pre-declared exception *StorageError*. Only if the error is unhandled, and causes termination, will the distinct error messages be produced.

12.3.9 User Errors

The Exceptions module of the standard library provides facilities for raising and catching various errors. Using these facilities, a program may regain control after an error and perform any appropriate error recovery action.

All of the errors described above may be caught and handled by the exception handlers, but it is also possible for the user to define other exceptions.

When programs define additional exception values, they define text message strings which are associated with those values. It is expected that when programs define additional exception values they intend to handle the traps caused by those exceptions. However, if a user defined exception is raised but is not handled by the user program, the normal procedure of termination and core-dump is followed. In that case the following user defined message is displayed

```
**** m2rts: "user-defined text string" ****
```

A procedure in the current version of the *Exceptions* module allows the text string associated with an exception to be extracted. Thus it is possible to display the message (or write it to a log file) even if the exception is caught and handled.

12.3.10 Assert errors and assertion checking

The insertion of assertions into user code, and the subsequent testing of such assertions at runtime is a most powerful tool of software engineering. If the correctness of some complex piece of code depends on some particular predicate then it is sensible to test the truth value of the Boolean expression. This guards against the case that the predicate is not true, but gives rise to an incorrect result rather than to a runtime error.

It is a common practice in well constructed Modula programs to have a user-defined *Assert* procedure which evaluates its actual Boolean parameter, and aborts the program if the result is *false*. **gpm** goes much further than this by providing a built-in *Assert* procedure in the special, system library *ProgArgs*.

Progargs.Assert evaluates its actual parameter and aborts the program if the result is *false*. However, the compiler produces “inline” code for the test, thus avoiding the overhead of a procedure call, and compiles a trap call which provides specific information on the site of the failed test. A typical error message would be

```
**** m2rts: Assert error in module <Foo> at line 1321 ****
```

Notice the importance of the compiler-produced message. Changes to the source code do not require any change to the procedure calls, since **gpm** automatically calculates the proper line numbers and module names at compile-time.

Since assertion tests are intended to catch errors in program logic, it is not appropriate for an exception handler to attempt to recover from such an error. In **gpm** the exceptions handling does not catch assert errors, so assert errors are *always fatal*, even if one or more exception handlers have been set by the use of *Call*.

The runtime overhead arising from the use of these tests is extremely small, and the potential for error detection very great. It is therefore quite acceptable to leave such tests permanently enabled in most cases. Nevertheless, if it is desired, the compilation of assertion tests may be turned off by use of the **–a** command-line option. When this option is used, the *Assert* procedure call in the source code is treated as an empty statement.

The current versions of **gpm** have an optional second parameter to *Assert*, which allows the user to specify a message, as well as the predicate to be tested. An example would be —

```
Assert(NOT broken,"broken hearted");
```

If the predicate is false, the user-specified message is printed, along with the line number information. This feature allows **gpm** programs to be moved to other implementations in which the *Assert* procedure is user-defined. In such a case, the user message is needed to preserve the reduced functionality which a user-defined procedure can offer.

12.3.11 Function return errors

If a value-returning function procedure should reach the end of its code without executing a *RETURN* statement, the following message is produced

```
**** m2rts: function ended without RETURN ****
```

In this case it is necessary to use the postmortem debugger to find the name of the function in which the trap was activated.

12.3.12 Coroutine return errors

If a coroutine should reach the end of its code without executing a *TRANSFER* call, the following message is produced

```
**** m2rts: coroutine ended without TRANSFER ****
```

12.3.13 Stack overflow errors

Stack overflow checking is seldom enabled except for multi threaded programs which use the *Coroutines* library. In any case, if a stack overflow error is detected, the following message is produced.

```
**** m2rts: stack overflow has occurred ****
```

If the error occurred in a coroutine, the workspace allocated by *ALLOCATE* and passed to *NEWPROCESS* must be increased. In the case of stack overflows in the main program, the value of the environment variable *M2STACK* must be increased, and the program built again (no recompilation is necessary).

Chapter 13

Interpreting Compiler Diagnostics

13.1 Introduction

This chapter gives detailed explanations for the compiler diagnostics which **gpm** emits. In some cases the explanations refer to extremely rare and improbable events (see, for example, the explanation for error 303). We have tried to detail all of the obvious and non-obvious circumstances under which each such error arises.

This chapter forms the basis for the online "more info" which is available by using the `gpm -I` option.

Some observations which apply to many error messages

(1) In many cases the compiler is trying to say "I expected this, and you gave me that". It may well be that the confusion was due to a typing or spelling error, which happened to match something different from what you intended. If the incorrect spelling did not match anything the compiler knows about, a clear "not defined" error results, but an unintentional match with some other valid name may go unnoticed until some property of the intended name is not satisfied by the accidental name.

(2) The terms *qualified identifier* and *designator* are used in several error messages; they refer to various forms of names for objects. A simple identifier such as *fred* may be qualified by a module name (*Jim.fred* meaning identifier *fred* of module *Jim*), giving rise to the term qualified identifier. It may also be combined with selection steps in complex names such as $jim.x[a + b].y$, giving rise to the term *designator* for anything which designates some object. (Note that in the last example, semantic checks must be used to determine whether the first "." is a module qualification or a record field selection.)

(3) Many messages are of the form "... not known in this scope ...". Remember that the Modula scope rules make identifiers in other modules invisible unless explicitly *IMPORT*ed or *EXPORT*ed, and that local identifiers of a procedure (including formal parameters) are invisible outside the procedure.

13.2 Lexical Errors

`1 Line ends inside literal string`

Literal strings are limited to a single line. Commonly, this error is due to omission of the closing quote (which must match the opening quote — either ' or "). If you wish to construct literal strings longer than editor or system line length limits, you must do so by programmed concatenation.

`2 Illegal character in input file`

All characters within the Modula character set are acceptable, and control characters in the range 1C to 37C are ignored. Other characters are invalid. If there is no apparent invalid character, use your text editor's "show non-printable characters" option or some dump utility to check for spurious characters.

`3 Input file ends inside a comment`

This error will occur if a closing comment bracket is omitted. Note that comments nest in Modula, so that a subsequent closing bracket will match only its corresponding opening bracket; note also that an intervening space between the '*' and the ')' characters will destroy a comment bracket. Since end-of-file is indicated by a null character (0C), this error will also occur if a null is introduced within a comment. The error is reported at the beginning of the unclosed comment.

`4 Invalid exponent in REAL constant`

Immediately after the E which introduces a real exponent, there must follow an optional sign, and then an unsigned integer.

`5 Illegal character in numeric constant`

Numeric constants may contain only the 'digits' 0..7 for octal constants (suffix B or C), 0..9 for decimal (no suffix), or 0..9,A..F for hexadecimal (H suffix).

`6 Floating-point error during constant evaluation`

The error value "HUGE" was produced when this constant was evaluated. See the machine reference manual for floating-point limits.

`7 Number too long`

Numeric constants are assembled in a buffer which is currently large enough for the number of significant digits in a double-precision floating point value.

`8 Character constant too large (377B is maximum)`

`9 Illegal use of underscore in identifier`

Underscores are allowed in identifiers only singly and internally — a leading or trailing underscore is not allowed; nor are two or more adjacent underscores.

These rules are relaxed in *interface* definition modules, or in any modules which import such modules. In that case, the use of underscores is entirely free, and error 9 should not arise.

13.3 Syntax Errors

The majority of the syntactic error messages are self-explanatory, and are not further explained here.

```
100 Invalid symbols precede start of module
101 No identifier at end of module
102 No fullstop at end of module
103 Expected END symbol
104 Expected module END symbol
105 Expected semicolon
106 Expected declarations
107 Expected equals sign
108 Expected identifier
109 Expected IMPORT symbol
110 Expected comma
111 Expected ')' symbol
112 Expected '..' symbol
113 Error in qualified identifier
114 Expected parameters
115 Expected ']' symbol
116 Expected OF symbol
117 Expected colon
118 Formal parameter bad
119 Expected '{' symbol
120 Error in expression
121 Expected '(' symbol
122 Expected '}' symbol
123 Expected '|' symbol
124 Expected EXPORT symbol
125 Expected selectors
126 Expected addops
127 Expected mulops
128 Error in statement
129 Expected DO symbol
130 Expected UNTIL symbol
131 Expected ':=' symbol
132 Expected TO symbol
133 Expected THEN symbol
134 Expected start of type
135 Expected start of factor
136 Expected BEGIN
137 Premature exit: too few ENDs in  block
138 Expected END identifier;
139 Resynchronizing here
140 Special import statement syntax is incorrect
```

Resynchronizing

gpm uses the names at the end of procedures and modules to help in recovery from errors which are due to too many or too few *END*s. In the case that the *END identifier* is found too soon, **gpm** abandons parsing the rest of (possibly nested) statement sequences and issues error 137.

If an *END* is found but the expected *identifier* is missing then error 138 is emitted and **gpm** searches for an *END* with the matching identifier. This search will stop at the start of any new declaration, so that a simple omission of the identifier will not cause any information to be skipped. However, if there are too many *END*s **gpm** will find the matching one, and issue "error" 139 to announce that it has found the correct resynchronization point.

13.4 Semantic Errors

`200 Identifier at block end does not match`

Modula requires that the *END* of a module (compilation unit, or nested module) or procedure be followed by the name of the module or procedure. Either you have omitted the matching identifier, or mis-spelt it, or perhaps incorrect pairing of *END*s with structures has misled the compiler? (Errors 137 – 139 should catch most incorrect pairings)

`201 Symbol file missing`

You have tried to import from a definition module, and its symbol file was not found. The import may be explicit, or the implicit import of its own definition module by an implementation module. Is the module name spelt correctly? Has the definition module been compiled? Is it in the current path? Do you have read access to it?

`202 Identifier is not exported from module`

You have tried to *IMPORT* a particular identifier from a definition module; the module's symbol file was found, but that identifier was not exported. Is it spelt correctly (or at least the same way — Modula is case sensitive). Is it defined in that definition module? (Quick check: `grepdef` *identifier* will find the identifier no matter which file it is in, or what directory on the search path the file may be in).

`203 Identifer already known in this scope`

You are trying to define a new object, and the name used already has some other meaning in the current scope. It may have been *IMPORT*ed from some external scope, or declared previously in this scope, or is being *EXPORT*ed from the current scope and clashes with a name in the enclosing scope ("this scope" is then the enclosing scope). If you wish to use a similar name, Modula's case-sensitivity may be used to distinguish them; but beware of confusion later — two variables of the same type, distinguished only by case, could easily be confused; on the other hand, two variables of different type, or a type and a variable, would almost certainly cause a compiler error if accidentally transposed.

`204 Identifier not known in this scope`

You are using an identifier which has no definition visible in this scope. Did you mis-spell, or fail to *IMPORT*? Remember that although global objects are visible by default in nested procedures, nothing is visible across module boundaries unless explicitly exported and/or imported. If you are trying to *IMPORT* into a nested module from a module which is not visible in the enclosing scope, "this scope" means that enclosing scope.

`205 Qualified identifier is not a type name`

In a situation where the syntax requires the name of a type, the identifier you have used is known, but is not a type name. Mis-spelt name matching another identifier? Or did you forget that a type is required here?

`206 Type is not an ordinal type`

An even more restricted version of 205 — the type must be ordinal. That is, it must be a type for which the next "counting" or "successor" value is defined. This allows *CARDINAL, INTEGER, BOOLEAN, CHAR,* a user-defined enumeration, or a subrange of one of those; *REAL,* or any structured type (arrays, records, sets) are not allowed. Ordinal types are required for *CASE* statement selectors, array indices, variant record tag types, *FOR* loop control variables, and arguments of *CHR, ODD, ORD, INC & DEC.*

`207 Expression is not compatible with declared type`

Modula enforces strict agreement between types of expressions and the context in which they are used. This error occurs if a label of a CASE statement branch does not match the selector type, an element in a set constructor does not match the set type, a bound of a subrange does not match the host type or the other bound, a record variant label does not match the tag type, or an array index does not match the index type.

`208 Identifier is not a constant`

The syntax requires a constant here. Did you mis-spell?

`209 Maximum of range is less than minimum`

Where a range *a..b* is allowed, *a* should be the lower bound and *b* the upper; the range is from *a* up to *b*, inclusive.

`210 Implementation limit exceeded for set base type`

gpm like most compilers, puts a limit on the size of sets by restricting the range of the base type; for **gpm** the limit is 256 members, starting at ordinal 0. Note that this means some small sets may cause this error — a *SET OF* [1980..1999] has 2000 possible members, not just 20; also a *SET OF* [-5..5] would breach the lower range limit. If necessary, you can work around these limitations by defining your own *HugeSet* or *NegativeSet* types, implemented as *ARRAY OF BITSET*, and mapping members of your desired set type to members of elements of the array.

`211 Target of forward reference not declared`

Modula generally expects that identifiers be declared before use, but there are exceptions: the target type of a pointer (*Fred* in *POINTER TO Fred*), and procedure names. These names must be subsequently declared within that compilation unit. Because of different internal processing of pointers and procedure names, **gpm** reports the two cases differently. Error 211 reports incomplete pointer definitions, at the point where it becomes clear that no definition can appear. Error 204 occurs for missing procedures.

`212 Type ident not expected here`

In an expression which is required by the syntax to have a constant value, the only type identifier which can occur is that of a set (giving the type of a set constant).

`213 Function HIGH cannot be used in a constant expression`

Since *HIGH* always returns a value which varies with the actual parameter corresponding to an open array formal parameter, it cannot be used in an expression which is required to have a constant value.

`214 Parameter is of wrong type`

This actual parameter does not match the type required by the formal parameter in the procedure declaration. This includes built-in functions such as *ABS, CAP & CHR*.

or

This argument to a built-in procedure or function is not a variable or type designator.

`215 Range of type exceeded`

A value which can be checked at compile-time has been found to be out of range. These include constant arguments to built-in procedures and functions, the results of evaluating constant expressions, set elements, and record variant labels. In some cases, error 207 may be reported instead of error 215 — e.g. a *CASE* statement selector is of *CARDINAL* subrange type, and so a negative branch label is rejected as incompatible (which, of course, implies out of range).

`216 Too many parameters`

You have supplied too many parameters for the procedure called. Check the definition, either in the documentation of a built-in procedure, or in the definition module of an external procedure, or the declaration of a local procedure.

`217 Conversion not implemented`

The constant value to be coerced by the built-in function *VAL* must be an ordinal type. Conversion to real types in constant declarations is not yet implemented.

`218 Not of numeric type`

(No longer used)

`219 Operation invalid on constant`

This operation is not appropriate for the constant operand supplied. Check the (implied) type of the constant, and that of the other operand, for consistency with one another and the operator. There is also the case of a specific value of the operand being inappropriate — division by zero.

`220 Type incompatible operands`

In general, operators combine operands of the same types; exceptions such as the set membership operator *IN* still require compatibility between the element being tested for membership and the base type of the set.

Note that if this error occurs, no check is made for the appropriateness of the operator; on correcting the operand incompatibility, an inappropriate operator will give an error such as 270–275.

`221 Not of Boolean type`

The operand of the Boolean operator *NOT* must be Boolean; so too must the conditions which are used in *IF*, *WHILE* and *REPEAT* statements.

`222 Record field name is not unique`

Within a single record, each field must have a distinct name; this includes the tag field of a variant, and each of the variant fields. Fields of other records (including records which are the types of fields of this record) may of course re-use the same names, since qualification by their variable or field name prevents an ambiguity.

`223 Opaque type only allowed in definition part`

Opaque types are intended to provide an exported declaration which allows limited access to the type while hiding other details; they are thus allowed only in *DEFINITION* modules. Note that this implies a limitation on nested modules — they cannot export opaquely.

`224 Opaque type not elaborated`

An opaque type declared in a *DEFINITION* module must be elaborated — its hidden details completed — in the corresponding *IMPLEMENTATION* module. This message will appear at the end of the implementation part, when it is clear that no complete declaration has been found, and will quote the offending type name. Note that references to variables of the opaque type within the *IMPLEMENTATION* module will not have caused errors, since the elaboration check error on the first pass suppresses further semantic checking.

`225 Exported procedure not declared`

This is a similar instance to error 224, except that the object partially declared in the *DEFINITION* module and not completed in the *IMPLEMENTATION* module was a procedure. You must supply the procedure body in the *IMPLEMENTATION* module.

`226 (Implementation restriction) Too many formals of same type`

In a parameter list of the form "(a,b,c,d:SomeType)", there can only be 15 items in the list of parameters all sharing the same type *SomeType*. You can easily circumvent the limit by splitting the list into two: "(a,b,c,...,o:SomeType; p,q,...,u:SomeType)"

`227 Invalid elaboration of opaque type (must be a pointer)`

An opaque type must turn out to be a pointer; if you want something else, you must define a pointer to it and opaquely export the pointer type.

`228 Invalid elaboration of procedure header`

The definition of a procedure in a *DEFINITION* module and its subsequent elaboration in the *IMPLEMENTATION* module must have the same headings. This error occurs if a parameter does not match in type and mode of passing (*VAR* or value), or if the *DEFINITION* specified a function and the *IMPLEMENTATION* a proper procedure (no result type). In the case of a parameter mismatch, the name of the offending parameter in the *IMPLEMENTATION* module is quoted. (Note that the parameter names used in *DEFINITION* and *IMPLEMENTATION* need not match, only their types and modes.)

Exactly the same rules apply for the elaboration of procedures which have been declared with the *FORWARD* keyword. Remember that forward declarations are not required for

gpm. The keyword is only recogized in order to provide source code compatability with those Modula compilers which have single-pass restrictions.

`229 Function return type not as defined`

The elaboration of a function in an *IMPLEMENTATION* module specifies a different result type from the *DEFINITION* module.

`230 Exported object not declared`

A local or nested module has exported a name, and that name has not been declared within the module. The error is reported at the end of the module, and quotes the offending name.

`231 Too many constants in enumeration`

There is an implementation limit of 256 values in an enumeration type. If you need more (!), you will have to use a CARDINAL subrange to represent them, and take appropriate care to avoid using CARDINAL operations which would be meaningless on the enumeration.

`232 Designator is not a record type`

A *WITH* statement allows shorter references to record fields by implicitly prefixing all relevant identifiers with that field name; clearly, the name following *WITH* must be the name of a variable of record type (it may be a complex reference, such as $a.b[c].e[f]$, but the end result must be a record).

`233 Fieldname not known for this type`

In the syntax $a.b$, a is a record variable name, but b is not a field of that record type.

`234 Attempted field selection not on a record structure`

In the syntax $a.b$, a has been determined not to be a module name; however, it is also not a variable of record type, so that the apparent selection of field b is invalid.

`235 Designator is not a variable`

There are various places where an object must be a variable: on the left-hand of an assignment; as the control variable of a *FOR* loop; anywhere where subscripting, field selection or dereferencing is performed; as the prefixing object of a *WITH* statement.

`236 Attempted pointer dereference not on a pointer type`

Clearly, only pointers can be dereferenced '^'- that is, variables whose type is *POINTER TO something* or *ADDRESS*.

`237 Attempted array index not on an array type`

Only arrays can be subscripted ([ ]).

`238 BY expression not within INTEGER value range`

The expression which gives the step between successive values of a *FOR* loop control variable must take *INTEGER* values; this is the number of values 'forward' or 'backward'. Even in the case of control variables of type CHAR, or some enumerated type, etc., the step must be the *INTEGER* number of values. A large *CARDINAL* ($>$ *MAX*(*INTEGER*)) is also unacceptable.

`239 Control variable not found in local scope`

Good (structured) programming practice suggests that the control variable of a structured statement such as a *FOR* loop should be declared locally — i.e. in the procedure whose body contains the loop. Modula enforces this good practice. This error occurs if the control variable is relatively global to the procedure.

`240 Control variable must not be a formal parameter`

Continuing on from error 239: one way in which the control variable could be a local name for a non-local object is via the parameters. Modula forbids even the use of a value parameter as a control variable.

`241 Control variable must not be imported or exported`

Another way of making an external variable appear local, and thus avoiding errors 239 & 240, is to *IMPORT* it from an enclosing module, or have it *EXPORT*ed by a nested module. Both of these are also unacceptable as *FOR* loop control variables.

`242 Selectors not permitted on constant`

Literal strings cannot be indexed, unlike constant constructors. In any case, you cannot 'select' a component by pointer dereferencing.

`243 Selectors not permitted on procedure name`

Procedure variables are unstructured — you cannot 'select' a component by array indexing, record field extraction or pointer dereferencing.

`244 Standard procs are not valid as proc-values`

It is a rule of the language that standard procedures may not be assigned as values of procedure variables. You can work around this restriction by declaring a procedure whose sole purpose is to call the standard procedure; since it is user-defined, it may be assigned to procedure variables. Note that the user-defined procedure must be declared in the module scope, not within any procedure, to conform with Modula's other restriction on procedure variable values (see error 287).

`245 Function name not known in this scope`

An object which appears to be a function call (e.g. b in `a := b(..)`) is not declared in, or *IMPORT*ed or *EXPORT*ed into this scope or any enclosing scope.

`246 Designator is not a function`

An object which appears to be a function call is known to be some other type (including a proper procedure).

`247 Designator is not a set type name`

An object which appears to be a set designator (e.g. `a{...}`) begins with an identifier (`a` in the example) which is not a set type.

`248 Too few parameters`

A procedure or function call does not have as many parameters as required by the procedure or function declaration.

`249 Designator is not a procedure name`

An object which appears to be a procedure call (e.g. `a(...);`) is not. It may be a function, or some other type.

`250 Designator is not a procedure variable name`

A variable used in the style of a call to a procedure variable (e.g. `a(...);`) is not a procedure variable.

`251 Missing function return expression`

A function procedure has no *RETURN* statement and so cannot return a result.

`252 Proper procedure cannot return a value`

The *RETURN* statement in a proper procedure should not specify a return value; proper procedures return values only via *VAR* parameters, while function procedures should return a single result.

`253 Actual value parameter not assignment compatible with formal`

Since a formal value parameter is treated as a local variable to which the value of the actual parameter is assigned at procedure entry, the same compatibility rules as for assignment statements apply. Some special cases: the second parameter of *INC* and *DEC* must be assignment compatible with either *INTEGER* or *CARDINAL*, depending on the type of the first parameter; the second parameter of *ROTATE* and *SHIFT* must be assignment compatible with *INTEGER*.

See error 258 for discussion of assignment compatibility.

`254 Actual variable parameter type not identical to formal`

Since a formal variable parameter allows direct access to the corresponding actual parameter, the compatibility requirements are strict — actual and formal must be of identical type. Note that *identical* means the same named type — it is *name equivalence* which is required, not just *structural equivalence*.

`255 Actual variable parameter must be a variable`

Since an actual variable parameter may be altered by a procedure, it must be a variable. Constants could not be altered, and expressions have no memory location to hold the updated value.

Also returned if the argument of ADR is not a variable.

`256 Actual parameter corresponding to open array formal not an array`

With the exception of the *universally conformable* *ARRAY OF WORD* and *ARRAY OF BYTE*, open array parameters are compatible only with actual parameters which are arrays of the appropriate type.

`257 Incompatible open array element type`

An open array formal parameter has an actual which is an array, but of the wrong element type.

`258 Expression not assignment-compatible with variable`

Assignment compatibility is required in an actual assignment statement, and also between the initial and final values of a *FOR* loop and the control variable.

Assignment compatibility is defined as follows: identical types are compatible; *INTEGER* and *CARDINAL* are compatible; a subrange is compatible with its host type; strings are compatible with string variables of equal or greater length.

`259 Return value not assignment-compatible with function type`

The value *RETURN*ed by a function procedure must be assignment compatible (see error 258) with the declared result type of the function.

`260 Designator is not a function variable name`

A variable used in the style of a call to a function variable (e.g. b in `a := b(...);`) is not a function variable.

`261 Selectors not permitted on set type name`

Components of set variables cannot be 'selected' by array indexing, record field extraction or pointer dereferencing.

`262 HIGH may only be applied to open array parameters`

The built-in function *HIGH* applies only to open array formal parameters. For normal array parameters, the upper bound is known from the type. If general size information is required, the *SIZE* or *TSIZE* functions should be used.

`263 Expression is not of type CHAR`

The operand of standard function *CAP* must be *CHAR*.

`264 Name of qualifying module clashes in outer scope`

The name of this nested module, which is visible in the enclosing scope, has already been used there for some other purpose.

`265 Enumeration constant name clashes in this scope`

This enumeration value name has already been used in the current scope. Note that the error message quotes the offending name in the case where only the enumeration type name was explicitly imported, but the resulting import of each of the value names caused the conflict.

`266 Name clashes with an enumeration constant name`

This is a more specific version of error 203: the name you have used is already used in this scope, as the name of a value of an enumeration type. Since it is easy to overlook enumeration names, this specific error highlights the problem. The clashing name may also have entered the scope by importation of its type name.

`267 Duplicate case selector in this range`

Each case branch selector must occur only once, so that the statements to be performed are uniquely determined. Have you included this value in a range, as well as this occurrence?

`268 Operand not of signed numeric type`

The arithmetic negation operator (unary or prefix minus) can only be applied to operands which are of signed type; it is illegal on a *CARDINAL* operand, or any non-numeric operand. Note that binary or infix minus may be applied to *CARDINAL* operands; thus for *CARDINAL* a and b the expression $a - b$ is valid, while $-b + a$ is not.

`269 Operand(s) not of Boolean type`

The single operand of *NOT*, or either operand of *AND* or *OR*, must be Boolean.

`270 Operand(s) not of numeric type`

Arithmetic operators apply only to numeric operands.

`271 Operand(s) not of whole number type`

The operators *DIV, MOD, REM* and '/' apply only to *INTEGER* or *CARDINAL* operands, implementing whole number arithmetic.

`272 Operand may not be compared`

Structured types such as arrays and records cannot be compared. Given your understanding of the elements of the structure, you must write an appropriate *Compare* procedure.

`273 Proper inclusion operator not defined for sets`

Only $<=$ and $>=$ inclusion operators are defined for sets. If you wish to test for proper inclusion, replace $a < b$ by $((a <= b)\mathrm{AND}(a <> b))$.

`274 This type may only be compared for (in)equality`

The only comparison operators defined for types *ADDRESS, WORD, BYTE, POINTER TO ..., PROCEDURE*, and opaque types are '=' and '<>'. If you believe a meaning can be attached to other comparisons, you must first coerce or cast to an arithmetic type, on which those operations are allowed.

`275 Right operand or first parameter not of set type`

The set membership operator *IN* tests *member IN set*; thus the right operand must be of some set type. The built-in include and exclude procedures *INCL* and *EXCL* have parameters (set, member); thus the first parameter must be of some set type.

`276 Exported enumeration constant clashes in outer scope`

Another variation of the problem reported by error 265: when you export an enumeration type you also export the name of each value of the type. One of those names has already been used in the scope into which you are exporting the type.

`277 Procedure in !LIBRARY module calls non-library procedure`

A module with the !*LIBRARY* pragma assures the compiler that it does not perform direct or indirect recursion, thus allowing the compiler to make optimizations. This assurance is invalidated if a procedure in the module calls a procedure in another module and that other module does not guarantee !*LIBRARY*.

`278 EXIT not within a LOOP`

The *EXIT* statement exits from the nearest enclosing *LOOP* statement, continuing execution with the statement after the *LOOP*. This *EXIT* statement is not within any *LOOP*.

`279 FOR loop control variable may not be modified`

Consistent with its use as the control of a count-controlled loop, a *FOR* loop variable may not be modified. Thus, it may not be assigned a value, or passed as a variable parameter to a procedure which could then modify it. If you wish to pass the value of the control variable as a *VAR* parameter, copy it to another variable.

`280 Name is not a module name`

The name from which you have tried to *IMPORT* is known, but is not an external module name.

`281 Expected proper procedure, not function`

The definition of this procedure specified that it was a proper procedure (i.e., not a function procedure). The implementation conflicts by specifying a function result type.

`282 ALLOCATE not known in this scope`

A call to *NEW* is treated as a call to *ALLOCATE*, with the appropriate size. There is no visible *ALLOCATE* — probably because it was not imported from *Storage*, though a compatible local *ALLOCATE* procedure will suffice.

`283 DEALLOCATE not known in this scope`

Similar to error 283.

`284 Not a valid substitution for NEW or DISPOSE`

The *ALLOCATE* or *DEALLOCATE* procedure used to implement *NEW* or *DISPOSE* must be of the form *PROCEDURE*(*VAR ADDRESS, CARDINAL*).

`285 Type ranges do not overlap at all`

In checking assignment compatibility and the need for range checking, the special case of a complete mismatch of two subranges causes this error.

`286 Selectors not permitted on type identifier`

Types cannot be 'selected' by array indexing, record field extraction or pointer dereferencing. Only variables of the appropriate structured types can be so selected, leading to components of those variables.

`287 Nested procedures are not valid as proc-values`

It is a language restriction that procedures assigned to procedure variables must be declared at the module level; i.e., they cannot be declared within procedures. (Actually, they must be at the outer level of a *static* module, that is one not nested within a procedure.)

`288 Implementation restriction: case range too large`

The implementation restriction on the range of case labels from smallest to largest is 1024. If you wish to use a larger range, *IF* statements to select appropriate ranges can be used, and are likely to be more compact.

`289 Duplicate identifier in export list of module`

This identifier being exported has already been mentioned in the export list.

`290 Actuals passed to amorphous formals must be simple`

Amorphous open array formal parameters (i.e. *ARRAY OF WORD* or *ARRAY OF BYTE*) will accept almost any actual parameter; however, there are some restrictions imposed by the implementation. Any variable is ok; expressions in general are not. Some special non-variables are allowed: string constants (including single characters), set constants, and set expressions which occupy more than a word and so are held in temporary variables.

If you wish to pass one of the prohibited objects to an amorphous open array parameter, simply assign it to a local variable and pass that.

For *ARRAY OF WORD*, there is of course the extra restriction that the actual must be word aligned and a multiple of word size (see error 307).

`291 No literals except sets and strings allowed here`

As described under error 290, it is an implementation restriction that in general constants are not allowed as actual parameters corresponding to amorphous open array formals. For similar reasons, it is not allowable to *CAST* constants to other types, except in these special cases.

`292 Values cast to structured types must be simple`

Values to be cast to structured types must have an address at runtime. Expressions are in general not acceptable — see error 290 for details and avoidance procedures.

`293 Value is too large to cast to unstructured type`

Unstructured types occupy one word or less; you are not allowed to *CAST* larger objects to these types, as there is no definition of which bits are to be retained and which discarded.

`294 Actual parameter must be a pointer type`

The first parameter of the *SYSTEM* procedures *INCADR, DECADR, DIFADR* must be either a pointer type or *ADDRESS*; so must the second parameter of *DIFADR*.

The parameter of *NEW* and *DISPOSE* must be a pointer type (not *ADDRESS*).

`295 Right operand must be greater than zero`

The whole number modulo arithmetic operators *DIV* and *MOD* are defined only for positive right operands. For negative right operands, the quotient remainder operators '/' and *REM* are available, but note that they have different semantics from *DIV* and *MOD* for negative left operands.

`296 FOR loop control variable is threatened in uplevel access`

The loop control variable has been threatened inside the body of a nested procedure. Threatening actions include being assigned to, passed as a *VAR*-mode parameter, having its address taken, or subjected to *INC* or *DEC*.

`297 FOR loop control variable is threatened`

The loop control variable has been threatened inside the body of the loop. Threatening actions include being assigned to, passed as a *VAR*-mode parameter, having its address taken, or subjected to *INC* or *DEC*.

`298 Feature not implemented -- read latest release notes`

Currently:

`299 Multi-dimensional open array parameters not implemented yet`

ARRAY OF ARRAY OF ... is not implemented, pending definition of what actual parameters are compatible.

`300 Incompatible keys for symbol files`

When *IMPORT*ing from various modules, repeated references to the same module are checked for the same version keys. Thus, for example, if module A imports from modules B and C, and each of B and C in turn imported from D, the two references to D must be consistent. Note that this check also applies to an *IMPLEMENTATION* module's implicit import of its own *DEFINITION* file; thus if the implementation of A imports from B, and B's *DEFINITION* imported from A, the check for consistency between the current A and that via B is made.

You must determine which module(s) are obsolete, and recompile in the appropriate order. Use of the −V (super-verbose) option of **gpm** is strongly recommended.

`301 Wrong name in symbol file`

The module name in the symbol file (quoted in the error message) is not the expected one (that in the *IMPORT* statement). With the symbol file name normally derived from the module name, this will occur only if another file has been renamed to the symbol file name. It is common to rename object files (or better still, use the **−f** option) when there are several alternative implementations, but you should *never* rename a symbol file.

`302 Linker name is not unique`

Since linker names are constructed from the first 10 characters of the module name followed by the first 20 characters of the exported procedure or variable name, it is possible for clashes to occur within a compilation unit, where the full names would not clash. Thus, for example, procedure *FilesMod001.WriteToLogfileAndStdError* would produce the linker name `FilesMod00_WriteToLogfileAndStd`. This name would clash with the name formed for the procedure *FilesMod002.WriteToLogfileAndStdOut*. Clearly, avoidance of module and exported procedure / variable names with long common prefixes will prevent this problem. This problem does not arise for non-exported names, since **gpm** itself takes account of all characters of identifiers no matter how long.

Note that further clashes may arise at build time, due to names constructed in independent compilation units.

`303 Fatal circular import through this module`

This *DEFINITION* module indirectly imports itself. This can only occur if it imports from another module which in turn imports from an **earlier version** of the module being compiled.

`304 Target object has zero storage size`

The object pointed to by the parameter of *NEW* has zero size. If the default *ALLOCATE* procedure was the result of the substitution for *NEW*, a run-time error would result.

`305 Header file not found`

It is assumed that the C header file for the runtime system file `m2rts.h` exists somewhere in the path given by the environment variable *$M2SYM*.

`306 Library name has bad format in header file`

This error is not used in the current version of **gpm** .

`307 Expression cannot be aligned with specified type`

When an expression is *CAST* to another type, the alignment requirements of the new type must not be any greater than the old. Since *CAST* does not generate any code, the old bit pattern, in its old alignment, must be usable as a value of the new type. Thus, for example, an *ARRAY* [1..4] *OF CHAR* cannot be cast to *CARDINAL*, since the required word alignment cannot be guaranteed.

`308 Ident was already uplevel referenced in this scope`

The identifier being declared has already been used in the current scope to reference an object in an enclosing scope — thus there is a conflict between the apparent reference to its previous (uplevel) meaning and the attempted new meaning.

`309 Procedure declared FORWARD was not elaborated`

A procedure declared using the *FORWARD* keyword was not elaborated within the same block. It is a rule that the forward declaration and its elaboration must be in the same lexical scope.

Remember that forward declarations are not required for **gpm**. The keyword is only recognized in order to provide source code compatability with those Modula compilers which have single-pass restrictions.

`310 Array exceeds machine size limit`

The C compiler limits the size of an array, and its index range, to less than 2^{31}. Even for sizes just less than 2^{31}, the loader fails with an obscure error. **gpm** sets an experimentally-determined limit of $(2^{31} - 90000)$, and rejects any larger sizes with this error. In **gpm-pc** this limit is 64k-bytes.

Note that this is of rather academic interest, since the executable file must be large enough to hold the array in question, and such a 2 gigabyte file would very likely fill the file system. See warning 496.

`311 Parameter name was repeated`

The previous procedure declaration had two formal parameters with the same name

`312 Expression must be a designator`

The first parameter to the exception handler procedure *CALL* must be the designator of a visible procedure, but (unlike the usual case with procedure parameters) need not be declared at level-0.

`313 Constructor has too few elements`

Value constructors for arrays and records must have exactly the correct number of elements, taking into account the multiplicity of values which appear with the *BY repeatCount* construct.

`314 Constructor has too many elements`

Value constructors for arrays and records must have exactly the correct number of elements, taking into account the multiplicity of values which appear with the *BY repeatCount* construct.

`315 Ranges not allowed in record or array constructors`

Ranges (constructor elements of the form *expression .. expression*) are only allowed in set constructors. They are not permitted in record or array contructors.

`316 Replicators only allowed for array constructors`

Replicator clauses (constructor elements of the form *expression BY repeatCount*) are only allowed in array constructors. They are not permitted in record or set constructors.

The plausible use of replicators in records with repeated elements of the same kind is not supported by ISO WG-13.

`317 Repetition count must be positive`

The repetition count in an array constructor must be a positive, constant value. Probably it does not make any sense unless it is a non-zero, positive, constant value.

Check that you have gotten the order of expressions correct. The left-hand-side of the *BY* is the expression value which you wish to replicate, the right-hand-side is the number of times which you want it repeated.

`318 Illegal assign of INTERFACE proc with open array`

Procedures from *INTERFACE* modules with open arrays have parameters passed in a different way to Modula procedures (no *HIGH* value is passed). Such procedures can only be assigned to variables of procedure types which are also imported from *INTERFACE* procedures.

If you really do need to assign this procedure, write a dummy interface definition which exports a procedure type with a conforming parameter list. Import this procedure type and declare the variable to be of this type. This is pretty tricky stuff — read the chapter *Interfacing to other languages* before going any further.

`319 Open arrays may only be accessed element by element`

Open array formal parameters may only be accessed one element at a time. In cases where all elements are to be accessed it is necessary to write code along these lines —

```
FOR index := 0 TO HIGH(param) DO
  -- do something with param(index) ...
```

`320 Procedure declaration nesting limit has been exceeded`

Procedure declarations may only be nested 18 deep. That is 17 procedures nested inside each other, inside some enclosing module. In counting the levels when modules are nested inside procedures, only the levels of *procedure* declarations count. This is because the data belonging to such dynamic modules shares the stack frame with the data of the enclosing procedure.

Compared to *Pascal* there is little need to nest procedure declarations in Modula. Instead, visibility of identifiers is best controlled by the declaration of modules.

`321 RETRY is not inside an EXCEPT clause`

The retry statement is only allowed to be used within an except clause, in the same way in which EXIT may only be used inside a loop.

`322 Forward IMPORT not elaborated`

An object IMPORTed into a local module may be declared later in the enclosing module, or in a later local module (provided it is exported from that local module). This message will appear at the end of the scope which is the source of the IMPORT, when the declaration has not so occurred. (cf errors 224, 225 and 309)

`323 Declaration must precede use in a declaration`

Although use of an identifier before declaration is allowed in general, it is not permitted when that use forms part of another declaration (i.e. a use of a type identifier). The exceptions to the restriction are declarations of pointer and procedure types. Most violations of this rule attract error 204 or 205; however, if an IMPORTed identifier which has not yet been declared is used in a declaration this message is given.

`324 Expression must be compatible with control variable`

Modula requires that the type of the upper bound expression of a FOR loop and the control variable have identical types, or share host (of subrange) type. Note that this is more restrictive that the requirement that the lower bound be assignment compatible with the control variable.

13.5 Warnings

`495 Name or function will change next release`

Where it is known that the name, functionality or formal parameters of a built-in procedure will change in the next release this warning is given. Only a single warning is given for each affected identifier, no matter how many times they are used.

In the current release, several *SYSTEM* procedures are known to be changing, due to proposals currently before ISO WG-13.

`496 Array is very large`

As explained under error 310, **gpm** allows arrays whose size may well embarass virtual memory management, file system capacity, or both. Since such a large size may be unintended, this warning is given for sizes greater than 16 megabytes.

`497 Last type has zero storage size`

The type used in this type or variable declaration has no useful capacity. While this is syntactically legal and may occur deliberately in some test programs, it may also result inadvertently from the deletion of code, or code ignored due to comment brackets.

`498 Case statement has very low density`

Of the range of values used by this *CASE* statement, less than 25% are actually referenced; since the compiler typically implements a *CASE* statement as a jump table, relatively large amounts of code may be generated. If the Modula code can be expressed as *IF* statements without loss of clarity, more compact code will usually result.

`499 Variant tags are ignored in this implementation`

In arguments to *SIZE, TSIZE, NEW* and *DISPOSE*, variant tags may be given to specify the size of a particular variant. **gpm** ignores these, and uses the size of the largest variant.

`500 Symbols follow module end`

It is legal to have further text following the *END ModuleName.* which must terminate any compilation unit; however, since this may not have been intended, this warning is issued.

`501 Obsolete syntax, colon is compulsory`

Modula syntax requires that an undistinguished variant record be declared as *CASE* : *TagType OF ...*; however, the older form *CASE TagType OF ...* is also accepted by **gpm** , with this warning.

`502 Obsolete syntax, export list is ignored`

Modula no longer requires that objects be explicitly exported from definition modules — all names defined in definition modules are exported automatically. The older form is simply ignored.

`503 Invalid option selection character (I,R,F,C only)`

Any comment beginning with optional whitespace followed by '$' is assumed to be an option selection. Like any comment, it has no effect on the meaning of the program, but nevertheless is interpreted by the compiler as a direction to generate code which implements that meaning in a specific way (if possible). The two characters following the $ must be an option selector character and the command character; for details, see the *Implementation Specifics* Chapter. This comment appears to be an option selector, but has meaningless option selector character.

`504 Too many levels of option restoration`

One of the option commands, specified by '=', is to restore the option setting to its value before the last change. If an '=' command has no corresponding change to undo, this warning results. Note that changes nest only to a depth of 8, so that the warning may be due to an attempt to restore the earliest of more than 8 nested changes to the same option.

`505 Invalid option operator (+, -, = are valid)`

The only option commands available are those specified by the three operators:

+ set option on (saving previous setting, up to 8 levels)

− set option off (saving previous setting, up to 8 levels)

= restore option setting to value saved before last change

`506 Obsolete syntax, use SYSTEM.CAST for type transfers`

The syntax of the unsafe, unchecked cast from one type to another is *CAST* (*NewType, value*); however, the older form *NewType*(*value*) is accepted.

`507 Procedure is not called, assigned, or exported`

Clearly, this procedure is of no use. This may be a valid state during program development, or it may reflect a procedure no longer used. It may, however, be a symptom of a mis-spelt reference to a procedure which unfortunately happened to match another procedure.

`508 No EXIT from this LOOP`

This *LOOP* is 'infinite' since there is no *EXIT* or *RETURN* within it. This may be deliberate, but may be an omission. Note that *EXIT* escapes from only the innermost enclosing *LOOP*, so that an outer *LOOP* will still need an *EXIT*. By contrast, the use of *RETURN* forcibly terminates all loops, and the procedure in which they are enclosed.

`509 Priority not implemented, ignored`

The module priority concept is not implemented, due to the lack of low-level access into *UNIX*.

Chapter 14

Interfacing to other languages

WARNING

The creation of the foreign and interface files which are described here is discouraged. The creation of such modules is not difficult, but tends to be more error prone that modules created entirely within the Modula system. Disadvantages include loss of the normal Modula type security, and the bypassing of the full power of the automatic version checking of the builder program. Furthermore, the runtime code produced will not have the important runtime checks which Modula includes as the default. Errors may show themselves as error messages from the C compiler cc or from the linker.

Despite all of these reasons, gaining access to existing software is an important goal. Therefore, for those who wish to ignore the above warning, and who have sufficient knowledge of both *C* and Modula, all the necessary details are given here.

14.1 Introduction to the facilities

The compiler possesses special features which allow an easy interface to the standard C-language libraries, and allow implementations to be written in other languages when that is absolutely necessary. Such non-Modula implementations are called *interface* and *foreign* modules. These two kinds of modules are intended for different purposes, and are described here in turn.

Interface modules provide definitions which allow Modula programs to access directly the facilities of the standard system libraries. In this case the Modula names of the objects are identical to the library names, rather than being compound names formed from module-name and identifier. In addition, such names obey language C's lexical rules, rather than the somewhat more strict rules for Modula identifiers.

Foreign modules allow for modules to possess a normal definition file, and to have linker names formed in the standard way, but may be implemented in any suitable language. Many of the libraries supplied with **gpm** are foreign modules, and are all implemented in either C or in assembly language.

These features are based on the ability for a definition part to signal that its implementation will not be a normal object- and reference-file pair. Instead, the definition module signals the source of its implementation by means of a non-standard *IMPORT* statement. This statement, in its simplest form, has the following format —

IMPORT IMPLEMENTATION FROM "*filename*";

for a description of the most general format of the special import statement see the final section in this chapter *The special import statement.*

14.2 Foreign definition part files

Foreign definition parts are used to create modules which are implemented in other languages. However, despite this fact, they have linker names which obey the normal **gpm** linker conventions. Typically they are used when there is some particular reason for believing that an implementation in a language other than Modula will bring some special advantage. They are *not* used for interfacing to existing library codes. For a simple solution to that problem see the next section *Interface definition part files.*

The compilation of an implementation file by **gpm** creates an object file and a *reference* file. The reference file contains the information which is used by **build** to generate the linker script and the initialization call chain. Since the implementations of foreign modules are not processed by **gpm**, no reference file will be produced. Therefore, a different mechanism must be provided for signalling to **build** that the relevant object file should be linked. This is done by using the non-standard import statement.

Consider the following example. A module *Foo* is to be defined with an implementation which will be written in C, and compiled into an object file called *foo.o.* The foreign definition file is as follows —

```
(* first example of a foreign module *)
FOREIGN DEFINITION MODULE Foo;
  IMPORT IMPLEMENTATION FROM "foo.o";
  TYPE FZZ = ARRAY [0 .. 7] OF CHAR;
  VAR  fzz : FZZ;
  PROCEDURE Bar(str : ARRAY OF CHAR);
END Foo.
```

The first token of this file, *FOREIGN*,[1] warns the compiler that this is a foreign definition module. The compiler does two things as a result: firstly, it looks for the special import statement, and then it flags the symbol file as foreign, so that later processing will not search for a reference file.

When this file is compiled, a symbol file is produced in the normal way. The special import statement causes information to be placed in symbol file *foo.syx* which will cause **build** to link the object file *foo.o.*

[1] It should be noted that *FOREIGN* is a **context sensitive mark**, rather than a keyword. The identifier *FOREIGN* may be used in the normal way even within the same program. The identifier only has its special meaning when it occurs immediately before the keyword *DEFINITION.*

Foreign modules may contain procedures and functions, types, variables and constants. The above example does not demonstrate constant definitions.

If a module is now written which imports module **Foo**, the symbol file is read in the normal fashion. In particular, any access to the objects of the module are checked in the same way as if **Foo** was a ordinary module. Suppose, we have

```
MODULE FooUser;
  FROM Foo IMPORT FZZ, fzz, Bar;

  TYPE StrProc = PROCEDURE(ARRAY OF CHAR);
  VAR  alias   : StrProc;

BEGIN
  fzz := "help.. ";
  Bar(fzz);
  alias := Bar;
  alias("ending");
END FooUser.
```

gpm will generate code to call the procedure *Bar* in exactly the same way as if the module was normal. The external names will be formed in the usual way. For example, the call to the procedure *Bar* will be directed to the linker symbol `Foo_Bar`, as is normal.

The actual parameters passed to *Bar* are a pointer to the array variable and the *HIGH* value. Since *fzz* is an array of eight elements, the high value wil be 7.

A possible implementation of *Foo* in language C might look as follows —

```
/* C implementation of module Foo */
#include <stdio.h>
unsigned char Foo_fzz[8];
/* note that Modula CHAR == unsigned char */

void Foo_Bar(p,h)
    unsigned char *p; unsigned h;
{
    for (; h >= 0; --h, ++p) {
        if (*p == 0) break;
        putchar(*p);
    }
}
```

This module can be compiled using command `cc -c foo.c`, in the normal way, and will produce output file **foo.o**. It will export the external symbols `Foo_Bar` and `Foo_fzz` to the linker.

14.2.1 Points to watch

Foreign implementation modules do not have the usual facilities for automatic initialization. The simplest solution to this problem is to generate an explicit initialization function which may then be explicitly called from a suitable point within the importing program.

Notice that the C compiler has no way of checking that the implementation conforms to the interface promised by the definition file. Programmers thus bear an increased level of responsibility for ensuring correctness. In particular, the conformance of the formal parameter list in the foreign definition and the formal parameter list in the foreign implementation should be checked most carefully, as should the actual linker names generated.

Finally, the conformance between the types used in C and the types declared in the definition should be checked using the information in the chapter *Implementation specifics*. In complex cases, such as C functions returning records structures, it may be useful to check the C intermediate code output of an importing file by using the **gpm –S** option.

14.3 Interface definition part files

Interface definition parts are used to allow Modula programs to access the facilities of libraries written for C or other language systems. Unlike the case of *foreign* definition modules described earlier, the names which appear in the definition part file are exactly the names as they are known to the *UNIX* linker.

Consider the following example which provides an interface to the hyperbolic trigonometric functions of the mathematics library.

```
INTERFACE DEFINITION MODULE Hyperbolic;
  IMPORT IMPLEMENTATION FROM "-lm";

  PROCEDURE sinh(x : REAL) : REAL;

  PROCEDURE cosh(x : REAL) : REAL;

  PROCEDURE tanh(x : REAL) : REAL;

END Hyperbolic.
```

The first token of this file, *INTERFACE*,[2] warns the compiler that this is an interface definition module. The compiler does several things as a result: first, it looks for the special import statement, and then it flags the symbol file as an interface symbol file, so that later processing will not search for a reference file. As well as this, the mark tells the compiler to treat the defined names literally, so that the external name of function *sinh* is `sinh`, and not the `Hyperbolic_sinh` which would otherwise be expected.

Open array parameters in interface definition files work differently than is the case within the Modula system. Procedures from interface definition modules with open array parameters do not have corresponding *HIGH* values passed to them. Note carefully that this is in constrast to the case with *foreign* definition files, which obey Modula parameter passing conventions.

Finally, the occurrence of the special mark warns the compiler to relax the lexical conventions for identifier names so that strange C identifiers such as "`__X_X__`" are permitted.

[2]It should be noted that *INTERFACE* is a **context sensitive mark**, rather than a keyword. The identifier *INTERFACE* may be used in the normal way even within the same program. The identifier only has its special meaning when it occurs immediately before the keyword *DEFINITION*.

Under normal circumstances such an identifier would provoke a lexical error 9. This relaxation of naming rules applies also to any modules which *import* an interface definition (otherwise they wouldn't be able to reference objects with names such as `__X_X__`).

When the file in the example is compiled, a symbol file is produced in the normal way. The special import statement causes information to be placed in symbol file *foo.syx* which will cause **build** to search the library *libm*.

14.3.1 Open arrays and interface definitions

Functions in C handle arrays in a different (and less secure) way than does Modula. In C, arrays formals are expected to receive a pointer to the element type as actual parameter, and the length of the array is signalled in some other way, such as the occurrence of an *Ascii.null* byte.

gpm understands this convention, and will suppress the passing of the usual *HIGH* value to interface procedures. As an example, an interface to the standard C library function *strcmp* could be defined as follows —

```
INTERFACE DEFINITION MODULE StrCompare;
  (* no special import statement needed *)
  TYPE  Result = INTEGER;
  (*
   * less    ==> result < 0
   * equal   ==> result = 0
   * greater ==> result > 0
   *)
  PROCEDURE strcmp(s1, s2 : ARRAY OF CHAR) : Result;
END StrCompare.
```

In this case, any call to *StrCompare.strcmp* will create a call to the external function with linker name `strcmp`. Such a call will have only two parameters passed to it, instead of the usual four. The two parameters correspond to the C declaration

```
int strcmp(s1,s2)
    unsigned char *s1, *s2;
{
    ...
```

Note that the actual library function *strcmp* may expect the two arguments to be arrays of *signed* characters, while **gpm** expects characters to be unsigned. Users should carefully consider the implications of this fact if they are using 8-bit character sets in a mixed language environment.

The reason that no special import statement is required in this example is that the function *strcmp* comes from the library *libc* which is always searched anyway. In effect there is always an implicit special import statement equivalent to

```
 IMPORT IMPLEMENTATION FROM "-lc";
```

14.3.2 Points to watch

As well as the usual points which apply to *foreign* modules, the following point needs to be considered.

Interface modules do not expand names to create compound names which include the module name. There is thus a greater possibility of names from interface modules colliding with each other, and with local symbols of importing modules. Extra care may thus be needed, and a watch kept for possible linker error messages.

Because of the different way of handling open arrays, there are restrictions on the assignment of interface procedures to procedure variables. These restrictions apply to the passing of interface procedures as actual parameters to procedures. Details of these restrictions are given in the following subsection.

14.3.3 Interface procedures and procedure variables

Because interface procedures have open arrays passed to them in a different way to ordinary procedures, a potential insecurity could arise. Suppose unrestricted assignment of interface procedures to procedure variables was permitted. In particular, suppose that some procedure variable contains a procedure value, and the procedure is to be called. The compiler has no simple way of knowing whether the last value assigned to the variable designated a normal procedure or an interface procedure. Short of making all procedure variables carry along a tag value, there is no solution to this problem. Assignments to procedure variables are therefore restricted to avoid this ambiguity.

The simple rule is —

> **Procedures with open array formal parameters which are imported from interface modules may only be assigned to variables of procedure types which are also imported from interface modules.**
>
> **Procedures with open array formal parameters which are declared in non-interface modules may only be assigned to variables of procedure types which are also declared in non-interface modules.**

In effect, the rules for assignment compatability of procedure types with open array formal parameters is augmented by the rule that the source and destination types must both have the *interface* attribute, or neither must have the attribute. **gpm** checks all procedure assignments for this rather subtle property, and issues error 318 in cases of violations.

Getting around the restriction

Suppose that it is wished to use the function *strcmp* in a Modula program, and the procedure is to be assigned to a procedure variable. The following declarations will not work.

```
     FROM C_Strings IMPORT strcmp;
       TYPE CompProc = PROCEDURE(ARRAY OF CHAR;
                                 ARRAY OF CHAR) : INTEGER;
       VAR  compare  : CompProc;
       ...
       compare := strcmp;  (* this is not a valid assignment *)
****   ^ Semantic error 318  ****
```

In order to allow the assigment, the procedure variable must be of a type which is also imported from an interface module (but not necessarily the same one as the procedure value comes from). Thus if it really is necessary to assign this procedure, a dummy interface definition must be produced which exports a conforming procedure type.

```
INTERFACE DEFINITION MODULE CompProcDef;
  TYPE CompProc = PROCEDURE(ARRAY OF CHAR;
                            ARRAY OF CHAR) : INTEGER;
END CompProcDef;
```

The previous example is now completed by importing this type, and declaring the procedure variable to be of the type.

```
FROM C_Strings IMPORT strcmp;
FROM CompProcDef IMPORT CompProc;

  VAR  compare  : CompProc;
  ...
  compare := strcmp;  (* this IS a valid assignment *)
```

Notice that logically it is impossible to declare a procedure variable which may contain either normal or interface procedures with open array formal parameters. For procedure types which do not have open arrays there is no such problem.

14.4 The special import statement

The special import statement is used to send information to **build** in cases where the usual reference file is not created by **gpm** . This information consists is of two possible kinds

- object files which need to be included in the linker script
- libraries which need to be searched

The special import statement allows any number of each of these data to be specified. The syntax of the special import statement is as follows —

specialImport	→	"IMPORT IMPLEMENTATION FROM" **litstring** ';'.
litstring	→	'"' **element** { '&' **element**} '"'.
element	→	object-file-name
	|	'–'library-name.

library names start with a minus sign, while filenames do not.

Note that since the string is a Modula literal string token it cannot extend over a linebreak with the current Modula conventions for strings. All tokens in the string may be separated by spaces, except that there must be no space between the minus sign and the library name. This is the normal *UNIX* convention.

When **build** creates its linker script, all the object file names from all of the modules in the program occur first, and are followed by all the library names. Thus the order of libraries and object file declarations within a particular interface module does not matter. The effect of the two versions of the following special import statement is the same.

```
IMPORT IMPLEMENTATION FROM "foo.o & -lm";
IMPORT IMPLEMENTATION FORM "-lm & foo.o";
```

Both will cause *foo.o* and *–lm* to appear in the linker script, in that order.

14.4.1 Where can the special import statement appear?

The special import statement may occur anywhere before the first definition in the definition file. If may occur before any other imports, after other imports, or in the middle of other imports. It is probably a good plan to place the special import first, in the interests of visibility.

14.4.2 Declaring name aliases

In some systems it may be desirable to access system call facilities which have names which cannot be expressed even using the relaxed lexical rules which apply to *INTERFACE* definitions. For example, in Apollo Domain systems the system-calls have names which include dollar signs ($).

The following mechanism has been decided to handle such cases. The interface definition module may include a **name-alias pragma**. This pragma is of the form —

```
(* !ALIAS modulaName = "_$illy_name$_" *)
```

The string on the right may be any literal string at all. Internally the defined object, which might be a variable or a procedure, is referred to by the defined name (`modulaName` in the above example). In the intermediate code, the external name is taken from the string. It is still the responsibility of the creator to ensure that the interface for procedures is declared to have the correct parameter mode. In the case of Apollo Domain, for example, all parameters are passed by reference in system calls. Thus in this case, all formal parameters which are not arrays must be defined as being of *VAR* mode.

These facilities are not needed in *UNIX* based implementations, which have C-language bindings to all the system-call facilities. However, this section is included to spell out the mechanism which will be used in all cases where an extension is required. The next release of the Apollo version will capture this feature.

Chapter 15

Coroutines

gpm provides a coroutine library as specified in Wirth's PIM. *ISO WG-13* has proposed a slightly different library, with somewhat enhanced (and somewhat incompatible) features. In due course **gpm** will support the new model also, but will continue to ensure compatibility for existing programs which use the traditional library.

15.1 Introduction to coroutines

The coroutine facilities of Modula-2 allow *multi-threaded* programs to be constructed. In such programs, several threads may be at various stages of execution at the same time. These threads are *quasi-concurrent*. That is, only one thread is actually active at any one time, but by interleaving the execution of the various threads all may progress apparently in parallel. The use of couroutines allows certain unique forms of program organization which are rather under-utilized in current practice, probably since few languages support coroutine primitives. In particular, coroutines form a natural foundation for simulation programs. Program threads are sometimes also known as *lightweight processes*, since they provide some of the functionality of *UNIX* processes, but are many, many orders of magnitude less costly in execution time.

Execution of each coroutine is explicitly suspended by transferring control to another coroutine. Each coroutine has its own activation stack at runtime, and these stacks are explicitly created and initialized by a call to the procedure *Coroutines.NEWPROCESS*.

Programs which do not use the coroutines library, so-called *single-stack programs* have little need to perform stack overflow testing. Typically, several hundred megabytes of virtual memory are available for expansion of the stack segment of such programs, although it is usual for *UNIX*́s process size limit to be exceeded well before this. Programs which use the coroutines library have a separate stack for each coroutine, suggesting the prudent use of stack overflow testing. The facilities provided for this are also available for single-stack programs, although the default continues to be for stack overflow testing to be disabled.

15.1.1 The Coroutines library

The coroutines library is an ordinary library in the sense that no knowledge of the library is required by the compiler. It is implemented in assembly language and uses the usual *FOREIGN* mechanism. The library must be explicitly imported by user programs, in keeping with the proposals of *ISO WG-13*. However, this version implements the *old* coroutines model exactly as specified by Wirth.

The source code of the implementation is included in the **gpm** distribution. The implementation is written in the assembly language of the target computer architecture. This code is extremely dependent on the exact procedure calling conventions of the Modula and C compilers, and is tightly coupled to the facilities of the runtime system. Modification of this code, except as suggested by any future upgrade notices, is not recommended.

15.1.2 Procedure NEWPROCESS

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

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

The first parameter

The first actual parameter of the procedure call must designate a parameterless procedure value, that is a *PROC*. The actual parameter must thus either be a procedure of this type, or a procedure variable of this type.[1]

The second and third parameters

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

How much workspace is required?

The workspace has three components. They are the coroutine state vector, the private soap-space, and the normal coroutine private stack. The state vector is only a few hundred bytes in size. The size of the soap-space is determined by the environment variable *SOAPSIZE* in a similar way to that used for single stack programs. However, there is no lower limit to the size of soap for coroutines. This is in contrast to the situation with main programs which always get at least 4096 bytes no matter how small the variable is.

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

All of the remaining workspace is available as stack space for the coroutine. The procedure sets a stack overflow limit exactly 512 bytes from the end of the workspace[2]. This safety zone provides space for cleanup procedures (which will execute in the context of the failed coroutine) to run to completion successfully.

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

New runtime error messages

If a coroutine (other than the main process) ends "normally" the following error message is produced.

```
**** m2rts: coroutine ended without TRANSFER ****
```

In the event that the workspace stack limit has been exceeded, and stack overflow testing was used in the procedures of the coroutine body, the following message is produced.

```
**** m2rts: stack limit has been exceeded ****
```

A typical example

In the following program two coroutines are created.

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

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

    ...

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

[2]**gpm-pc** does not have *SOAP* and only reserves 200 bytes of safety zone. In the PC implementation, the minimum workspace size is about 400 bytes

15.1.3 Procedure TRANSFER

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

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

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

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

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

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

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

Transfer speed

On *RISC* architectures, the speed of coroutine transfers tends to be fairly slow, often being a factor of 10 or more slower than a procedure call. This is an inherent property of these machine organizations, and arises from the comparatively large processor state, due to the typically large size of the register files. As an example, **mips** R2000 based machines require about 14.5 μsec to perform a transfer. This implies that no more than approximately 50 000 transfers per second may be expected. The situation is even worse with the SPARC architecture which requires register windows to be saved.

Appendix A

Debugging with gdb — getting started

A.1 Introduction

Several of the native code versions of **gpm** use `gdb` as the standard debugger. This provides portability, since `gdb` works in essentially the same way on each of the platforms on which it is available, apart from some minor differences inherent in the different machine architectures. `Gdb` is the standard debugger from the *Free Software Foundation* (FSF). `Gdb` is not part of the **gpm** distribution, and must be obtained separately, from one of the usual freeware sources. Some versions of **gpm**, such as **gpm-solaris** use the vendor's own assembler, but still produce the symbol table information which is required for `gdb` to work. In the case of **gpm-linux** and **gpm-djgpp** the Modula-2 system uses the FSF's assembler `gas` in the same way as FSF's C-compiler `gcc` does.

`Gdb` is a *symbolic* debugger, which is to say that it knows about the names and structures of various data elements in the program. Variables may be referred to by name, and in the case of aggregates, components of structures may be accessed by name or by index. `Gdb` understands Modula-2's record, array, pointer, enumeration and subrange types. It also understands a subset of the Modula-2 syntax of expressions, so that components of values may be referred to by the familiar syntax.

Although `gdb` is a symbolic debugger, it does work with the representation of data, rather than with the abstractions which appear in our program source codes. For example, when we communicate with `gdb` we need to take account of the fact that parameters of *VAR* mode are actually pointers to the corresponding parameter. Thus it is difficult to use such a debugger for any of the more complex tasks without at least being aware of some of the issues of data representation which the language itself hides away.

In principle, `gdb` could be used on programs with multiple coroutines, through the light-weight thread support facility. However, this has not been implemented in the current version. If `gdb` is used on a program with multiple coroutines, only the state of the currently executing coroutine can be examined.

All of the examples in this introduction have been copied from the output of a **gpm-solaris** system called "grange" running Solaris 2.3, and gdb 4.12. There may be minor differences of detail with other versions.

A.1.1 Preparing a program for debugging

Programs compiled with **gpm** can be compiled with more or less information available for the assistance of the debugger. Programs compiled with the **–g** command line flag (the "g-flag") have full debugger support. For some platforms, `gdb` can extract useful amounts of information from programs which have not been compiled using the g-flag, but this varies from platform to platform.

Using the g-flag has some penalties at runtime. The size of the executable file on disk may be increased by as much as 50%, although the size in memory does not necessarily increase. As well, the use of the g-flag is incompatible with some of the optimizations which **gpm** uses so that programs compiled with the g-flag may also run slower by a small factor.

In general, if a large program is being debugged, only the modules of interest need to be compiled with the g-flag. This strategy may be used to limit the amount of recompilation which is required to provide complete debugging support. However, if some modules have debugging information, and other modules do not, this needs to be taken into account when examining stack traces, for example.

A.1.2 Name-munging and gpm

Recall that Modula-2 does not require that procedure names be unique within a program. Indeed, two modules may even export procedures with the same name. The standard linker programs require however that all globally visible symbols (such as exported procedures) must have globally unique names. Thus **gpm** modifies the names of all exported procedures and variables so as to make the names unique. It does this by *munging* the names in the following way.

> The external name is formed by taking the module name, and truncating to 10 characters. The object name is similarly truncated to 20 characters, and the two parts connected by an underscore character. The overall effect is to produce an external name which has a similar appearance to the qualified name for the same object.

Thus the familiar procedure *WriteString* from module *InOut* would have qualified name *InOut.WriteString* and external (munged) name `InOut_WriteString`. Similarly, the munged name for *GenSequenceSupport.InitCursor* would be `GenSequenc_InitCursor`.

With the g-flag, **gpm** un-munges the names, so that the names which are known to the debugger are the unqualified names which the original program contained. In the presence of name clashes `gdb` can usually tell which object is required, since it understands the file structure of the original program.

Thus, if a program has been compiled with some modules using the g-flag, and others without, then the stack trace may have a mixture of munged and unmunged names. This should seldom cause any confusion.

The body part of separately compiled modules, that is, the part after the last *BEGIN* in a compilation unit, is called the initialization part. It is so called because it is used to initialize any state which the module encapsulates. So far as the programmer is concerned, these body parts do not have *names*, since they cannot be explicitly called. Instead, they are automatically called, in the approriate order, by the startup code which is generated by the **build** program. These initialization entry points thus do have linker names, which in this

	Modula-2 name	Linker name	`gdb` name (only with g-flag)
Entry point of module *GpFiles*	no name	`InitGpFiles`	`GpFiles`
Main module entry *Example1*	no name	`StartExample1`	`Example1`
Procedure exported from *InOut*	*InOut.Write*	`InOut_Write`	`Write`
Non-exported procedure from *Example1*	*StrOut*	*not visible*	`StrOut`
Variable exported from *InOut*	*InOut.Done*	`InOut_Done`	`Done`
Non-exported variable from *Example1*	*arr*	*not visible*	`arr`
Local variable from *Example1*	*ix*	*not visible*	`ix`

Figure A.1: Munged and un-munged names

case are invented by **gpm** itself. These synthetic names sometimes appear during debugging. The rule which is used by **gpm** for generating these names is to concatenate the string '`Init`' with the module name. In the case of main modules, the body part entry point is named `Start`*ModuleName*. If a module is compiled with the g-flag, then the entry point name is known to the debugger simply as *ModuleName*.

Table A.1 gives examples of all the synthetic names generated by **gpm**, and how they appear to the linker and debugger.

A.2 Post-mortem debugging with `gdb`

On most *UNIX* systems, when a program terminates abnormally, a complete image of the memory of the program is written to disk as the file `core`. This image is called the "core dump". `gdb` can be used to examine such core dumps, in order to find out information about the program at the time of termination. Because an abnormally terminated program is commonly described as having "died", such examination of the memory image is usually called *post-mortem debugging*.

In this section, only the basics of post-mortem debugging are treated. However, many of the data examination facilities described later will work correctly on core files as well as on executing programs.

A.2.1 Examining the procedure call chain

Perhaps the most important information which a programmer wishes to know after a program has crashed, is exactly where the flow of control was when the program crashed. This information is often supplied by the **gpm** runtime system, even without the presence of a debugger. For example consider the following (erroneous) program —

```
MODULE Example1;
    IMPORT InOut;

  PROCEDURE StrOut(str : ARRAY OF CHAR);
    VAR ix : CARDINAL;
  BEGIN
    ix := 0;
    WHILE str[ix] <> "" DO
      InOut.Write(str[ix]); INC(ix);
    END;
  END StrOut;

  VAR arr : ARRAY [0 .. 10] OF CHAR;

BEGIN
  arr := "hello world";
  StrOut(arr);
END Example1.
```

When we compile and run this program we get the following result —

```
grange> gpm example1
grange> build example1
grange> example1
**** gp.rts: index error: 11 not in [0 .. 10] ****
Abort(coredump)
```

The runtime system has signalled an index error. The index has reached 11, when the upper bound of the array is 10. In this case it is not hard to see *which* array bound has been exceeded, since there is only one in this simple example. However, we might like to know *why* it has been exceeded. Is it not a fact that Modula-2 places a nul byte at the end of strings?[1]

So, let us have a look at the core dump. We start up the debugger, with the name of the program to be debugged as the first argument. If we are examining a `core` file, we give this filename as a second argument.

```
grange> gdb example1 core
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.12 (sparc-sun-solaris2.3),
Copyright 1994 Free Software Foundation, Inc...
Core was generated by `example1'.
Program terminated with signal 6, Aborted.
```

[1]No, as a matter of fact Modula-2 doesn't always do this. Read on...

```
procfs (find_procinfo):  Couldn't locate pid 0
#0  0xef7956c8 in _soapLim ()
(gdb)
```

The debugger starts by printing out its copyright notice, and is now ready to accept commands. The prompt is the string `(gdb)`.

Since we are interested in the control path which lead to the crash we ask for a *stack back-trace*, with the command '`bt`'.

```
(gdb) bt
#0  0xef7956c8 in _soapLim ()
#1  0x126f4 in _catcher ()
#2  0x125f0 in _gp_iTrpLHU ()
#3  0x139f0 in StrOut ()
#4  0x13a8c in StartExample1 ()
#5  0x129ac in _gp_Init ()
#6  0x1202c in main ()
(gdb)
```

The top three procedures are part of the operating system or the runtime system of **gpm**, and have to do with printing the error message and halting the program. In this particular example the runtime startup procedure `_gp_Init` called the body part of *Example1*, which called the non-exported procedure *StrOut*. This last procedure has called the index error procedure `_gp_iTrpLHU`.[2] Notice that since the module was compiled without the g-flag it is the munged names which appear in the stack back-trace, and that no line numbers or parameter values are known to `gdb`. Notice also that although the procedure *StrOut* is unknown to the linker, as shown in table A.1, the name is still able to be printed in the stack trace.

`Gdb` is able to display this information because it is able to locate the procedure return information on the runtime stack image in the core file. However, some versions of **gpm** do not create stack frames unless it is absolutely necessary, and hence sometimes may not be able to even produce a stack backtrace without the g-flag.

If we repeat the previous example, but compile with the g-flag then we find the following —

```
grange> gpm -g example1.mod
grange> build example1
grange> example1
**** gp.rts: index error: 11 not in [0 .. 10] ****
Abort(coredump)
grange> gdb example1 core
...
(gdb) bt
#0  0xef7956c8 in _soapLim ()
#1  0x126f4 in _catcher ()
#2  0x125f0 in _gp_iTrpLHU ()
```

[2] Just in case you were wondering, the procedure name indicates that this is an **i**ndex trap procedure, which is passed **L**ow and **H**igh bounds as well as the trapped value, which is interpreted as an **U**nsigned value.

```
#3  0x139f4 in StrOut (str=0x27ed0, str$hi1=10) at example1.mod:9
#4  0x13a94 in Example1 () at example1.mod:17
#5  0x129ac in _gp_Init ()
#6  0x1202c in main ()
(gdb)
```

Now we have line numbers for the procedures of the module. In the case of *StrOut* we also have parameter information. Notice that the name of the main body has been un-munged, and now is just *Example1*. Notice also that the second, hidden *HIGH* value which forms part of an open array parameter is called *str$hi1* and has the value 10. If you count the characters in the literal string in the program you may already be able to work out why the program is wrong.

A.2.2 Examining global and local data

In general we would like to know data values as well as knowing the procedure call chain. `Gdb` allows us to print the values of global and local variables, and to ask about the types of the various data. If a module has been compiled without the g-flag, then `gdb` will only be able to find exported global variables, and will not know their types. In all such cases `gdb` will assume that the variables are of integer type. Such exported global variables have names which are munged by the same algorithm as exported procedure names, as shown in table A.1

With the g-flag, `gdb` knows about the values and types of exported variables, non-exported variables, and local variables of the currently selected procedure. In the case of exported variables `gdb` knows both the munged and unmunged names.

Taking the same example as before, we shall examine the data of the program which is available in the core file. The more complicated commands used to examine data, and which are described later, can be applied here as well.

First we shall look at the non-exported, statically allocated variable *arr*. After starting `gdb` on the core file we first ask for information about this variable using the command '`whatis`' —

```
(gdb) whatis arr
type = CHAR [11]
(gdb)
```

`Gdb` tells us that this is an array of eleven characters. We now ask for the value of the array to be printed —

```
(gdb) print arr
$1 = "hello world"
(gdb)
```

Now, we want to know whether or not the array actually has a nul character at the end. We ask for the last character of the array *arr*[10] to be displayed —

```
(gdb) print arr[10]
$2 = 100 'd'
(gdb)
```

By now the solution to the problem is clear. The last character in the array is 'd' (the 100 is the decimal value of character 'd' in the *Ascii* collating sequence). When the array was declared we did not leave room for the terminating nul character which *StrOut* assumed would be there.[3] Let us nevertheless investigate some of the other facilities of `gdb`.

Looking at the stack backtrace from the last section, we would like to make *StrOut* the procedure of current focus. We can do this by using the '`up`' and '`down`' commands which move the focus to higher and lower numbered procedures on the call chain.

```
(gdb) bt
#0  0xef7956c8 in _soapLim ()
#1  0x126f4 in _catcher ()
#2  0x125f0 in _gp_iTrpLHU ()
#3  0x139f4 in StrOut (str=0x27ed0, str$hi1=10) at example1.mod:9
#4  0x13a94 in Example1 () at example1.mod:17
#5  0x129ac in _gp_Init ()
#6  0x1202c in main ()
(gdb) up 3
#3  0139F4H in StrOut (str=0x27ed0, str$hi1=10) at example1.mod:9
9              InOut.Write(str[ix]); INC(ix);
Current language:  auto; currently modula-2
(gdb)
```

`Gdb` has printed the current line out. Also, because we have switched files from the runtime system (written in *C*) to `example1.mod` (written in Modula-2) it tells us that it will now accept Modula-2 expressions.

If we do not remember the names of the local variables we can ask `gdb` to list the code in the vicinity of the current position by using the '`list`' command, or we can just ask for information on all local variables using the command '`info locals`'. If we ask for the types of the parameters and local variable, we see the following.

```
(gdb) whatis str$hi1
type = CARDINAL
(gdb) whatis ix
type = CARDINAL
(gdb) whatis str
type = CHAR (*)[]
(gdb) print ix
$3 = 11
(gdb)
```

As we by now expect, *ix* has value 11, having stepped off the end of the open array.

The type of *str* requires some explanation. Although `gdb` understands different language rules for expressions, it always prints out values and types using a variation on the style of the language *C*. In this particular case it is sufficient to know that `gdb` is telling us that *str* is a pointer to the type *CHAR*, or possibly the address of an array of *CHAR*.[4] Now, **gpm**

[3] Of course, the procedure *StdOut* shouldn't be written this way. Simply lengthening the array declaration will remove the error, but the program is still badly designed.

[4] *C* uses the asterisk character '`*`' to denote "pointer to" much as Pascal uses the carat character.

passes open arrays by reference, so the value denoted by the variable is actually a pointer to the formal parameter. We may access this in the same way that we would access a parameter of *VAR* mode in `gdb`.

```
(gdb) print str^
$4 = 0x27ed0 "hello world\000"
(gdb) print str^[str$hi1]
$5 = 100 'd'
(gdb)
```

As can be seen, the formal parameter does have a terminating nul byte (in this case quite by accident), but this nul does no good as it is past the end of the array as denoted by the *HIGH* value.

A.3 Runtime debugging

As well as examining core dumps, `gdb` is also able to examine the data and control flow of programs while they are running — well, actually while they are temporarily paused for examination.

In order to do this we start up the program under the control of `gdb`, but without the specification of the core file.

```
grange> gdb example1
...
(gdb)
```

We may run the program until it crashes, using the '`run`' command

```
(gdb) run
 Starting program: /export/home/gough/wrk/example1
 **** gp.rts: index error: 11 not in [0 .. 10] ****
 hello world
 Program received signal SIGABRT, Aborted.
 0xef7956c8 in _kill ()
 (gdb) bt
 #0  0xef7956c8 in _kill ()
 #1  0xef76c36c in abort ()
 #2  0x126f4 in _catcher ()
 #3  0x125f0 in _gp_iTrpLHU ()
 #4  0x139f4 in StrOut (str=0x27ed0, str$hi1=10) at example1.mod:9
 #5  0x13a94 in Example1 () at example1.mod:17
 #6  0x129ac in _gp_Init ()
 #7  0x1202c in main ()
 (gdb)
```

With minor variations, this is now in the same situation as that when the program is started with a core file.

Rather more interesting is the possibility of stopping the program at selected points *before* it crashes. We do this with the '`break`' command. We may ask for the program to be halted at the entry point of particular procedures, or at particular line numbers. The two forms are —

> `break` *Procedure-name*
> `break` *File-name*:*Line-number*

Then when we run the program it will halt at the specified point.

The points at which we ask for the program to be halted are called the breakpoints of the program.

```
(gdb) break Example1
Breakpoint 1 at 0x13a4c: file example1.mod, line 1.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /export/home/gough/wrk/example1

Breakpoint 1, Example1 () at example1.mod:1
1       MODULE Example1;
Current language:  auto; currently modula-2
(gdb)
```

Notice that the program has halted, and is displaying the source line of the module entry. Just to be clear, we look at the stack backtrace.

```
(gdb) bt
#0  Example1 () at example1.mod:1
#1  0129ACH in _gp_Init ()
#2  01202CH in main ()
(gdb)
```

It is important to realise that the source code line which `gdb` displays is the line which is about to be executed. In other words, you get to look at the statement before it executes, and may inspect the data which the statement will use when you give the command to continue.

We may now step through the code, line-by-line, using either '`step`' (step line-by-line), or '`next`' (step line-by-line, but do not step into procedures). Since in this case we do want the control to step into *StrOut* we shall use '`step`'.

```
(gdb) step
16          arr := "hello world";
(gdb) step
17          StrOut(arr);
(gdb) step
StrOut (str=0x27ed0, str$hi1=10) at example1.mod:4
4           PROCEDURE StrOut(str : ARRAY OF CHAR);
(gdb)
```

We are now in the nested procedure. We step some more —

```
(gdb) step
7           ix := 0;
(gdb) step
8           WHILE str[ix] <> "" DO
(gdb) step
9             InOut.Write(str[ix]); INC(ix);
(gdb) next
9             InOut.Write(str[ix]); INC(ix);
(gdb) next
9             InOut.Write(str[ix]); INC(ix);
(gdb)
```

Notice that we switched to '`next`', since we do not want to step into the library procedure. We can print out the values of variables after each step, in order to track progress. In fact, we can place a breakpoint at line 9 and ask for any sequence of commands to be executed at that point. In this case we shall ask for *ix* to be printed, and *str*[*ix*].

```
(gdb) break example1.mod:9
Breakpoint 2 at 01399CH: file example1.mod, line 9.
(gdb) commands
Type commands for when breakpoint 2 is hit, one per line.
End with a line saying just "end".
print ix
print str^[ix]
end
(gdb)
```

We must not forget that to `gdb` *str* is a pointer to an array, so we must use the form `str^[ix]` rather than the `str[ix]` that we would have used in a program.

Now when we restart the program from the beginning we may step from breakpoint to breakpoint using the command '`continue`' which we may abbreviate to just 'c'.[5]

```
(gdb) run
Breakpoint 1, Example1 () at example1.mod:1
1       MODULE Example1;
(gdb) c
Continuing.

Breakpoint 2, StrOut (str=0x27ed0, str$hi1=10) at example1.mod:9
9             InOut.Write(str[ix]); INC(ix);
$3 = 0
$4 = 104 'h'
(gdb) c
Continuing.
```

[5]In fact, all of the commands which have been met so far may be shortened to just a single letter.

```
Breakpoint 2, StrOut (str=0x27ed0, str$hi1=10) at example1.mod:9
9             InOut.Write(str[ix]); INC(ix);
$5 = 1
$6 = 101 'e'
(gdb) c
Continuing.

Breakpoint 2, StrOut (str=0x27ed0, str$hi1=10) at example1.mod:9
9             InOut.Write(str[ix]); INC(ix);
$7 = 2
$8 = 108 'l'
(gdb)
```

... and after some more continuations —

```
(gdb) c
Continuing.

Breakpoint 2, StrOut (str=0x27ed0, str$hi1=10) at example1.mod:9
9             InOut.Write(str[ix]); INC(ix);
$23 = 10
$24 = 100 'd'
(gdb) c
Continuing.
**** gp.rts: index error: 11 not in [0 .. 10] ****
hello world
Program received signal SIGABRT, Aborted.
0EF7956C8H in _kill ()
(gdb)
```

In more complicated cases, it is usually possible to stop a program just before it is going to crash, and examine all the relevant data of the statement about to be executed.

A.4 Dealing with types

`Gdb` is able to understand and display type information from source programs in a reasonably complete way.[6] This type information may be extracted by using the '**whatis**' command. The command takes a variable as argument, and prints the type name if the variable is of a named type, or the type structure if the variable is anonymous.

Suppose we have the following declarations —

```
MODULE Types;
  IMPORT CardSequences;

```

[6] This section deals with more complicated data structures, and it thus unsuitable for beginning Modula-2 programmers. Skip this section at the first reading, at least.

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

VAR  sequ  : CardSequences.Sequence;
     today : Days;
```

If we now query the types and values of the variables during execution of the program, we obtain —

```
(gdb) whatis today
type = Days
(gdb) print today
$2 = {val = tuesday, ord = 1}
```

Notice that when we print out a value of an enumeration type, we are told of both the *value* of the variable, and the ordinal number to which that value corresponds. In this case, since enumeration ordinals count from zero, tuesday is the day with ordinal of one.

For the other variable we shall not only ask what type the variable is, but also what is the structure of the type. We do this with the '`ptype`' (print type) command. In this case we obtain —

```
(gdb) whatis sequ
type = Sequence
(gdb) print sequ
$1 = {first = 0x0, last = 0x0}
(gdb) ptype sequ
type = struct {
    C_char *first;
    C_char *last;
}
```

The Modula-2 declaration of the *Sequence* type is a record with *first* and *last* fields of some opaque type. Notice that `gdb` has printed this out in *C* style as a "*struct*", with the fields being of type "`C_char *`". This corresponds to generic pointers in that language.

In general, records in Modula-2 become *struct*s in `gdb`, and enumerations are represented internally as *C* unions with value and ordinal fields. Variant records in Modula-2 are displayed as a single structure by `gdb`, but a structure in which the fields belonging to different variants will overlap. Unravelling such structures is a relatively advanced topic.

Pointer types are displayed by `gdb` as "*TypeName* `*`" where, as mentioned previously, the asterisk means "pointer to". Remembering that *VAR* parameters are references to the actual parameter, variable parameters of pointer types will appear as "*TypeName* `**`", that is pointer to pointer to the type.

Array types are displayed with the number of elements shown in square brackets. `Gdb` is able to handle arrays which index from minimum values other than 0. However, open array parameters are always accessed through a pointer, since the stack offset of such values cannot be determined at compile time.

Subranges are not distinguished by `gdb` except for size. Thus a 1-byte sized subrange of *INTEGER* is reported by `gdb` as being of 8-bit integer type *Int8*. In the case of subranges of non-numeric types, `gdb` reports the host-type of the subrange.

Opaque types, as seen in the example above, are treated as pointers to language *C* char. If the target type is elaborated in the implementation module, and the implementation has been compiled with the g-flag, then we may force `gdb` to treat the opaque value as a pointer to the actual target type. The syntax for this is that of the traditional typecasts of Modula-2, now deprecated by **gpm**. In this form, the typename is used as the name of a *type transfer function*, with the value to be typecast as the single argument to the function.

For example if we have imported the opaque type *Tree*, which is elaborated as a pointer to a type *TreeBlock* which contains *key, left* and *right* fields, then we may force `gdb` to print the root node of the tree by the command —

```
(gdb) whatis tree
type = C_char *
(gdb) print TreeBlock(tree^)
$1 = {key = 25, left = 0x29e84, right = 0x29e70}
(gdb)
```

Notice in this case that we cast the *pointed-to value* to the bound type of the pointer. Once we have cast the opaque type, we may freely browse the rest of the tree —

```
    (gdb) whatis TreeBlock(tree^).left
    type = TreeBlock *
    (gdb) print TreeBlock(tree^).right^
    $4 = {key = 37, left = 0x29eac, right = 0x0}
(gdb)
```

These expressions are not strictly legal Modula-2 syntax, as we are being permitted to perform further selection on a value resulting from a "function call". In effect, we have *C* semantics overlaid onto the Modula-2 syntax.

A.5 Finding out more about gdb

`Gdb` has its own extensive documentation, available with the '`help`' command. *UNIX* systems typically have documentation available using the command '`man gdb`'. This introduction has scarcely scratched the surface of its capabilities. Learn to use the basic commands set out here, and then gradually add to your repetoire.

There is a short summary of the most common commands which have been found to be useful with **gpm**, called **Using gdb with gpm**. Any peculiarities which relate to using `gdb` with Modula-2 are highlighted in that document.

There is also a WWW site with full hypertext `gdb` manuals online, at

`http://www.cygnus.com/doc/gdb/index.html`

Appendix B

Using dbx to obtain a stack unwind listing

Program termination and the coredump

When a program aborts with a runtime error, the runtime system will attempt to create a memory *coredump* in a file in the current directory called **core**. The debugger utility **dbx** may be used to analyse this file, in order to locate the position in the program in which the fault occurred. The use of a coredump file from an aborted program to diagnose the cause of the exception is called *post-mortem debugging.*

The program **dbx** is capable of performing quite detailed analysis on coredump files. It may also be used as a *runtime debugger*, allowing programs to be executed under the control of **dbx** with the setting and clearing of breakpoints. An introduction to using **dbx** in this way appears in another appendix. In most cases the more detailed information is only available if the source file has been compiled with special options. However, the simplest, and most basic analysis is available to all programs, provided the object file has not been passed through the *strip* utility (the *strip* utility removes from the object file the symbol table information which is needed to report procedure names). The information which *is* available for all non-stripped programs is the *stack unwind list.* This list gives the sequence of procedure calls which gave rise to the program abort. For this reason the information is also called the *procedure call chain.*

The core dump file may be thought of as a *snapshot* of the data of the program at the instant that the exception occurred. One of the important data structures of the program is the procedure call chain which is held on the runtime stack. This information in on a stack so that as the procedures return in last-in-first-out order, the chain of procedure calls may be retraced. The runtime stack also holds procedure local variables and actual parameters. By "unwinding" the sequence of procedure activation records from the runtime stack, as frozen in the coredump, the procedure call chain may be reconstructed.

Example

Consider the following program.

```
MODULE Crash;

  VAR array : ARRAY [0 .. 2] OF CHAR;

  PROCEDURE RecurseUntilDead(x : CARDINAL);
  BEGIN
    array[x] := 0C;
    RecurseUntilDead(x + 1);
  END RecurseUntilDead;

BEGIN
  RecurseUntilDead(0);
END Crash.
```

This program, as the name of the procedure hints, recurses until finally the value of the formal parameter x causes an index bounds violation in the access to the array *array*.

When this program is compiled and then executed, a runtime error results.

```
$ gpm -g crash.mod
$ build crash
$ crash
**** m2rts: index error: 3 > 2 ****
abort - core dumped
$
```

The message indicates that the upper bound of the array index has been exceeded by one, and that a core-dump file has been produced.

The program **dbx** may now be started.

```
$ dbx crash core
```

dbx begins by printing information regarding the invocation and finally gives the user input prompt, which in this case is the string "(dbx)"

```
$ dbx crash core
dbx version 1.31
Type 'help' for help.
Corefile produced from file "crash"
reading symbolic information ...
[using memory image in core]
(dbx)
```

In this appendix the only **dbx** commands described are `where`, which produces the stack unwind list, and `quit` which exits the program. There are of course many, many other commands, and these are described in detail in the documentation which came with your system.

The stack unwind listing is obtained by typing 'where' at the command prompt. In the example this produces the following result.

```
(dbx) where
>  0 abort.abort(0x0, 0x0, 0x0, 0x0, 0x400be8) ["setjmp.s":113, 0x401eec]
   1 _catcher(0x3, 0x2, 0x0, 0x0, 0x40114c) ["m2rts.c":420, 0x400f0c]
   2 _ixChk(0x2, 0x0, 0x0, 0x2, 0x401168) ["m2rts.c":318, 0x400be4]
   3 RecurseUntilDead(x = 3) ["crash.mod":7, 0x401148]
   4 RecurseUntilDead(x = 2) ["crash.mod":8, 0x401164]
   5 RecurseUntilDead(x = 1) ["crash.mod":8, 0x401164]
   6 RecurseUntilDead(x = 0) ["crash.mod":8, 0x401164]
   7 StartCrash() ["crash.mod":12, 0x40118c]
   8 main(0x7fffeec4, 0x0, 0x0, 0x0, 0x0) ["/tmp/bld6353.c":25, 0x4002a4]
(dbx)
```

In this particular example the program was compiled with the **−g** option, and has additional information shown which would not otherwise be present. Procedures from the file compiled with the **−g** option have the the parameters listed in symbolic form. The other routines in the trace only have the parameter register values shown.

The program is terminated by typing in the quit command 'quit'.

Reading the procedure call chain

The procedure call chain is printed by **dbx** in the order in which the procedure activation records are unwound from the stack, that is, last-in-first-out. Each entry is in the format

called-procedure (*parameters*) [*filename* : *line-num* , *program-counter-value*]

The list in the example shows that the last procedure called was the system procedure *abort*, which causes the coredump to be produced. *abort* was called from *_catcher*, an internal procedure of the exception handler in the **gpm** runtime system. *_catcher* is the procedure which catches all exceptions, trapping back to the valid exception handler if there is one, or printing a message and calling *abort* in the absence of a handler.

In the example, *_catcher* was called from *_ixChk*, the index range check procedure, which in turn was called by the user's procedure *RecurseUntilDead*. The recursive calls of *RecurseUntilDead* were launched by the main line of module *Crash*. The call to the main module of every program always has an entry point which is known to the linker by the synthetic name `Start`*module*. In the event that a user defined symbol has an internal name which clashes with this synthetic name, the internal name is changed by adding a prefix.

The procedure *main* occurs in every call chain. *main* is the name of the function created by the load builder **build**, which constructs the module initialization calls. If it should happen that a program crashes during the initialization of an imported module body, then this is recognizable because the symbol called from *main* has the synthetic name `Init`*module*. The procedure *main* consists of *initialization* calls to the bodies of all imported modules whose bodies are not empty, using the module name prefixed by `Init` as the entry symbol. Procedure *main* finishes with a call to the body of the *program module*, which has an entry symbol formed from the module name with the prefix `Start`.

Notice how the line numbers in the procedure call chain correspond to the line numbers in the program listing. The line numbers and program counter values in the chain correspond to the *saved* values, that is, the values at the point where control passes to the next procedure. Thus in the example the main body calls the first invocation of `RecurseUntilDead` from line 12 of the file *crash.mod*. The recursive calls of the procedure come from line 8, and the call to `_ixChk` is hidden in line 7.

Errors and difficulties

The only significant difficulty which arises in practice, occurs when a bad memory reference has actually destroyed the procedure call chain. This sometimes occurs when a bad pointer value has been accessed, and also when an array argument on the stack has been accessed out of bounds. If this should occur, **dbx** will announce that it cannot unwind the stack data structure in the coredump.

Although the example shown here used the **–g** option, it is quite satisfactory to produce stack unwind traces without this. Information on the parameters of each call are much less helpful, but otherwise all information shown in the stack unwind trace itself is unchanged. In particular the file line numbers are still present. It is as well to know that use of the **–g** option is incompatible with many code optimizations, since it requires variables to be written to memory, rather than to be held in registers. Use of this option thus causes a significant decrease in execution speed of some object codes.

Appendix C

Getting started with dbx

gpm is capable of working well with **dbx** in the runtime debugger mode. This appendix sets out just enough information to get users started. For a more detailed explanation, and for the more advanced features of **dbx** section 5 of the system manuals should be carefully studied.

The following sections trace a **dbx** session, using a binary tree program to demonstrate concepts. The complete code of the program is shown at the end of this appendix.

Notes : All lines in the source code should contain one statement. If more than one statement is present on any line, results of single line stepping through the source may be difficult to interpret.

In all of the following, shortened forms of commands are used. The full form of the commands, and their corresponding abbreviations are shown in a table at the end of the section.

The example program

The example program consists of two modules. *BinTree* is an implmentation of a simple *table* abstract data type (*ADT*) as a binary tree. This Module has a a matching pair of definition and implementation parts. The module *TestBinTree* is a simple driver program to test the other module.

Both modules should be compiled using the **gpm –g** option. This option ensures that procedure parameters may be displayed in symbolic form, rather than as uninterpreted values.

To compile the program the following commands are typed at the shell prompt —

```
$ gpm bintree.def
$ gpm -g bintree.mod
$ gpm -g testbint.mod
$ build testbint
```

where the arguments are the names of the files.

Running the program

To now run the program under the control of **dbx** the following commmand is entered —

```
$ dbx testbint
```

The following lines will be displayed —

```
dbx version 1.31
Type 'help' for help
reading symbolic information...
main:    Source not available
(dbx) _
```

The note that source is not available will occur with all programs compiled with **gpm** . This is because the procedure *main* is a dummy procedure temporarily created by **build** to hold the module initialization call chain. It is automatically deleted by **build** unless you specify the persistent option. In any case, this file is never needed since it contains no user-written code.

Commands to **dbx** may now be entered at the "`(dbx)`" prompt. A complete list of commands may be displayed by typing 'help' or '?'.

Recording input and output

A log of input and output from **dbx** may be obtained by using the record commands, which specify files to store each log.

```
(dbx) ri input.dbx
[2] record input input.dbx (0 lines)
(dbx) ro output.dbx
[3] record output output.dbx (0 lines)
```

Setting an initial breakpoint

Breakpoints are positions in the program at which execution will be halted by **dbx** . Breakpoints may be specified in terms of line numbers or particular procedure names. In this example a command is shown which specifies the filename and line-number at which the breakpoint is to be inserted.

```
(dbx) stop at "testbint.mod":13
[4] stop at "testbint.mod":13
```

The user types the first line, at the prompt. The system replies with the second line verifying that the breakpoint has been entered. The example command inserts a breakpoint at line 13 of file `testbint.mod`.

Starting execution

The program may now be started with the run command. If the program requires command line arguments these are entered on the same line, following '`r`'.

```
(dbx) r
[4] stopped at [StartTestBinTr:13 ,0x4023e4] Insert(bt1,arr[i]);
```

dbx announces that it has stopped at a breakpoint. Note that **dbx** displays the module name and the hexadecimal contents of the program counter register, followed by the text of the source at line 13.

To see what has happened so far, a stack unwind listing may be obtained by typing `where` or the short form 't'.

```
(dbx) t
>  0 StartTestBinTr() ["testbint.mod":13, 0x4023e4]
   1 main(0x7fffee94, 0x0, 0x0, 0x0, 0x0) ["/tmp/bld1597.c":25, 0x4002c4]
```

The arrow at level 0 shows the 'activation level' of the stack, that is, the active scope or stack frame. This is usually a procedure. The most recently called procedure is numbered zero. Procedure *main* appears in every program. It is created by **build** and consists of initialization calls to the bodies of all imported modules. In this case there are no such calls, because *BinTree* defines an *ADT* and not some object which requires initialization of its hidden state information.

main finishes by calling the program module body which has the name "Start*module-name*".

Setting another breakpoint

Now to set another breakpoint — this time in a procedure.

```
(dbx) stop in Insert_BinTree
[5] stop in Insert_BinTree
```

Note the difference between "`stop at`" (or 'b') which stops at a line, and "`stop in`" or 'bp' which stops in a specified procedure. **gpm** names exported program procedures by concatenating the module name and the procedure name.

Continuing after a breakpoint

To continue execution type '`continue`' or 'c'.

```
(dbx) c
[5] stopped at [Insert_BinTree:19 ,0x40223c] NEW (root);
(dbx) t
>  0 Insert_BinTree(root = 0x10001df4, ch = 116) ["bintree.mod":19, 0x40223c]
   1 StartTestBinTr() ["testbint.mod":13, 0x4023f8]
   2 main(0x7fffee94, 0x0, 0x0, 0x0, 0x0) ["/tmp/bld1597.c":25, 0x4002c4]
```

The use of the stack trace command 't' shows the current activation levels. Note the parameters of `Insert_BinTree`. *root* is displayed as a hexadecimal value, while *ch* is displayed in decimal, in this case the *ASCII* code for 't'. This is because *root* is a pointer while *ch* is a character.

The type of any procedure or variable may be determined by using the '`whatis`' command followed by the variable name. Unfortunately the information is displayed as a language C type and so is of limited use when debugging a MODULA-2 program.

Removing a breakpoint

Suppose we wish to take out a breakpoint. This may be accomplished by the following sequence — which first examines the status of the session, and then asks for the removal of the effect of the numbered command using the `delete` command.

```
(dbx) status
[4] stop at "testbint.mod":13
[5] stop in Insert_BinTree
[2] record input input.dbx (8 lines)
[3] record output output.dbx (13 lines)
(dbx) delete 4
(dbx) status
[5] stop in Insert_BinTree
[2] record input input.dbx (10 lines)
[3] record output output.dbx (16 lines)
```

The breakpoint '`stop at "testbint.mod":13`' which had the status number 4 has been deleted, so that **dbx** will no longer trap the program at that point.

Stepping through the program execution

To continue, then print out the information about current procedure:

```
(dbx) c
[5] stopped at [Insert_BinTree:19 ,0x40223c] NEW (root);
(dbx) dump
Insert_BinTree(root = 0x10005004, ch = 106) ["bintree.mod":19, 0x40223c]
```

Note that this procedure has been called but it has yet to be executed. To execute on a line by line basis use '`step` n' where n is the number of lines to be executed. The default is 1. The short form of the step command is 's'.

A line is displayed before it is executed, so to execute a series of lines 's' must be typed after the line that we wish to execute is displayed. This means that an additional line will be displayed as below —

```
(dbx) s
[Insert_BinTree:20 ,0x402248] root^.info := ch;
(dbx) s
[Insert_BinTree:21 ,0x40225c] root^.left := NIL;
(dbx) s
[Insert_BinTree:22 ,0x402270] root^.right := NIL;
(dbx) s
[Insert_BinTree:23 ,0x402284] ELSIF ORD (ch) < ORD (root^.info) THEN
```

Examining variables

To print out the value of a variable the '`print`' or '`p`' command is used. Note that **dbx** understands Modula's selection operators for indexing, field selection and pointer dereference, { `'[ ]'`, `'.'`, `'^'`}, to be used[1]. The next command shows the use of the dereference operator to select a pointer target datum.

```
(dbx) p root^.info
106
```

The specified field contained the value 106_10 Variables in scopes outside the current activation level are able to be displayed provided their names are not *occluded* by a more local variable with the same name. For example the loop counter i in *StartTestBinTr*() may be printed as shown.

```
(dbx) p i
2
```

To see how i changes, the loop is continued 4 times —

```
(dbx) c
[5] stopped at [Insert_BinTree:19 ,0x40223c] NEW (root);
(dbx) c
[5] stopped at [Insert_BinTree:19 ,0x40223c] NEW (root);
(dbx) c
[5] stopped at [Insert_BinTree:19 ,0x40223c] NEW (root);
(dbx) c
[5] stopped at [Insert_BinTree:19 ,0x40223c] NEW (root);
```

The stack trace now shows the recursion clearly.

```
(dbx) t
>  0 Insert_BinTree(root = 0x10005034, ch = 103) ["bintree.mod":19, 0x40223c]
   1 Insert_BinTree(root = 0x10005014, ch = 103) ["bintree.mod":24, 0x4022c0]
   2 Insert_BinTree(root = 0x10005004, ch = 103) ["bintree.mod":24, 0x4022c0]
   3 Insert_BinTree(root = 0x10001df4, ch = 103) ["bintree.mod":24, 0x4022c0]
   4 StartTestBinTr() ["testbint.mod":13, 0x4023f8]
   5 main(0x7fffee94, 0x0, 0x0, 0x0, 0x0) ["/tmp/bld1597.c":25, 0x4002c4]
(dbx) p i
6
```

As expected, i is now $2 + 4 = 6$

Any variable may be traced using the command '`trace` *varname*', where *varname* is the name of the variable. In this case **dbx** informs the user at which point the variable changed values, and prints its old and new values. However a variable cannot be traced beyond the

[1] beware however of the difficulties caused by variant records (which have synthetic union field names), and the normalization of arrays so as to index from 0 as expected by languge C

end of a program, and to obtain a normal termination, the trace must be deleted from the status list before the termination of a program.

Suppose we now wish to trace the behaviour of the recursive procedure `Insert_BinTree()`. Having the call chain displayed, we may change the activation level to 3 using the '`func`' command:

```
(dbx) func 3
Insert_BinTree:  24  Insert(root^.left, ch);
```

If we were to now type 't' we would find the arrow pointing at activation level 3. We may now print the values of this instance of `Insert_BinTree` —

```
(dbx) p root^.info
116
```

Since 116 ¿ 103 (the value of *ch*) we would expect that *ch* would be inserted into the left subtree of the tree.

```
(dbx) p root^.left
0x10005010
```

Since this value is not *nil*, there is a non-empty left subtree. To examine this node we may go down one activation level (that is, toward the top of the stack.

```
(dbx) down
Insert_BinTree:  24  Insert(root^.left, ch);
```

The activation level is now 2. To check that this subtree is indeed the left branch of the subtree we observed at level 3 we may print the value of root. Note that we cannot determine the value of root from the parameter on the stack because it is a *VAR* parameter, and hence is a reference to the value. We may thus use the dereference operator to display the value.

```
(dbx) p root^
0x10005010
```

This confirms parameter is unchanged across procedure call.

Traversing the data structure

We could print out the values at this activation level, but we do not need to do so in order to traverse the tree structure —

```
(dbx) p root^.info
106
(dbx) p root^.left^.info
105
```

Since *ch* is still less than the value held in the tree we must descend another level in the tree. The activation level should be one, so we obtain call chain to check this.

```
(dbx) down
Insert_BinTree:  24  Insert(root^.left, ch);
(dbx) t
   0 Insert_BinTree(root = 0x10005034, ch = 103) ["bintree.mod":19, 0x40223c]
>  1 Insert_BinTree(root = 0x10005014, ch = 103) ["bintree.mod":24, 0x4022c0]
   2 Insert_BinTree(root = 0x10005004, ch = 103) ["bintree.mod":24, 0x4022c0]
   3 Insert_BinTree(root = 0x10001df4, ch = 103) ["bintree.mod":24, 0x4022c0]
   4 StartTestBinTr() ["testbint.mod":13, 0x4023f8]
   5 main(0x7fffee94, 0x0, 0x0, 0x0, 0x0) ["/tmp/bld1597.c":25, 0x4002c4]
```

Since there is only one more activation level, we would expect that there is no left subtree at this point. That is, `root^.left` should have the value *nil*.

```
    (dbx) p root^.left
    (nil)
```

Tracing calls and returns

We may now go down and single step through the sequence that sets up a leaf node:

```
(dbx) down
Insert_BinTree:  19  NEW (root);
(dbx) s
[Insert_BinTree:20 ,0x402248] root^.info := ch;
(dbx) s
[Insert_BinTree:21 ,0x40225c] root^.left := NIL;
(dbx) s
[Insert_BinTree:22 ,0x402270] root^.right := NIL;
(dbx) s
[Insert_BinTree:23 ,0x402284] ELSIF ORD (ch) < ORD (root^.info) THEN
```

Since *NEW* has assigned a new value to root at level 0, this value should also be the value of the activation level 1. To see if this is the case, first print the value of root at level 0, then continue the single stepping until activation level 1 becomes activation level 0, that is, until procedure return is made and the previous stack frame is discarded.

```
    (dbx) p root^
    0x10005050
```

now check the call chain to determine parameter of level 1

```
(dbx) up
Insert_BinTree:  24  Insert(root^.left, ch);
(dbx) t
   0 Insert_BinTree(root = 0x10005034, ch = 103) ["bintree.mod":23, 0x402284]
>  1 Insert_BinTree(root = 0x10005014, ch = 103) ["bintree.mod":24, 0x4022c0]
   2 Insert_BinTree(root = 0x10005004, ch = 103) ["bintree.mod":24, 0x4022c0]
```

```
   3 Insert_BinTree(root = 0x10001df4, ch = 103) ["bintree.mod":24, 0x4022c0]
   4 StartTestBinTr() ["testbint.mod":13, 0x4023f8]
   5 main(0x7fffee94, 0x0, 0x0, 0x0, 0x0) ["/tmp/bld1597.c":25, 0x4002c4]
```

We now go back down and single step to the end of the procedure, then step into the first line of the next procedure and check the call chain.

```
(dbx) down
Insert_BinTree:  23  ELSIF ORD (ch) < ORD (root^.info) THEN
(dbx) s
[Insert_BinTree:27 ,0x402300] END;
(dbx) s
[Insert_BinTree:25 ,0x4022c4] ELSIF ORD (ch) # ORD (root^.info) THEN
(dbx) t
>  0 Insert_BinTree(root = 0x10005014, ch = 103) ["bintree.mod":25, 0x4022c4]
   1 Insert_BinTree(root = 0x10005004, ch = 103) ["bintree.mod":24, 0x4022c0]
   2 Insert_BinTree(root = 0x10001df4, ch = 103) ["bintree.mod":24, 0x4022c0]
   3 StartTestBinTr() ["testbint.mod":13, 0x4023f8]
   4 main(0x7fffee94, 0x0, 0x0, 0x0, 0x0) ["/tmp/bld1597.c":25, 0x4002c4]
```

As may be seen, the stack has been cut back by one frame. Note that to step across a procedure call or return we use the 's' command, whereas if we wished to avoid stepping into new procedures the command '`next`' or 'n' should be used.

We may now check the value of the root of the left subtree.

```
(dbx) p root^.left
0x10005050
```

It is expected value.

To run the program to the end we may remove the breakpoint and issue the continue command.

```
(dbx) status
[5] stop in Insert_BinTree
[2] record input input.dbx (48 lines)
[3] record output output.dbx (75 lines)
(dbx) delete 5
```

Note that the program output is displayed on screen interspersed with **dbx** output.

```
(dbx) c

g i j o q r t w y

Program terminated normally
```

The first line is the program output demonstrating that the information is in an ordered sequence.

Quitting dbx

To exit dbx type '`quit`' or 'q'.

```
(dbx)q

$ _
```

We are now back at the unix shell prompt.

Table of commands used in this appendix

Command	**Alias**	**Description of command**
record input	ri	records command input to nominated file
record output	ro	records **dbx** output to nominated file
stop at	b	set a breakpoint in the code produced from the nominated file at the nominated line
run	r	start execution of program
where	t	display trace of activation frames
stop in	bp	set a breakpoint at start of procedure
continue	c	continue execution after a breakpoint
status		display status information
delete	d	deletes specified status item
dump		displays variable information for the active procedure
step	s	execute a line of source text
next	n	execute a line in the current procedure
print	p	prints the value of the specified designator
func	f	moves activation to specified level on the stack
down		moves activation down one level on the stack
up		moves activation up one level on the stack
quit	q	terminates **dbx**

Listings of example program

```
DEFINITION MODULE BinTree;
TYPE BType;
PROCEDURE Create (VAR root: BType);
PROCEDURE Insert (VAR root: BType; ch: CHAR);
PROCEDURE Display (root: BType);
END BinTree.
```

```
MODULE TestBinTree;
FROM InOut IMPORT Write,Read,WriteString,WriteLn;
FROM BinTree IMPORT BType, Create, Insert, Display;

VAR bt1: BType;
    arr: ARRAY [1..10] OF CHAR;
    i  : INTEGER;

BEGIN
  arr := 'tjwiogqrty';
  Create (bt1);
  FOR i := 1 TO 10 DO
    Insert(bt1,arr[i]);
  END;
  Display(bt1);
  WriteLn;
END TestBinTree.
```

```
IMPLEMENTATION MODULE BinTree;
FROM Storage IMPORT ALLOCATE;
FROM InOut IMPORT Write;

TYPE BType = POINTER TO NodeType;
     NodeType = RECORD
                  info : CHAR;
                  left, right : BType;
               END;

PROCEDURE Create (VAR root: BType);
BEGIN
  root := NIL;
END Create;

PROCEDURE Insert (VAR root: BType; ch : CHAR);
BEGIN
  IF root = NIL THEN
    NEW (root);
    root^.info := ch;
    root^.left := NIL;
    root^.right := NIL;
  ELSIF ORD (ch) < ORD (root^.info) THEN
    Insert (root^.left, ch);
  ELSIF ORD (ch) # ORD (root^.info) THEN
    Insert (root^.right, ch);
  END;
END Insert;
```

```


PROCEDURE Display (root: BType);
BEGIN
  IF root # NIL THEN
    Display (root^.left);
    Write (root^.info);
    Write (' ');
    Display (root^.right);
  END;
END Display;

END BinTree.
```

Appendix D

Using XDB to obtain a stack unwind listing

Program Termination and the Coredump

When a program aborts with a runtime error, the runtime system will attempt to create a memory coredump in a file in the current directory called *core*. The debugger utility **xdb** may be used to analyse this file, in order to locate the position in the program where the fault occurred. The use of a coredump file from an aborted program to diagnose cause of the exception is called **post-mortem debugging**.

The program **xdb** is capable of performing quite detailed analysis on coredump files. It may also be used as a runtime debugger, allowing programs to be executed under the control of **xdb** with the setting and clearing of breakpoints. In most cases the more detailed information is only available if the source file has been compiled with special options. However, the simplest, and most basic analysis is available to all programs, provided the object file has not been passed through the **strip** utility. strip removes symbol file information from the file which is needed to report procedure names. The information which is available for all programs is the stack unwind list. This list gives the sequence of procedure calls which gave rise to the program abort. For this reason the information is also called the **procedure call chain**.

The core dump file may be thought of as a snapshot of the data of the program at the instant that the exception occurred. One of the important data structures of the program is the procedure call chain which is held on the runtime stack. This information is on a stack so that as the procedures return in last-in-first-out order, the chain of procedure calls may be retraced. The runtime stack also holds procedure local variables and actual parameters. By "unwinding" the sequence of procedure activation records from the runtime stack, as frozen in the coredump, the procedure call chain may be reconstructed.

Example

Consider the following program —

```
MODULE Crash;

```

```
  VAR array : ARRAY [0..2] OF CHAR;

PROCEDURE RecurseUntilDead(x : CARDINAL);
BEGIN
  array[x] := 0C;
  RecurseUntilDead(x + 1);
END RecurseUntilDead;

BEGIN
  RecurseUntilDead(0);
END Crash.
```

This program, as the name of the procedure hints, recurses until finally the value of the formal parameter x causes an index bounds violation in the access to the array *array*. When this program is compiled and then executed, a runtime error results.

```
$  gpm -g crash.mod
$  build crash
$  crash
**** m2rts:  index error:  3 > 2 ****
abort - core dumped
$
```

The message indicates that the upper bound of the array index has been exceeded by one, and that a core dump file has been produced. The program **xdb** may now be started.

```
$  xdb crash core
```

xdb begins by printing information regarding the invocation and finally gives the user input prompt, which in this case is '>'

```
$  xdb crash core
Copyright Hewlett-Packard Co. 1985.  All Rights Reserved.
<<<< XDB Version A.07.05 HP-UX >>>>
Procedures:  2
Files:  1
Child died due to:  IOT instruction.
(file unknown):  _raise +0x0000001f:  (line unknown)
>
```

In this appendix the only **xdb** commands described are **t**, which produces the stack unwind list, **T** which gives the stack unwind list with more information, and **q**uit which exits the program. There are of course many other commands which are described in detail in the documentation which came with your system.

As mentioned above, the stack unwind listing is obtained by typing 't' at the command prompt. In the example this produces the following result:

```
> t
 0  _raise + 0x0000001f (0, 0, 0, 0x40012f66)
 1  _abort + 0x0000002c (0, 0, 0, 0)
 2  _catcher + 0x00000054 (0, 0, 0, 0)
 3  _ixChk + 0x0000004c (0, 0, 0, 0)
 4  Crash_RecurseUntilDead (x = 3)    [crash.mod:  7]
 5  Crash_RecurseUntilDead (x = 2)    [crash.mod:  9]
 6  Crash_RecurseUntilDead (x = 1)    [crash.mod:  9]
 7  Crash_RecurseUntilDead (x = 0)    [crash.mod:  9]
 8  StartCrash ()    [crash.mod:  13]
 9  main + 0x00000028 (0, 0, 0, 0)
10  _start + 0x000000068 (0, 0, 0, 0)
```

In this particular example the program was compiled with the –g option and has additional information shown which would not otherwise be present. Procedures from the file compiled with the –g option have the parameters listed in symbolic form. The other routines in the trace only have the parameter register values shown.

If the **xdb** command **T** is used, the same information is produced, but any local variables in the procedures are shown. To demonstrate this two local variables i and j are added to the crash program, so that the code now appears as:

```
MODULE Crash;

  VAR array : ARRAY [0..2] OF CHAR;

PROCEDURE RecurseUntilDead(x : CARDINAL);
VAR i,j : INTEGER;
BEGIN
  i := 3;
  j := 4;
  array[x] := 0C;
  RecurseUntilDead(x + 1);
END RecurseUntilDead;

BEGIN
  RecurseUntilDead(0);
END Crash.
```

If the program is then recompiled as before and **xdb** is invoked and the T command is used the following is produced —

```
> T
 0  _raise + 0x0000001f (0, 0, 0, 0x40012f66)
 1  _abort + 0x0000002c (0, 0, 0, 0)
 2  _catcher + 0x00000054 (0, 0, 0, 0)
 3  _ixChk + 0x0000004c (0, 0, 0, 0)
 4  Crash_RecurseUntilDead (x = 3)    [crash.mod:  10]
```

```
                  i    =    3
                  j    =    4
    5  Crash_RecurseUntilDead (x = 2)    [crash.mod:  12]
                  i    =    3
                  j    =    4
    6  Crash_RecurseUntilDead (x = 1)    [crash.mod:  12]
                  i    =    3
                  j    =    4
    7  Crash_RecurseUntilDead (x = 0)    [crash.mod:  12]
                  i    =    3
                  j    =    4
    8  StartCrash ()    [crash.mod:  16]
    9  main + 0x00000028 (0, 0, 0, 0)
   10  _start + 0x000000068 (0, 0, 0, 0)
```

The **xdb** program is terminated by typing in the quit command **q**.

Reading the Procedure Call Chain

The procedure call chain is printed by **xdb** in the order in which the procedure activation records are unwound from the stack, that is, last-in-first-out. Each entry is in the format

called-procedure (parameters) [filename: line-num]
local variable = value[1]

The list in the example shows that the last procedure called was the system procedure *_raise*. *_raise* was called from *_abort* which produces the coredump. *_abort* was called from *_catcher*, an internal procedure of the exception handler in the **gpm** runtime system. *_catcher* is the procedure which catches all exceptions, trapping back to the valid exception handler if there is one, or printing a message and calling *_abort* in the absence of a handler.

In the example, *_catcher* was called from *_ixChk*, the index range check procedure, which in turn was called by the user's procedure *Crash_RecurseUntilDead*. The recursive calls of *RecurseUntilDead* were launched by the main line of module *Crash*. The call to the main module of every program always has an entry point which is known to the linker by the synthetic name Start*module*. In the event that a user defined symbol has an internal name which clashes with this synthetic name, the internal name is changed by adding a prefix.

The procedure main occurs in every call chain. main is the name of the function created by the load builder build, which constructs the module initialization calls. If it should happen that a program crashes during the initialization of an imported module body, then this is recognizable because the symbol called from main has the synthetic name Initmodule. The procedure main consists of initialization calls to the bodies of all imported modules which bodies are not empty, using the module name prefixed by `Init` as the entry symbol. Procedure main finishes with a call to the body of the program module, which has an entry symbol formed from the module name with the prefix `Start`.

[1] only for **T** command

The line numbers in the procedure call chain correspond to the line after the procedure call is made from the program. That is, the line number which is saved is the line where execution must begin when the calling procedure is reactivated. In the example above the first call to *RecurseUntilDead* is on line 12 of the program. However, line number 13 is saved because when the *StartCrash* code begins execution after return from the procedure call on line 12, execution will start at line 13. Similarly, the recursive calls to *RecurseUntilDead* all occur on line 8 so line number 9 is stored. The call to *_ixChk* is different in that there will be no return from this procedure and the call to it is hidden on line 7.

Errors and Difficulties

The only significant difficulty which arises in practice, occurs when a bad memory reference has actually destroyed the procedure call chain. This sometimes occurs when a bad pointer value has been accessed, and also when an array argument on the stack has been accessed out of bounds. If this should occur, **xdb** will announce that it cannot unwind the stack data structure in the coredump.

Although the example shown here used the –g option, it is quite satisfactory to produce stack unwind traces without this. Information on the parameters of each call are much less helpful, but otherwise all information shown in the stack unwind trace itself is unchanged. In particular the file line numbers are still present. It is as well to know that the use of the –g option is incompatible with many code optimizations since it requires variables to be written to memory, rather than to be held in registers. Use of this option thus causes significant decrease in execution speed of some object codes.

Appendix E

Using adb to obtain a stack unwind listing (HP-UX)

Program termination and the coredump

When a program aborts with a runtime error, the runtime system will attempt to create a memory *coredump* in a file in the current directory called **core**. On HP-UX systems the standard *UNIX* utility **adb** may be used to analyse this file, in order to locate the position in the program in which the fault occurred. The use of a coredump file from an aborted program to diagnose the cause of the exception is called *post-mortem debugging*.

The program **adb** is capable of performing quite detailed analysis on coredump files. In most cases the more detailed information is only available if the source file has been compiled with the **–g** option. However, the simplest, and most basic analysis is available to all programs, provided the object file has not been passed through the *strip* utility. The *strip* utility removes symbol file information from the file which is needed to report procedure names. The information which is available for all programs is the *stack unwind list*. This list gives the sequence of procedure calls which gave rise to the program abort. For this reason the information is also called the *procedure call chain*.

The core dump file may be thought of as a *snapshot* of the data of the program at the instant that the exception occurred. One of the important data structures of the program is the procedure call chain which is held on the runtime stack. This information in on a stack so that as the procedures return in last-in-first-out order, the chain of procedure calls may be retraced. The runtime stack also holds procedure local variables and actual parameters. By "unwinding" the sequence of procedure activation records from the runtime stack, as frozen in the coredump, the procedure call chain may be reconstructed.

Example

Consider the following program.

```
MODULE Crash;

  VAR array : ARRAY [0 .. 2] OF CHAR;

  PROCEDURE RecurseUntilDead(x : CARDINAL);
  BEGIN
    array[x] := 0C;
    RecurseUntilDead(x + 1);
  END RecurseUntilDead;

BEGIN
  RecurseUntilDead(0);
END Crash.
```

This program, as the name of the procedure hints, recurses until finally the value of the formal parameter x causes an index bounds violation in the access to the array *array*.

When this program is compiled and then executed, a runtime error results.

```
$ gpm crash.mod
$ build crash
$ crash
**** m2rts: index error: 3 > 2 ****
abort - core dumped
$
```

The message indicates that the upper bound of the array index has been exceeded by one, and that a core-dump file has been produced.

The program **adb** may now be started.

```
$ adb crash
```

The program actually takes a second argument, but substitutes the default (`core` from the current directory) if no argument is supplied. The program does not give any prompts or indications, but waits for a command to be entered. The command to produce a stack unwind listing is '$c'. Note that unlike the above examples, the dollar character is not the system prompt, but is entered by the user

```
$c
abort()    from _catcher+30
_catcher() from _ixChk+3C
_ixChk() from RecurseUntilDead+24
RecurseUntilDead() from RecurseUntilDead+34
RecurseUntilDead() from RecurseUntilDead+34
RecurseUntilDead() from RecurseUntilDead+34
RecurseUntilDead() from StartCrash+10
```

```
StartCrash() from main+28
main() from _start+14
_start() from $START$+30
```

adb then waits for another command.

The program is terminated by typing in the quit command '$q'.

Reading the procedure call chain

The procedure call chain is printed by **adb** in the order in which the procedure activation records are unwound from the stack, that is, last-in-first-out. Each entry is in the format

called-procedure '`() from `' *calling-procedure* '`+`' *code-offset*

The list in the example shows that the last procedure called was the system procedure *abort*, which causes the coredump to be produced. *abort* was called from *_catcher*, an internal procedure of the exception handler in the **gpm** runtime system. *_catcher* is the procedure which catches all exceptions, trapping back to the valid exception handler if there is one, or printing a message and calling *abort* in the absence of a handler.

In the example, *_catcher* was called from *_ixChk*, the index range check procedure, which in turn was called by the user's procedure *RecurseUntilDead*. The recursive calls of *RecurseUntilDead* were launched by the main line of module *Crash*. The call to the main module of every program always has an entry point which is known to the linker by the synthetic name `Start`*module*. In the event that a user defined symbol has an internal name which clashes with this synthetic name, the internal name is changed by adding a prefix.

The procedure *main* occurs in every call chain. *main* is the name of the function created by the load builder **build**, which constructs the module initialization calls. If it should happen that a program crashes during the initialization of an imported module body, then this is recognizable because the symbol called from *main* has the synthetic name `Init`*module*. The procedure *main* consists of *initialization* calls to the bodies of all imported modules whose bodies are not empty, using the module name prefixed by `Init` as the entry symbol. Procedure *main* finishes with a call to the body of the *program module*, which has an entry symbol formed from the module name with the prefix `Start`.

Errors and difficulties

The only significant difficulty which arises in practice, occurs when a bad memory reference has actually destroyed the procedure call chain. This sometimes occurs when a bad pointer value has been accessed, and also when an array argument on the stack has been accessed out of bounds. If this should occur, *adb* will announce that it cannot unwind the stack data structure in the coredump.

It is as well to know that use of the **–g** compiler option is incompatible with many code optimizations, since it requires variables to be written to memory, rather than to be held in registers. Use of this option thus causes a significant decrease in execution speed of some object codes.

Appendix F

Using the Profiling Tools

(mips-architecture machines)

One of the most important tools in the tuning of application programs for maximum speed is a *runtime profiler*. Using this tool it is possible to find out exactly where the program is spending its time, so that attention may be given to seeking improvements to the code in those parts where significant benefit may be obtained.

There are two kinds of information which are used in profiling: procedure call-counts, and percentage time analysis. Gardens point modula provides a simple-to-use interface to the standard *UNIX* tool **prof** to obtain both kinds of information. If a program is profiled to obtain call counts, the number of times each procedure was called during the execution of the program is produced. This information is always exact. Percentage time analysis seeks to find out what percentage of the total execution time is spent in each procedure of the program. Because of the method of measurement, this information is statistical in nature and is only accurate for very long-running programs.

Computers based on the **mips** architecture have a unique tool **pixie** available which produces profiling information which is much more detailed and accurate than the information provided by other systems. This chapter describes the way in which **gpm** interfaces with **pixie**.

F.1 Getting execution time percentages

If only the percentage time distribution is required, the modules of the program are compiled as usual, but are linked with a special version of the builder called **bldprf**. This version accepts exactly the same options as **build** but generates a linkage to the standard *UNIX* profiling system.

Example

Suppose it is wished to profile the well known *dhrystone* benchmark program. After compilation of the modules of the program the optional version of the builder is invoked and the program subsequently executed.

```
$ bldprf dhry
Circular imports, initialization order is
   <Dhry3> (empty body)
   <Dhry2> (empty body)
$ dhry
Benchmark running...
elapsed time : 26
machine benchmarks at 19056 dhrystones per second
$
```

The results of profiling may now be analyzed by the program **prof**, using the command `prof` *filename*. This first example of output shows the complete output including the header. Later examples have the headers deleted.

```
$ prof dhry

Profile listing generated Wed Apr 11 16:06:48 1990 with:
   prof dhry

----------------------------------------------------------------------------
*  -p[rocedures] using pc-sampling;                                        *
*  sorted in descending order by total time spent in each procedure;       *
*  unexecuted procedures excluded                                          *
----------------------------------------------------------------------------

Each sample covers 8.00 byte(s) for 0.015% of 68.7100 seconds

%time     seconds  cum %   cum sec  procedure (file)

 26.5     18.1800   26.5     18.18 StdStrings_Compare (stdstrin.mod)
 15.6     10.7500   42.1     28.93 strcpy (strcpy.s)
 13.4      9.2400   55.6     38.17 Dhry2_Proc1 (dhry2.mod)
 13.1      9.0300   68.7     47.20 Dhry2_Proc0 (dhry2.mod)
  7.1      4.8500   75.8     52.05 Dhry3_Func2 (dhry3.mod)
  5.7      3.9100   81.4     55.96 Dhry3_Proc6 (dhry3.mod)
  5.0      3.4200   86.4     59.38 Dhry3_Proc8 (dhry3.mod)
  3.2      2.2200   89.7     61.60 Dhry3_Func1 (dhry3.mod)
  2.7      1.8500   92.3     63.45 Proc4 (dhry2.mod)
  2.2      1.5400   94.6     64.99 Dhry2_Proc2 (dhry2.mod)
  2.2      1.5300   96.8     66.52 Dhry2_Proc3 (dhry2.mod)
  1.4      0.9300   98.2     67.45 Func3 (dhry3.mod)
  1.2      0.8300   99.4     68.28 Dhry3_Proc7 (dhry3.mod)
  0.6      0.4100  100.0     68.69 Proc5 (dhry2.mod)
  0.0      0.0100  100.0     68.70 write (write.s)
  0.0      0.0100  100.0     68.71 malloc (malloc.c)
```

This output shows that the dhrystone program spends about 25% of its time in the string comparison function *Compare* from module *StdStrings*, and another 16% in the string copy function *strcpy*. The other entries in the table are the various procedures of the program, listed by their linker names as generated by **gpm**.

The information provided by profiling is statistical in nature, and may vary from run to run. Nevertheless, for programs which run for a significant length of time the statistics will be fairly accurate. In this case there are only small variations in the percentages.

F.2 How profiling works

The profiler works by interrupting the program at every clock tick, and checking to see which procedure is currently being executed. Since these interrupts only occur fifty or sixty times per second of runtime, it is not possible to accurately profile programs which run for only a few seconds. In such cases it may be necessary to run your program repeatedly, and average the results of many executions.

Under unusual circumstances, where a program exhibits cyclic behavior which synchronizes with the clock ticks, it is possible to get *very* misleading results from using **prof**. These occurrences are rare, but should always be borne in mind if inexplicable results are encountered. If in doubt, repeat the experiment several times.

F.3 Basic-block counting (using pixie)

The program **pixie** allows profiles to be generated which show the number of actual processor cycles spent in each procedure of a program. It is also possible to count the calls of each procedure, and to find how many of these calls originate from any particular line in the program. Finally, it is possible to find out how many processor cycles are spent on each line of the program. This information is exact.

To obtain all this information, the program to be analyzed is compiled and linked in the normal way. No additional compiler options are used, and the normal builder program **build** is used. After the program is built, the program **pixie** is invoked.

```
$ pixie -o dhry.pixie dhry
pixie registers r31, r23, and r30
oldcode 31024 bytes, new code 88756 bytes (2.9x)
$
```

The program takes the executable file `dhry` and produces a new file `dhry.pixie`. This new file is executed to perform the profiling.

The new, "pixified" version of the file is usually about three times the size of the original. It executes correspondingly slower, producing output to two files `dhry.Addrs` and `dhry.Counts`.

```
$ dhry.pixie
Benchmark running...
elapsed time : 64
machine benchmarks at 7812 dhrystones per second
$
```

The timing of this execution should be ignored, since it includes all of **pixie's** processing. The true information is obtained by running the profiling program with the **−pixie** option.

Three options of **prof −pixie** are shown here. First, procedure cycle counts, are produced with the **−p**[**rocedure**] option. Only the first few lines are shown, and most of the header has been deleted —

```
Profile listing generated Wed Apr 11 16:13:02 1990 with:
   prof -pixie -proc dhry

353509999 cycles

    cycles %cycles  cum %     cycles  bytes procedure (file)
                               /call  /line

 104500000   29.56  29.56        209     15 StdStrings_Compare (stdstrin.mod)
  66000402   18.67  48.23        132      5 strcpy (strcpy.s)
  43000085   12.16  60.39   43000085     19 Dhry2_Proc0 (dhry2.mod)
  36000000   10.18  70.58         72     17 Dhry2_Proc1 (dhry2.mod)
  22500000    6.36  76.94         45     19 Dhry3_Proc8 (dhry3.mod)
  21500000    6.08  83.02         43     25 Dhry3_Func2 (dhry3.mod)
  13500000    3.82  86.84         27     22 Dhry3_Proc6 (dhry3.mod)
  12000000    3.39  90.24          8     12 Dhry3_Func1 (dhry3.mod)
   8500000    2.40  92.64         17     10 Dhry2_Proc2 (dhry2.mod)
   8000000    2.26  94.91         16     12 Dhry2_Proc3 (dhry2.mod)
   6000000    1.70  96.60          4      8 Dhry3_Proc7 (dhry3.mod)
   5500000    1.56  98.16         11     13 Func3 (dhry3.mod)
   4500000    1.27  99.43          9     12 Proc4 (dhry2.mod)
   ...
```

Note that the percentages are slightly different to those obtained by PC-sampling. It is speculated that this is due to memory caching not being taken into account in the counting of cycles.

Procedure invocation counts are obtained by using the **−i**[**nvocations**] option. The procedures in this case are ordered by number of calls. Note that in the 500 000 cycles of the dhrystone benchmark the function *Func1* is called three times per cycle: twice from procedure *Proc0* at line 141 of file `dhry2.mod`, and once from *Func2* at line 71 of file `dhry3.mod`.

```
Profile listing generated Wed Apr 11 16:15:22 1990 with:
   prof -pixie -i dhry

called procedure  #calls %calls  from line, calling procedure (file):

Dhry3_Func1      1000000  66.67        141  Dhry2_Proc0 (dhry2.mod)
                  500000  33.33         72  Dhry3_Func2 (dhry3.mod)
Dhry3_Proc7       500000  33.33         43  Dhry2_Proc3 (dhry2.mod)
                  500000  33.33         75  Dhry2_Proc1 (dhry2.mod)
                  500000  33.33        135  Dhry2_Proc0 (dhry2.mod)
```

```
strcpy             500000 100.00        130  Dhry2_Proc0 (dhry2.mod)
                        2   0.00         37  Terminal_WriteCard (terminal.c)
                        1   0.00        116  Dhry2_Proc0 (dhry2.mod)
                        1   0.00        118  Dhry2_Proc0 (dhry2.mod)
                        0   0.00         19  ProgArgs_VersionTime (/tmp/bld13
                        0   0.00         16  ProgArgs_EnvironString (/tmp/bld
                        0   0.00         12  ProgArgs_GetArg (/tmp/bld13294.c
Dhry2_Proc1        500000 100.00        139  Dhry2_Proc0 (dhry2.mod)
StdStrings_Compare  500000 100.00         79  Dhry3_Func2 (dhry3.mod)
Dhry2_Proc2        500000 100.00        146  Dhry2_Proc0 (dhry2.mod)
Dhry3_Proc6        500000 100.00         73  Dhry2_Proc1 (dhry2.mod)
...
```

It may be noticed that this profile shows those procedure calls which were never exercised. The program is thus useful for determining code coverage in program testing.

The final example given here is of the **l[ine]** option. This option shows the number of cycles spent on each line of source text. Once again only the first few lines have been shown. Things to note in the output from this example are the very high cycle counts in those parts of the *Compare* function which are inside the loop which compares characters of the two strings. In the dhrystone program, the strings which are compared are of length 20.

```
Profile listing generated Wed Apr 11 16:19:01 1990 with:
   prof -pixie -l dhry

procedure (file)                          line bytes     cycles %cycles

StdStrings_Compare (stdstrin.mod)          246    20    2500000    0.71
                                           250    16    1000000    0.28
                                           252    16    1500000    0.42
                                           253     8     500000    0.14
                                           255    12    1500000    0.42
                                           256    28    3500000    0.99
                                           258     8   19000000    5.37
                                           259    16   18000000    5.09
                                           260    20   36000000   10.18
                                           261    28   20000000    5.66
                                           284    12    1000000    0.28
strcpy (strcpy.s)                          104     4     500004    0.14
                                           110     4     500004    0.14
                                           111     4     500004    0.14
                                           112     4     500004    0.14
                                           113     4     500004    0.14
                                           114     4     500004    0.14
                                           115     4     500004    0.14
                                           116     4     500004    0.14
                                           117     4     500004    0.14
                                           118     4     500004    0.14
```

```
                                          119      4      500004    0.14
                                          120      4      500004    0.14
...
```

F.4 Summary

Percentage time analysis is of most use for programs which have substantial execution time, as an aid to locating the profitable areas for code improvement. Unless the runtime is substantial, or some averaging is performed between multiple executions, the statistics produced are of limited accuracy.

Basic block counting is a very useful technique for programs with complex control flow since the counts can reveal information which is otherwise non-obvious. For example call counting can determine the average number of calls to the *StdStrings.Compare* procedure which are made for each call of *Lookup* in a binary tree based implementation of a symbol table. At the cost of a single, much slower execution of the program, a wealth of information is obtained for detailed tuning of the final code.

A word of advice

It is a common and costly mistake to attempt to optimize programs too soon. The issues in software engineering are correctness, algorithmic elegance, and execution speed, in that order. Without correctness, programs are untrustworthy, a bad choice of algorithm can throw away orders of magnitude in execution speed. Only when these issues have been adequately addressed should attention be given to the last 10 or 20% of improvement in speed which the use of profiling promises.

Appendix G

Interpreting the stack unwind trace on gpm-pc

Program completion and abnormal termination

When a program aborts with a runtime error, the runtime system will attempt to return to DOS in an orderly fashion. Because of the unprotected nature of the hardware environment, there are possible circumstances in which this is impossible, and the machine may "hang-up" and require rebooting. The **gpm-pc** interpreter attempts to minimize the occurrence of such events.

Inside the runtime support system, all errors are channeled into a utility procedure `__catcher` which produces the error message and then calls another utility `_m2_abort`. Some errors detected in the interpreter call `_m2_abort` directly. This last procedure attempts to perform a diagnostic unwinding of the procedure stack before actually exiting the program.

When a typical runtime error occurs, a message of the following general form is sent to the standard error stream —

```
**** m2rts: assert error: <sqrt of negative value> ****
abnormal program termination
<foreign stack frame>
RealMath_sqrt   Line-num = ??, file [realmath.mod]
GetAndTestValue Line-num = 18, file [mathtest.mod]
StartMathTest   Line-num = 33, file [mathtest.mod]
```

The first two lines are produced by the runtime system, and give general information about the error, while the rest are produced by the stack unwinder.

Unwinding the stack

At any point in the execution of a program, there is a **dynamic call chain**. This chain is the ordered sequence of procedures which have been called but have not yet completed.

In conventional implementations of procedural langauges such as Modula, space for local variables is allocated on a **runtime stack**. Each procedure invocation in the dynamic call chain has space allocated for its local variables and parameters on the runtime stack in a data structure called a **stack frame**. In the case of procedures called recursively, every separate call of the procedure has its own stack frame.

When a program crashes, it is possible to find out information about the state of the program at the time of the crash by examining the information in the runtime stack. This is done by two main methods. Either the whole memory state of the machine may be dumped to a file for later analysis, or some helpful information may be extracted at the time. In the case of **gpm-pc**, the second approach is adopted. In *UNIX* versions of **gpm** the so-called *post-mortem* analysis of core-dump files is the normal method.

Consider the following example program —

```
MODULE Crash;

  VAR array : ARRAY [0 .. 2] OF CHAR;

  PROCEDURE RecurseUntilDead(x : CARDINAL);
  BEGIN
    array[x] := 0C;
    RecurseUntilDead(x + 1);
  END RecurseUntilDead;

BEGIN
  RecurseUntilDead(0);
END Crash.
```

This program, as the name of the procedure hints, recurses until finally the value of the formal parameter x causes an index bounds violation which causes the program to terminate. The sequence of events, when the program is run is as follows. Various procedures are called which initialize parts of the runtime system. These return, and finally the main module body is called. This "procedure" is always given the synthetic name `Start`*module-name*, so in this case the procedure is called `StartCrash`. The main module calls the procedure `RecurseUntilDead`, which in turn calls itself from the second line of its code. Finally, an array bounds violation occurs in the first line of `RecurseUntilDead`, terminating the program.

Running the program produces the following output —

```
C:\gpm\wrk> crash

**** m2rts: index error: 3 not in [0 .. 2] ****
abnormal program termination
<foreign stack frame>
RecurseUntilDead   Line-num = 7, file [crash.mod]
RecurseUntilDead   Line-num = 8, file [crash.mod]
RecurseUntilDead   Line-num = 8, file [crash.mod]
RecurseUntilDead   Line-num = 8, file [crash.mod]
StartCrash  Line-num = 12, file [crash.mod]
```

The top frame on the stack at the time of the crash is a runtime support procedure, and does not have any diagnostic information available. The next frame belongs to the final call of the procedure and terminated at line number 7 which just happens to be the first line of the procedure.

The three next frames are the recursive calls of the procedure with actual parameter values of 2,1 and 0. Note that in each case the procedure was at line number 8 when it suspended its execution by the next procedure call.

The final frame on the stack belongs to the module body. In this case the line number is 12, corresponding to the fact that the call instruction is the very first instruction in the procedure.

Note that any procedures which have been called and have returned are not listed in the stack unwind. Only procedures currently active have stack frames on the runtime stack.

The stack unwinder is always able to find the names of the procedures on the stack, even when they belong to coroutines other than the main one. However the module needs to be compiled with the **–g** option to have the line number information stored.

Now let us consider the first example in more detail.

```
**** m2rts: assert error: <sqrt of negative value> ****
abnormal program termination
<foreign stack frame>
RealMath_sqrt   Line-num = ??, file [realmath.mod]
GetAndTestValue Line-num = 18, file [mathtest.mod]
StartMathTest   Line-num = 33, file [mathtest.mod]
```

The first line tells us that an assert error has been detected in the program, and the error has the associated message *sqrt of negative value*. This is a very specific clue.

The stack unwind lines tell us that the error was detected in the procedure `RealMath_sqrt` from the file `realmath.mod`. This first is the linker name for the procedure *RealMath.sqrt* from the standard library *RealMath*. The line number is not of practical importance, since the source of this module is not released to users. In fact, the module has been compiled without the –g switch, so that the no line number is shown.

The next line tells us that the `sqrt` procedure was called from a procedure `GetAndTestValue` in the file `mathtest.mod`. (The fact that the module name is not prepended to the procedure name shows that this is not an exported procedure.) This procedure was, in turn, called from the procedure with the synthetic name `StartMathTest`. This "procedure" is the body of the main module of the program. In this case the mathtest module has been compiled with the –g switch.

Procedures which are not implemented in D-code do not create the characteristic interpreter stack frames. In such cases, the stack unwinder skips over the frame, emitting the message

```
<foreign stack frame>
```

If procedures with foreign implementations call D-code procedures, the foreign stack frames may even appear in the middle of the stack unwind record.

Stack unwinding and coroutines

When a program crashes in a coroutine other than the main coroutine the stack unwinder is invoked in the usual way. In this case the procedure call chain appears to start out of nowhere, since the base of the coroutine is not a module body.

```
**** m2rts: assert error: <sqrt of negative value> ****
abnormal program termination
<foreign stack frame>
RealMath_sqrt   Line-num = ??, file [realmath.mod]
GetAndTestValue Line-num = 18, file [cotest3.mod]
Body1   Line-num = 47, file [cotest3.mod]
<foreign stack frame>
```

In practice, the procedure *Coroutines.NEWCOROUTINE* always sets up a dummy base frame on the stack. This frame appears to be a call from the runtime support routine `__endTrp`. This deception ensures that termination of a coroutine "returns" to the trap. The trap is responsible for producing the *Coroutine ended without transfer* message.

Thus the stack unwinding of crashed coroutines always appears to start with a foreign stack frame. This is the dummy frame which points to the end trap. In the case of programs which abort because a coroutine has ended without a transfer, the endtrap routine is all that is left on the stack and the following characteristic message is produced —

```
**** m2rts: Coroutine ended without TRANSFER
abnormal program termination
<foreign stack frame>
```

This indicates that the procedure call chain of the coroutine has underflowed, and that no Modula stack frames are currently active.

Using the –g option

If the –g option is not used, a useful stack unwind record is still produced, with both procedure and filenames shown for all the procedures implemented in Modula-2. However, in order to gain maximum debugging information it is necessary to compile with the option on.

When the option is used, the compiler places special line-marker D-codes in the object file. These D-codes are treated as no-op instructions by the interpreter, but enable the line numbers to be found for an aborting program. The presence of these additional bytes in the object file expands the file by a small amount (less than 2000 extra bytes in a typical 1000 line program). It also slows the execution by about 5 percent, as the dummy codes have to be fetched and skipped. These extra overheads usually constitute a worthwhile investment during the development of programs.

Appendix H

The PC-specific libraries

There are three special libraries for **gpm-pc**. These are *Wildcards, PcProcesses* and a special version of the normal *UxFiles* module, which understands DOS file attributes.

Some libraries in **gpm-pc** are different in ways which do not affect the user. It is possible that libraries which are *FOREIGN* in *UNIX* versions are implemented in Modula in the pc version. *RealInOut* is an example.

Similarly, libraries which are *INTERFACE* in *UNIX* versions might be *FOREIGN* in **gpm-pc**. This is the case for *FREXP*, which is implemented as part of the floating point simulator in the interpreter of *gpm-pc*.

All other supplied libraries have the same definitions and as far as possible the same semantics as the *UNIX* versions. However, the chapter in the release notes on portability should be consulted for any other slight differences which are enforced by the differences in the underlying operating systems.

H.1 The PcProcesses library

This library provides a mapping to the DOS system facilities for spawning of programs. For some programs, these libraries can replace the facilities provided by *UxProcesses*. For example, the driver programs `build` and `gpm` use the *Spawns* function to execute the compiler (and, if necessary the editor). In the *UNIX* versions *fork, exec* and *wait* calls provide the same functionality.

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

FOREIGN DEFINITION MODULE PcProcesses;

  IMPORT IMPLEMENTATION FROM "pcprocesses.obj";

  FROM Types IMPORT SHORTINT;
  FROM SYSTEM IMPORT ADDRESS;
  IMPORT BuildArgs;

  PROCEDURE Spawns(comPath : ARRAY OF CHAR;
                   argStrn : ARRAY OF CHAR) : SHORTINT;
  (* Spawns another process, and waits for return result *)
  (* comPath is an absolute pathname, with extension.    *)
  (* argStrn is the additional arguments of the command  *)
  (* Arg-0 of command is comPath, others from argStrn.   *)
  (* Result is exit code of the spawned process          *)

  PROCEDURE Spawnv(comPath : ARRAY OF CHAR;
                   argvBlk : BuildArgs.ArgPtr) : SHORTINT;
  (* Spawns another process, and waits for return result *)
  (* Discards zero-th argument and concatenates the rest *)
  (* to form a standard command string for Spawns.       *)
  (* Arg-0 of command is comPath, others from argbBlk.   *)
  (* Result is exit code of the spawned process          *)

  PROCEDURE System(command : ARRAY OF CHAR) : SHORTINT;
  (* Spawns another copy of the command processor as     *)
  (* specified by the environment variable COMSPEC, this *)
  (* executes the command. Returns non-zero on failure   *)

  PROCEDURE PSP() : ADDRESS;

END PcProcesses.
```

There are three flavours for the spawn command. All of them work by releasing all memory above the current top of the heap. The DOS `exec` function then spawns and executes the specified program. After the specified program has executed, control passes back to the caller, or *parent* process. All available memory is once more allocated to the parent process, so that the parent may expand the heap to larger sizes than before the spawning operation.

System works by spawning a copy of the command processor and passing the command parameter to that program. Ususally the commmand processor, specified by the environment variable `COMSPEC` is the program `COMMAND.COM`. This command is the simplest to use, since the command processor automatically appends the required `.bat, .com, .exe` extension and searches the specified path.

In cases where it is important to give as much space as possible to the child process, one of the *Spawn* family is often a better choice than *System*. Additionally, the *Spawn* functions are able to return the exit code of the child process, but *System* cannot.

The *Spawn*`s` function takes two strings as parameters. The first is the name of the program to be spawned, while the second is the command string to be passed to the program. It is important to realise that the first parameter must be an absolute pathname, unless the file is in the current directory. In any case, the file must have an explicit extension such as `f:\bin\vi.exe`. The second string provides the additional arguments to the spawned program. It is commonly the case that the procedure *PathLookup.FindAbsName* must be used to find the file on the executable path. The overhead of including and linking this library is very much less than the space used by an additional copy of `COMMAND.COM`.

The *Spawn*`v` function takes a *UNIX* style argument block as second parameter. The type is a pointer to a *NIL* terminated array of *Ascii.nul* terminated character strings. The module *BuildArgs* provides portable facilities for manipulating these blocks as abstract data types. It is normal under *UNIX* for the first argument in such an argument block to be the program name. The *Spawnv* function actually discards the first argument and concatenates the rest to form a command line, then chains internally to the underlying *Spawns* function. Using *Spawns* is more in the *UNIX* style, and will require less work to transform into a *Fork* and *Exec* call.

Under *UNIX* a program, say `testargs`, invoked using

```
Exec("/usr/bin/testargs",Arg2("testargs","foo"));
```

will have first argument `"testargs"`, and second argument `"foo"`. By contrast, under dos, the call

```
Spawnv("\usr\bin\testargs.exe",Arg2("testargs","foo"));
```

will have first argument `"\usr\bin\testargs.exe"`, and second argument `"foo"`. This passing of the absolute pathname to the program is the closest DOS can come to emulating the *UNIX* behaviour, and only works for versions of DOS later than 3.0.

H.2 The DOS version of UxFiles

This library has differences only in the file permission bits, which have a different role under DOS. In this case they indicate whether the file has the archive bit set, whether it is a directory and so on. The attributes are exactly the bits defined by Microsoft in the file control block documentation.

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

FOREIGN DEFINITION MODULE UxFiles;

  IMPORT IMPLEMENTATION FROM "uxfiles.obj";

(*
 * WARNING WARNING: THIS IS THE MSDOS VERSION.  FILE    *
 * PERMISSION BITS ARE DIFFERENT TO THE UNIX VERSIONS   *
 *
 * This module provides the low level interface to the  *
 * UNIX file system, it links to the library <stdio.h>  *
 * The user programs are protected against the UNIX     *
 * identifiers which are introduced in the header file  *)

FROM SYSTEM   IMPORT ADDRESS, BYTE;

TYPE
  File;
  OpenMode = (ReadOnly, WriteOnly, ReadWrite);

  FilePermissionBits =
              (rdOnly,hidden,system,volId,subDir,archive);
  FileMode =    SET OF FilePermissionBits;

  FileAttrib = FilePermissionBits; (* synonyms for use by *)
  FileAttSet = FileMode;           (* WildCards library   *)

PROCEDURE GetMode(  name : ARRAY OF CHAR;
                VAR mode : FileMode;
                VAR done : BOOLEAN);
(* precondition  : name must be a nul-terminated variable
                   array,or a literal string.
   postcondition : if done then mode has permission bits *)
```

... Continued

```
PROCEDURE SetMode(  name : ARRAY OF CHAR;
                    mode : FileMode;
                VAR done : BOOLEAN);
(* precondition  : name must be a nul-terminated variable
                   array,or a literal string.
   postcondition : if done then file has permission bits *)

PROCEDURE Open(VAR f:    File;       (* Open an existing file *)
                   name: ARRAY OF CHAR;
                   mode: OpenMode;
               VAR done: BOOLEAN);

PROCEDURE Create(VAR f:    File;     (* Open a new file *)
                     name: ARRAY OF CHAR;
                 VAR done: BOOLEAN);

PROCEDURE Close(VAR f:    File;      (* Close a file *)
                VAR done: BOOLEAN);

PROCEDURE Delete(str : ARRAY OF CHAR;
             VAR ok  : BOOLEAN);

PROCEDURE Reset(f: File);
(* Position the file at the beginning and set to "ReadMode"   *)

PROCEDURE ReadNBytes(    f:              File;
                         buffPtr:        ADDRESS;
                         requestedBytes: CARDINAL;
                     VAR read:           CARDINAL);
  (* Read requested bytes into buffer at address *)
  (* 'buffPtr', number of effectively read bytes *)
  (* is returned in 'read'                       *)

PROCEDURE WriteNBytes(    f:              File;
                          buffPtr:        ADDRESS;
                          requestedBytes: CARDINAL;
                      VAR written:        CARDINAL);
  (* Write requested bytes from buffer at address    *)
  (* 'buffPtr', number of effectively written bytes  *)
  (* is returned in 'written'                        *)
```

... Continued

```
PROCEDURE ReadByte(    f: File;      (* Read a byte from file *)
                   VAR b: BYTE);

PROCEDURE WriteByte( f: File;     (* Write a word to file *)
                     b: BYTE);

PROCEDURE EndFile( f : File) : BOOLEAN;
  (* returns true if an attempt has been made
     to read past the physical end of file   *)

PROCEDURE GetPos(   f : File;
                VAR p : CARDINAL);

PROCEDURE SetPos( f : File;
                  p : CARDINAL);
  (* GetPos and SetPos get and set the file position *)

PROCEDURE AccessTime(path     : ARRAY OF CHAR;
                     VAR time : CARDINAL;
                     VAR ok   : BOOLEAN);
  (* finds time of last access to named file *)

PROCEDURE ModifyTime(path     : ARRAY OF CHAR;
                     VAR time : CARDINAL;
                     VAR ok   : BOOLEAN);
  (* finds time of last modification to file *)

PROCEDURE StatusTime(path     : ARRAY OF CHAR;
                     VAR time : CARDINAL;
                     VAR ok   : BOOLEAN);
  (* finds time of last status change of file *)

END UxFiles.
```

H.3 The WildCards library

This library provides a straightforward interface to the standard DOS functions for searching for wildcard filenames.

```
(* ======================================================== *)
(* Preliminary library module for Gardens Point Modula      *)
(* This module is part of the gpm-pc distribution. It is    *)
(* required because the DOS shell does not expand wildcards. *)
(* ======================================================== *)

FOREIGN DEFINITION MODULE Wildcards;

  IMPORT IMPLEMENTATION FROM "wildcard.obj";
  IMPORT Types, UxFiles;

  TYPE  FileAttrib = UxFiles.FileAttrib;
            (* rdOnly,hidden,system,volId,subDir,archive *)
        FileAttSet = UxFiles.FileAttSet;
            (*   SET OF FileAttSet  *)

  TYPE FfBlk = RECORD
                 reserved : ARRAY [0 .. 21] OF CHAR;
                 fftime   : Types.Int16;
                 ffdate   : Types.Int16;
                 ffsize   : CARDINAL;
                 ffname   : ARRAY [0 .. 12] OF CHAR;
               END;

  PROCEDURE FileAttOf(ffBlk  : FfBlk) : FileAttSet;
  (* extracts attribute set from the given FfBlk *)

  PROCEDURE FindFirst(pathNm : ARRAY OF CHAR;
                      attrib : FileAttSet;
                  VAR ffBlk  : FfBlk;
                  VAR found  : BOOLEAN);

  PROCEDURE FindNext (VAR ffBlk : FfBlk;
                      VAR found : BOOLEAN);

END Wildcards.
```

Examples of use

Here is a simple program which demonstrates usage of the *WildCards* library. It accepts arguments from the command line, and then finds any files that match these patterns in the current directory. A more complex example on the distribution is the source code for the compiler driver program gpd.mod. This file demonstrates the use of both the *WildCards* and *PcProcesses* libraries.

```
MODULE WildTest; (* demonstrates the Wildcards library *)

  FROM ProgArgs IMPORT ArgNumber, GetArg;
  FROM InOut IMPORT WriteString, WriteLn;
  FROM Wildcards IMPORT
         FileAttOf, FindFirst, FindNext, FileAttrib, FileAttSet, FfBlk;

  VAR ffblk : FfBlk;
      found : BOOLEAN;
      count : CARDINAL;

  VAR cArg  : ARRAY [0 .. 79] OF CHAR;

BEGIN
  FOR count := 1 TO ArgNumber() - 1 DO
    GetArg(count,cArg);
    WriteLn;
    WriteString("Command line arg -- ");
    WriteString(cArg);
    WriteLn;
    FindFirst(cArg, FileAttSet{subDir}, ffblk, found);
    WHILE found DO
      WriteString(ffblk.ffname);
      IF subDir IN FileAttOf(ffblk) THEN WriteString(" <dir>") END;
      WriteLn;
      FindNext(ffblk,found);
    END; (* while *)
  END;
END WildTest.
```

The second example program uses the *FileMode* type of the DOS version of *UxFiles* to print information regarding the mode of named files. The program interactively accepts filenames from the user, looks up the files, displaying their attributes if found.

The program displays the characters **r a s h** to indicate if the file has the read-only, archive, system, and hidden attributes. It is a simple exercise to combine the ideas of the two demonstration programs to print the attributes of files found as a result of wildcard lookups.

```
MODULE ModeTest;
  IMPORT InOut, UxFiles;
  FROM InOut IMPORT Write;

  VAR  str : ARRAY [0 .. 127] OF CHAR;
       ok  : BOOLEAN;
       mod : UxFiles.FileMode;
```

```
       index : CARDINAL;

BEGIN
  InOut.WriteString("File mode-test : type a filename, ^C to exit");
  InOut.WriteLn;
  LOOP
    InOut.WriteString(">> ");
    InOut.ReadString(str);
    UxFiles.GetMode(str,mod,ok);
    IF ok THEN
      IF UxFiles.subDir IN mod THEN
        InOut.WriteString("directory");
      ELSIF UxFiles.volId IN mod THEN
        InOut.WriteString("volume-ID");
      ELSE
        IF UxFiles.rdOnly IN mod THEN Write("r") ELSE Write(".") END;
        IF UxFiles.archive IN mod THEN Write("a") ELSE Write(".") END;
        IF UxFiles.system IN mod THEN Write("s") ELSE Write(".") END;
        IF UxFiles.hidden IN mod THEN Write("h") ELSE Write(".") END;
      END;
    ELSE InOut.WriteString("ModeTest: file not found");
    END;
    InOut.WriteLn;
  END;
END ModeTest.
```