AUSTRALIAN NUCLEAR SCIENCE
AND TECHNOLOGY ORGANISATION

LUCAS HEIGHTS RESEARCH LABORATORIES

MOD2: A PRE-PROCESSOR FOR THE MODULA-2 LANGUAGE FOR C COMPILERS

by

J. CRAWFORD

F. P. CRAWFORD†

ABSTRACT

This report describes a translator from Modula-2 to C written for a Pyramid 90x computer. It contains an overview of the Modula-2 language, followed by a detailed description of the translator. Also included is a description of how to use the translator and a summary of the installation procedure.

† Now at Q.H. Tours, PO Box 630, North Sydney, NSW 2060

National Library of Australia card number and ISBN 0 642 59875 4

The Australian Nuclear Science and Technology Organisation replaced the Australian Atomic Energy Commission on 27 April 1987. Reports issued after April 1987 have the prefix ANSTO with no change of the symbol (E, M, S or C) or numbering sequence.

CONTENTS

1. INTRODUCTION

Modula-2 is a general-purpose programming language, designed primarily for system programming. Although it is similar to its predecessor Pascal, the design goals for each were quite different. Pascal was designed by Wirth [Jensen and Wirth 1975] as a teaching language that emphasised structured programming concepts and portability, whereas Modula-2 [Wirth 1980] has taken most of the features of Pascal and transformed or extended them to allow system and multiprogramming.

In its general features, Modula-2 clearly demonstrates the influence of Pascal. It has adopted most of the data-type concepts of Pascal with some significant additions. The language's main additions are as follows:

(i) The *module* concept, and in particular the facility to split a module into a *definition* part and an *implementation* part.

(ii) The concept of the *process* as the key to multiprogramming facilities.

(iii) The *low-level facilities* which make it possible to breach the rigid type consistency rules.

(iv) The *procedure type* which allows procedures to be dynamically assigned to variables.

Unfortunately, there are not many Modula-2 compilers available, and none for any of the operating systems supported at the Lucas Heights Research Laboratories. There are, however, a number of C compilers available for these systems. As C is also a system programming language it was decided to develop a translator to convert Modula-2 programs to C.

Other possible languages as the target of the translator were FORTRAN and Pascal. FORTRAN is so dissimilar to Modula-2 that an enormous effort would have been required. As an example, FORTRAN does not support recursion, a very basic concept in Modula-2 (and C). Although Pascal is more similar to Modula-2 than C, Modula-2's extensions over Pascal would be difficult, if not impossible, to implement in Pascal.

The language C was designed and implemented in 1972 by Ritchie [Kernighan and Ritchie 1978]. It has been used to implement a wide variety of applications, mostly under the UNIX™ operating system. It has proved to be a very popular language, with compilers being developed for most modern computers, including many of the computers used at Lucas Heights.

C is a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators. It is neither a 'very high level' language, nor a 'big' one, and is not dedicated to any particular area of application. But its absence of restrictions and its generality made it more convenient and effective for this particular task, because it has many of the features also available under Modula-2.

The option of developing a Modula-2 compiler rather than a translator was rejected for many reasons, but primarily both because of the time involved and the objective of making the result portable across a number of different machines.

2. DESCRIPTION OF THE PRE-PROCESSOR

There is no one program that can be said to be the Modula-2 to C translator; rather it is a set of separate programs each of which performs a specific function. The flow of translation is shown in **figure 1.**

™ UNIX is a registered trademark of AT&T in the USA and other countries.

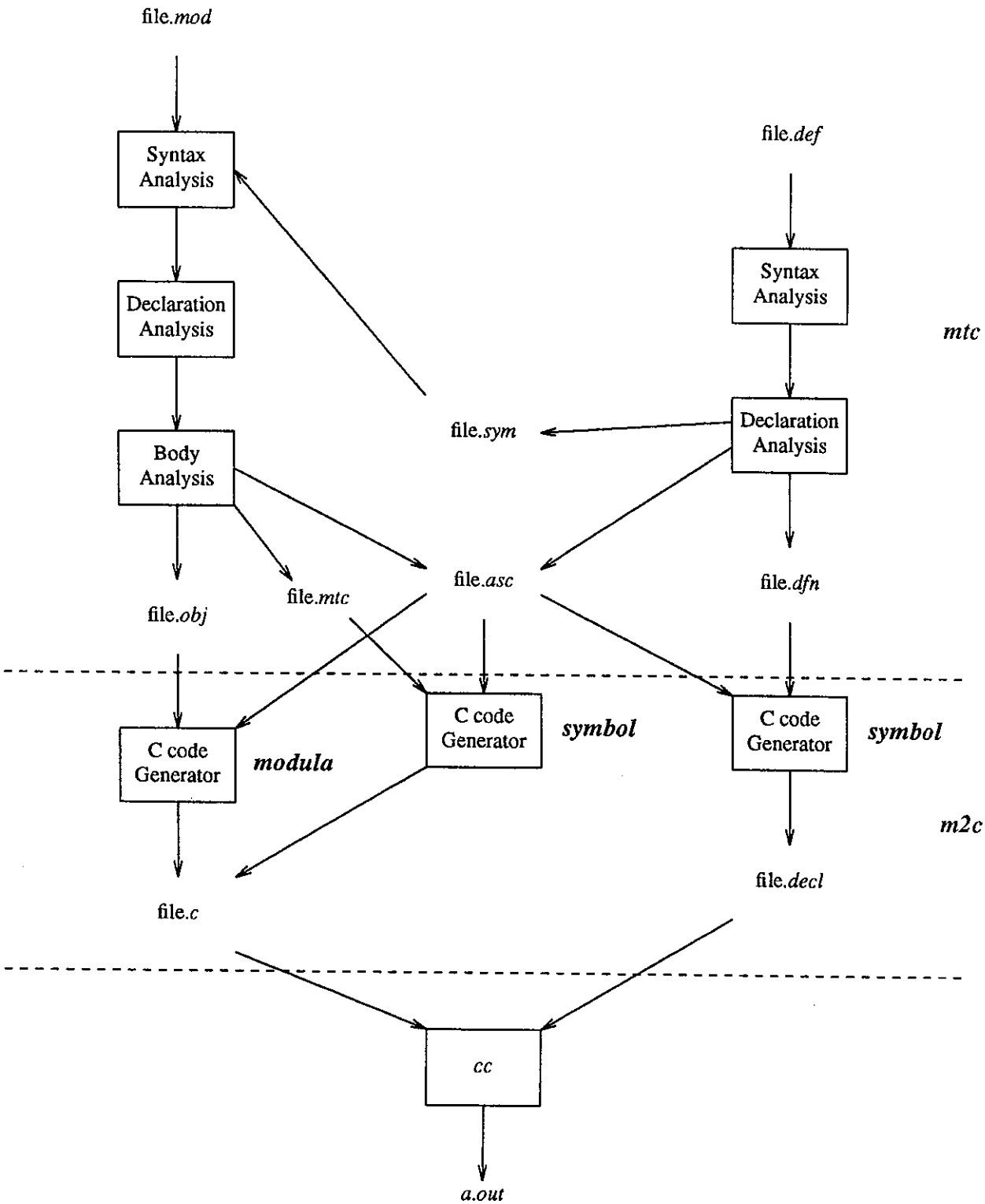

Figure 1. Compiler Overview

The translation consists of a number of distinct parts, namely,

(a) parsing and transformation into a tokenised form (performed by *mtc*),

(b) generation of the appropriate C declarations and definitions (largely performed by *symbol*),

(c) generation of the C code for statements and expressions (performed by *modula*),

(d) merging of the separate parts of the C code, *i.e.* (b) and (c) above, (performed by *m2c*), and

(e) compilation of the C code to produce an executable file (performed by *cc*).

The inclusion of the C compiler in this list allows executable code to be generated on any system, however this is ancillary to the translator.

2.1 *mod2.sh*

As there are a number of different programs to be invoked in a set sequence (see **appendix A**), rather than expecting a user to carry out the sequence, it is preferable to automate the process. Under the UNIX operating system this is done by creating a *shell script*. In its simplest form this is just a list of commands to be executed sequentially by the user's chosen command processor (*i.e. shell*). In more complicated examples, it can include high-level control flow structures and variable substitution.

A script to perform a Modula-2 translation and subsequent compilation of the C code is called *mod2.sh* or, more commonly, *mod2* (see **appendix B**). This script takes as its arguments a number of options to control the translation, followed by a list of filenames whose extensions indicate what type of translation is required. A full description of the use of this procedure is given in **appendix C**, with a listing of the error messages that might be given in a compilation in **appendix D**. The extensions understood by *mod2.sh* are given in **table 1**.

TABLE 1.
EXTENSIONS ACCEPTED BY *mod2.sh*

Ext	Program
.mod	*mtc*
.def	*mtc*
.obj	*m2c*
.mtc	*m2c*
.dfn	*m2c*
.c	*cc*
.o	*cc*

2.2 *mtc*

mtc was developed from Wirth's original Modula-2 compiler [Geissmann 1981] implemented on the Lilith computer. The original compiler was written in Modula-2, containing four separate passes and generated 'M-code'. Aside from the Lilith compiler, a Modula-2 compiler producing code for PDP-11 computers running RT-11 had been developed [Geissmann and Jacobi 1981] which was later ported to a PDP-11/45, running UNIX Version 6, by Dr J. Tobias.

The compiler was originally organised into a base part and a number of separate overlays, with the base remaining in memory for the whole of the execution and the overlays loaded as required. The reason for this was to overcome addressing limitation on small address space machines. As this problem does not apply to the Pyramid computer, the structure has been reorganised to eliminate overlays, by incorporating them as procedures.

Since the aim of the project was to produce C code, the first three passes of the Lilith compiler were taken and modified to produce output suitable for conversion. This involved extensive modifications to the format of the symbol table to keep more detailed information on the structure and order of declarations, to keep track of the scope of variables, to include back pointers to procedures and modules, to generate

identifiers unique to the C code, and to maintain import and export lists.

As well, the output in the intermediate code for statements and expressions contains additional information about the types of the operands and information to remove the nesting of procedures, to generate appropriate declarations for imports and exports, and to handle differences in the method of passing arguments to procedures/functions.

Mtc handles both *implementation* and *definition* modules. It assumes that files with a *.def* extension are *definition modules* and those with a *.mod* extension are either an *implementation module* or a *program module*. Because *mtc* requires all imported *definition modules* to be compiled before they are referenced by any other module, it is generally necessary to compile all the *definition modules* first. Further, the compiler generates a timestamp on each *definition module* compilation; recompilation of such a module can generate errors when compiling other modules. In this case, the *implementation module* and all units importing this module must be recompiled as well.

2.3 *m2c*

The generation of the C code from the intermediate form produced by *mtc* is accomplished by *m2c*. The process is divided into two parts - the generation of declarations, and the generation of statements and expressions. These are carried out by separate programs (*symbol* and *modula*) and the resulting output is combined to form the final C program.

m2c is a simple program that invokes *symbol* and *modula* and then, by the use of UNIX *pipes* (tools that allow the user to take the output of one command and use it as the input for another command without creating temporary files), it reads the output from each program in turn, switching between the two when it detects a specific synchronisation character in the input stream. Processing of this input ceases and *m2c* writes this merged stream to generate the C code.

2.3.1 *symbol*

symbol is used to produce both the *definition module* and the definitions and declarations within *implementation* and *program modules*. It is a recursive descent parser for the grammar described in an Extended Backus-Naur Form (EBNF) given in **appendix E.**

It takes its input from an intermediate file generated by *mtc* containing a stream of one byte long tokens. Constants such as integer and real numbers take an appropriate number of bytes. Strings end with a NUL (*i.e.* a single byte of value 0). Identifiers are stored in a separate file and an offset in this file is given.

2.3.2 *modula*

modula is used to produce the statements and expressions from the *implementation* and *program modules*. As with *symbol*, it is a recursive descent parser for the grammar described in EBNF given in **appendix F.**

It takes as its input a stream of two-byte tokens, the first byte of which contains the token value and the second its position on the current Modula-2 source code line (presently ignored). Constants, strings and identifiers are handled in a similar manner to *symbol*.

3. DEVELOPMENT OF THE PRE-PROCESSOR

The development of the translator stemmed from an existing compiler on the PDP-11 computer, as described above. The initial stages involved modifying the Lilith compiler on the PDP and sending the intermediate output to the Pyramid for the final stages of the translation. Later the required passes of the compiler were run through the above procedure to move all of the separate parts onto the same machine (the Pyramid).

The modifications performed to *mtc* were relatively straightforward once the translation procedure was established. The major problem was to remain within the size restraints imposed by the PDP's architecture, as the existing compiler was already close to the limits.

The development of the parts on the Pyramid was a much more complicated process, as all the programs had to be developed from basic principles. This was aided by the tools available under UNIX. These included

- *yacc* [Johnson 1979], a program to take a syntax description and generate efficient C code to parse the grammar,
- *make* [Feldman 1979], a program to maintain a group of files, and using a knowledge of their dependencies perform appropriate operations on the files to generate new programs, and
- *error*, which inserts error messages in the correct locations in the source files.

The effects of some of these tools can be seen in the structure of the translator. For example, *yacc* can only parse one grammar at a time; to parse the two intermediate streams it is necessary to write the separated programs, *symbol* and *modula*, then to merge the results.

4. EVALUATION OF THE PRE-PROCESSOR

A final step in the development of the translator was the evaluation of its performance, in relation to its usability, its conformance with the standard and its speed.

The shell script *mod2*, enables the user to treat the compilation of a Modula-2 program in a manner similar to any other compiler on the system, for example *cc*. The options for the casual user are the standard over the operating system, whereas those who wish to use all the features of the translator are still accomodated.

As the first three passes are based on another Modula-2 compiler, the translator conforms very closely to the language as described by Wirth [1980]. Further, as Modula-2 and C are similar, almost all of the features of Modula-2 are supported. This has been demonstrated both by the translation of the compiler, which makes extensive use of the language, and a number of other programs, most of which required no changes and, in the worst case, only the modification of some non-portable features.

In terms of speed of the translator/compiler, there are two different aspects - the time taken for compilation and the execution time of the program. Obviously, the time taken for compilation will be greater than for the equivalent C program. Some simple tests, on small programs, have indicated that compiling a program with *mod2* takes about twice as long as the compilation of an equivalent C program. However, this relative difference should not be expected for larger programs, as *mod2* has a number of predeclared objects which would add a constant overhead to the compilation time. The relative difference would probably reduce to a factor of about 1.5 for large programs.

The execution time of the resulting code is of more interest, and the results of a number of test are given in the **table 2**. The basic algorithm used for the results was Eratosthenes' sieve for generating prime numbers. This was coded in Modula-2 (and converted to C on the Pyramid 90x), C and Pascal and the time taken to generate the numbers between one and 10003 repeated 100 times was found. This test was conducted ten times and the results averaged on an IBM 4381-3 (System 370), PDP-11/45, Pyramid 90x and a VAX-11/780. A limited test was also conducted on an IBM PC-XT. To further test the efficiency of the code two variants of the program were tested, declaring variables dynamically (*e.g. register* for C, local to the procedure for Pascal and Modula-2) and as static or global quantities.

In general, the times for the Modula-2 translator compare favourably considering that no effort has been given to producing efficient code. The only case where there is a major difference in time between the output from the translator and the other languages is for local references on the Pyramid 90x. This is caused by the architecture of that machine. If possible, all local variables of the size of an integer are stored in a register and both C and Pascal take advantage of this; however, the Modula-2 translator generates structures rather than separate variables and cannot take advantage of this fact.

TABLE 2.
EXECUTION TIMES (s) FOR ERATOSTHENES' SIEVE

Machine	Local			Global		
	C	Pascal	Modula-2	C	Pascal	Modula-2
IBM 4381	3.92	4.41	8.23	6.47	4.41	6.65
Pyramid 90x	6.48	7.56	24.13	23.20	24.44	24.04
VAX-11/780	13.92	18.28	16.73	16.31	17.95	16.84
PDP 11/45	26.64	46.43[1]	48.08	42.97	49.03[1]	46.18
IBM PC-XT	70.2	90.2	95.1	89.4	93.4	91.6

5. UNSUPPORTED FEATURES

Although the translator supports most of the features of Modula-2, there are three that have not been implemented:

- *Concurrent processes.* These could be implemented, but, as it was not needed for the applications we were interested in, this has not been done. Furthermore, something similar could be done using the existing system calls *fork* and *exec*.
- *Priorities and monitors.* In Modula-2, these features are provided with the intention of developing operating systems. Under UNIX, low level interrupts are handled by the system itself and therefore the features were not implemented.
- *Runtime index tests.* This is a possible future feature, as the type of environments that the programs will run in often require it.

6. FUTURE DIRECTIONS

As the translator is working, the task has changed to one of improving it, both in terms of the code produced and its internal operation. The first task is to remove as much redundant code as possible. This is both possible and simple as, now that the previous restrictions on the size of the code have been lifted, the original passes of the Modula-2 compiler can be examined as a single unit rather than as separate units.

The original Modula-2 compiler, and this translator, were based on the first definition of Wirth [1982]. Since then, he has published two more [Wirth 1983, 1985] As each edition has introduced small changes to the language, a future task is to bring the translator up to date.

7. CONCLUSIONS

The Modula-2 to C translator, *mod2*, is now running on the ANSTO Pyramid computer and is being used for its original purpose, that of moving programs from the aging PDP-11/45. Even though it is now in general use it is in need of further testing and, as indicated above, there is probably more development work to be done.

The translator has proved to be very popular at sites other than Lucas Heights. There have been a number of requests to install it on Pyramids and other computers. This popularity is partly due to the increased emphasis on Modula-2 within universities, but also because of the flexibility of the translator. The ability to use it on any machine which has a C compiler conforming to recent standards means that it is not restricted by either machine type, or even operating system.

---

[1] Compiled with Modula-2 compiler as no Pascal compiler was available.

The project has demonstrated the closeness in structure of Modula-2 and C. Despite the fact that they use different symbols for operators and control structures, the difference is merely cosmetic, the underlying fabric being the same.

8. REFERENCES

Feldman, F. I. [1979] - Make - A Program for Maintaining Computer Programs, UNIX Programmer's Manual, 2A. Bell Telephone Laboratories Inc. Seventh Edition.

Geissmann, L. [1981] - Overview of the Modula-2 Compiler. Institut fur Informatik, ETH. Draft version.

Geissmann, L. and Jacobi, C. [1981] - Overview of the Modula-2 Compiler M2M. Institut fur Informatik, ETH. Draft version.

Jensen, K. and Wirth, N. [1975] - Pascal User Manual and Report. Springer-Verlag, New York. Second Edition.

Johnson, S. C., [1979] - Yacc: Yet Another Compiler-Compiler, UNIX Programmer's Manual, 2B. Bell Telephone Laboratories Inc. Seventh Edition.

Kernighan, B. W. and Ritchie, D. M. [1978] - *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey.

Wirth, N. [1980] - "MODULA-2". *ETH Institut fur Informatik Report No. 36.*

Wirth, N. [1982] - Programming in Modula-2. Springer-Verlag, Berlin, Germany.

Wirth, N. [1983] - Programming in Modula-2. Springer-Verlag, Berlin, Germany. Second Edition.

Wirth, N. [1985] - Programming in Modula-2. Springer-Verlag, Berlin, Germany. Third Edition.

APPENDIX A
mod2 COMPONENTS

There are a number of separate processes required for the translation from the Modula-2 source code to the final executable:

- *mtc*,
- *m2c*, and
- *cc*.

These programs also have a number of separate sub-components:

(i) for *m2c*

- *symbol*, and
- *modula*.

(ii) for *cc*

- *cpp*,
- *ccom*,
- *as*, and
- *ld*.

In this report we assume that *cc* can be treated as a single module, details of the options and the separate parts can be found in the UNIX Programmer's Manual, Vol 1.

A1. *mtc*

mtc takes the Modula-2 source and generates the intermediate code. It is invoked with the following command:

mtc [-options] filename

where [*-options*] are

Option	Default	Meaning
-l	off	Generate a listing file
-q	off	Query for symbol file names
-v	off	Print compiler version information

N.B. All options are given as a single letter preceeded by a dash ('-').

The *filename* is the name of the file to be translated. The suffix *.mod* implies an *implementation* module or a *program* module and *.def* implies a *definition* module. If a suffix is not given the default is assumed to be *.mod*.

During execution of *mtc*, a message indicating the start of each pass is written to the standard output, and any errors detected during this pass are signalled by the message "---- error" on the standard output, with the corresponding error number written in the listing file (if a listing is requested - see '*-l*' above). If errors are detected then parsing stops after pass 3.

A2. *m2c*

As explained previously, *m2c* runs two separate processes and handles the interleaving of their output.

This is accomplished by invoking the two programs simultaneously, with an option to produce a synchronisation character at appropriate points. The synchronisation charater is currently ASCII character STX (hex 2) and must occur as the first character on the line. This entire line is discarded.

m2c is used as follows:

m2c [-a ascfile] [-D] [-d] [-h] [-n] [-O] [-o outfile] [-p] [-s symfile] [-v] infile

Possible options for *m2c* are

Options	Default	Meaning
-a *ascfile*	file.*asc*	File name for stings file from *mtc*.
-D	false	Force *symbol* to treat input as a definition module.
-d#	0	Set debug level (0-3).
		0 - no debug
		1 - debug output from *m2c*
		2 - debug output from *modula*
		3 - debug output from *symbol*
-h	false	Print usage message.
-n	false	Force *modula* not to test for a definition module.
-O	false	Process intermediate files in PDP format (512 byte blocks).
-o *outfile*	file.*c*	Output file name.
		An output file name of '-' signifies standard output.
-p	false	Parse only - no output files produced.
-s *symfile*	file.*mtc*	Declaration file name from *mtc*.
	file.*dfn*	
-v	false	Verbose option - print more informational messages.

N.B. Multiple options can be concatenated where it makes sense, *e.g.* the string '-D -n -O' is equivalent to '-DnO'.

In most cases, the options to *m2c* are passed directly to the relevant program, the only cases where this is not so is with the debug option ('-d') and with the symbol file option ('-s') which becomes the file name given to *symbol* to be processed.

m2c can be given multiple file names to be processed, which can be either of the form file.*obj* for an *implementation* or program module, or file.*dfn* for a *definition* module. If no extension is given then it is assumed to be an *implementation* module. The output file is given *.c* as an extension (*i.e.* a C program).

The only messages produced by *m2c* are informational or messages concerning implementation errors. One notable message is 'm2c: Extra ETX on objfile'[1] (or 'symfile'). This indicates that extra synchronisation characters were found on one of the input streams after the other had terminated. It is usually caused by either *modula* or *symbol* terminating abnormally.

A3. *modula*

modula processes the statements and expressions from *mtc* and can be run separately for debugging purposes.

1. The message should read 'm2c: Extra STX ...' but has not been changed for historical reasons.

Its usage is:

modula [-a ascfile] [-c] [-d] [-h] [-O] [-o outfile] [-n] [-p] [-s symfile] infile

The meaning of most of these options is the same as for *m2c*, with the only differences being

Option	Default	Meaning
-c	false	Include synchronisation characters in output.
-d	false	Debug enabled.
-s		Currently not supported.

At present only a single file can be processed; it follows the same naming conventions for *implementation* or program modules in *m2c*, and produces a file with extension *.c*[2]. Unless the verbose or debug option is given there should be no messages produced.

modula has the ability to invoke *symbol* if it finds that a new declaration file is required.

A4. *symbol*

symbol processes the declarations from both *implementation* and program modules and from *definition* modules. It is invoked by

symbol [-a ascfile] [-c] [-d] [-D] [-h] [-O] [-o outfile] [-p] infile

where the options are as for *m2c* and *modula*.

As for *modula*, only one file can be processed at a time. The input and output file extensions depend on the module type and are given by

Module Type	Input	Output
Program	*.mtc*	*.c*
implementation	*.mtc*	*.c*
definition	*.dfn*	*.decl*

If no extension is given, it is assumed to be an *implementation* module. The output from a program or *implementation* module is expected to be merged with the output from *modula*, but that from the *definition* module should be used as an include file by the C preprocessor (*i.e. #include* "file.*decl*").

In the case of *modula*, unless the verbose or debug option is given no messages should be produced.

2. Although this is valid C code, it is not usually compilable as it is missing all the declarations.

APPENDIX B
EXTRACT FROM mod2.sh

The following extract is designed to indicate the procedure for automating the translation process. Its usage is described in **appendix C**, however this description is aimed at the level of a normal user rather than a developer.

Aside from the options given in **appendix C** there are a number of other options available within this script to allow more control. These options are

Option	Meaning
+t1*	Additional arguments to be given to the *mtc* step.
+t2*	Additional arguments to be given to the *m2c* step.
+t3*	Additional arguments to be given to the *cc* step.
+1	Only perform the first step (*mtc*).
+2	Only perform the first two steps (*mtc* and *m2c*).
+r	Do not remove any intermediate files when finished.

These options should not be used in normal circumstances and may be removed in the future.

This script should be taken only as a guide to how the parts of the translation process fit together and not as the only way to do the translation.

```
#! /bin/sh
#
#       Modula-2 to C translator
#
#       Written by:
#               Jagoda Crawford and Frank Crawford
#                                       28 Jul 86
#
#       Australian Nuclear Science and Technology Organisation
#


#
#       A shell to do all of the Modula-2 Compilation
#
# Some definitions
Bin=/u2/modula/Bin
Lib=/u2/modula/Lib
Include=/u2/modula/Include

PATH=$Bin:$PATH; export PATH

# Parse the args
while [ $# -ne 0 ]
do
    case "$1" in

    -q) mtcopt="$mtcopt -q"
        ;;

        ...
```

```
    +t[123]*)
      case `expr "$1" : '+t23]).*'` in

      1)  mtcopt="$mtcopt "`expr "$1" : '+t1)'`
          ;;
      2)  m2copt="$m2copt "`expr "$1" : '+t2)'`
          ;;
      3)  ccopt="$ccopt "`expr "$1" : '+t3)'`
          ;;
      esac
      ;;

    +1) nom2c=1
      nocc=1
      ;;

    +2) nocc=1
      ;;

    +r) norem=1
      ;;

    [-+]*|'')
        echo "Usage: `basename $0` [-qlv] [-Dd*n] [-cg*pOSI*] [+t[123]*] [+12r]\
[-o name] file [...]" 1>&2
        exit 1
        ;;

    *)  break
        ;;

    esac

    shift
done

if [ $# -eq 0 ]
then
    echo "Usage: `basename $0` [-qlv] [-Dd*n] [-cg*pOSI*] [+t[123]*] [+12r]\
[-o name] file [...]" 1>&2
    exit 1
fi

for i
do
    [ "$output" = "$i" ] && \
        echo "`basename $0`: -o option would overwrite $i" 1>&2 && \
        exit 1
    name=`basename $i`
    name=`expr "$name" : '\(.*\)\..*' \| "$name"`

    case `expr "$i" : '.*\.\(.*\)'` in

    mod) mtcfile="$mtcfile $i"
```

```
        m2cfile="$m2cfile $name.obj"
        ccfile="$ccfile $name.c"
        delfile="$delfile $name.tmp"
        [ "$nom2c" -ne 1 ] && delfile="$delfile $name.obj $name.mtc $name.asc"
        [ "$nocc" -ne 1 ] && delfile="$delfile $name.c"
        ;;
    def) mtcfile="$mtcfile $i"

        ...

    esac

done

if [ -n "$output" -a $# -ne 1 -a \( "$nocc" -eq 1 -o -z "$ccfile" \) ]
then
    echo "`basename $0`: -o option given with multiple output files" 1>&2
    exit 1
fi

[ -n "$mtcfile" ] && {
    mtc $mtcopt $mtcfile || exit
}
[ "$nom2c" -ne 1 -a -n "$m2cfile" ] && {
    m2c $m2copt $m2cfile || exit
}
[ "$nocc" -ne 1 -a -n "$ccfile" ] && {
    cc -o ${output-a.out} -I$Include/Decl $ccopt $ccfile $Lib/libmodula.a || exit
}
[ "$norem" -ne 1 -a -n "$delfile" ] && rm -f $delfile

exit 0
```

APPENDIX C
USAGE OF *mod2*

NAME

mod2 – Modula-2 compiler

SYNOPSIS

mod2 [option] name ...

DESCRIPTION

Mod2 is a Modula-2 compiler. If given an argument file ending with .mod, it will compile the file and load it into an executable file called, by default, *a.out*.

A program may be separated into more than one .mod file. *Mod2* will compile a number of argument .mod files into object files (with the extension .o in place of .mod). Argument files ending in .def are assumed to be a definition module and are converted into an equivalent file with a .decl extension. Object files may then be loaded into an executable *a.out* file. Exactly one object file must supply a program module to create an executable *a.out* file successfully. The rest of the files must consist only of implementation and definition modules which comprise separate modules within the program.

Object files created by other language processors may be loaded together with object files created by *mod2*. The functions and procedures they define must have been declared in .def files imported by all the .mod files which call those routines.

The following options have the same meaning as in *cc*(1), *pascal*(1) and *f77*(1). See *ld*(1) for load-time options.

-c Suppress loading and produce '.o' file(s) from source file(s).

-g Have the compiler produce additional symbol table information for *sdb*(1).

-gx Have the compiler produce additional symbol table information for *dbx*(1).

-O[G,P] Invoke an object code optimiser. The **–O** option, which is equivalent to **–OG**, performs both global and peephole optimisations. The **–OP** option performs peephole optimisations only. The global optimiser significantly increases compile time and, under normal circumstances, should only be invoked when development of the program is complete. The global optimiser assumes a single process and should not be used on programs that use forks, signals, or shared memory.

-p Prepare object files for profiling, see *prof*(1).

-S Compile the named program, and leave the assembler-language output on the corresponding file suffixed '.s'. (No '.o' is created.).

-o *output* Name the final output file *output* instead of *a.out*.

-I *dir* '.decl' files are always sought first in the directory of the *file* argument, then in directories named in **–I** options, then in directories on a standard list.

The following options are peculiar to *mod2*:

-l Make a program listing during translation including error messages.

-q Prompt for .sym files

-v Verbose option.

-D Force generation of .decl file.

-d* Set debugging level.

-n Don't generate .decl file.

Other arguments are taken to be loader option arguments, perhaps libraries of *modula* compatible routines.

FILES

file.mod	modula source files
file.def	definition module source file
mod2	control program (shell script)
/u2/modula/bin/mtc	
/u2/modula/bin/m2c	compilers
/bin/cc	C compiler
/bin/ld	link editor
/u2/modula/Lib/libmodula.a	intrinsic functions and I/O library
/lib/libc.a	standard library, see *intro*(3).
/u2/modula/Include/{Decl,sym,def}	definition modules for system library.

BUGS

Definition modules must be processed before the corresponding implementation module or any module that imports it.
There are a lot more files generated than those given above; some of them have to be retained between compiling the definition module and the implementation module.

APPENDIX D
ERROR MESSAGES

The following is a list of the error messages generated by *mtc* in a source listing, if it is requested.

```
 0  illegal character
 1
 2  constant out of range
 3  open comment at end of file
 4  string terminator not on this line
 5  too many errors
 6  string too long
 7  too many identifiers (identifier table full)
 8  too many identifiers (hash table full)

20  identifier expected
21  integer constant expected
22  ']' expected
23  ';' expected
24  block name at the END does not match
25  error in block
26  ':=' expected
27  error in expression
28  THEN expected
29  error in LOOP statement
30  constant must not be CARDINAL
31  error in REPEAT statement
32  UNTIL expected
33  error in WHILE statement
34  DO expected
35  error in CASE statement
36  OF expected
37  ':' expected
38  BEGIN expected
39  error in WITH statement
40  END expected
41  ')' expected
42  error in constant
43  '=' expected
44  error in TYPE declaration
45  '(' expected
46  MODULE expected
47  QUALIFIED expected
48  error in factor
49  error in simple type
50  ',' expected
51  error in formal type
52  error in statement sequence
53  '.' expected
54  export at global level not allowed
55  body in definition module not allowed
```

```
 56  TO expected
 57  nested module in definition module not allowed
 58  '}' expected
 59  '..' expected
 60  error in FOR statement
 61  IMPORT expected

 70  identifier specified twice in import-list
 71  identifier not exported from qualifying module
 72  identifier declared twice
 73  identifier not declared
 74  type not declared
 75  identifier already declared in module environment
 76
 77
 78  value of absolute address must be of type CARDINAL
 79  scope table overflow in compiler
 80  illegal priority
 81  definition module belonging to implementation not found
 82  structure not allowed for implementation of hidden type
 83  procedure implementation different from definition
 84  not all defined procedures or hidden types implemented
 85
 86  incompatible versions of symbolic modules
 87
 88  function type is not scalar or basic type
 89
 90  pointer-referenced type not declared
 91  tag-fieldtype expected
 92  incompatible type of variant-constant
 93  constant used twice
 94  arithmetic error in evaluation of constant expression
 95  range not correct
 96  range only with scalar types
 97  type-incompatible constructor element
 98  element value out of bounds
 99  set-type identifier expected
100
101  undeclared identifier in export-list of the module
102
103  wrong class of identifier
104  no such module name found
105  module name expected
106  scalar type expected
107  set too large
108  type must not be INTEGER or CARDINAL
109  scalar or subrange type expected
110  variant value out of bounds
111  illegal export from program module
112  code block for modules not allowed
```

```
120   incompatible types in conversion
121   this type is not expected
122   variable expected
123   incorrect constant
124   no procedure found for substitution
125   unsatisfying parameters of substituted procedure
126   set constant out of range
127   error in standard procedure parameters
128   type incompatibility
129   type identifier expected
130   type impossible to index
131   field not belonging to a record variable
132   too many parameters
133
134   reference not to a variable
135   illegal parameter substitution
136   constant expected
137   expected parameters
138   BOOLEAN type expected
139   scalar types expected
140   operation with incompatible type
141   only global procedure or function allowed in expression
142   incompatible element type
143   type incompatible operands
144   no selectors allowed for procedures
145   only function call allowed in expression
146   arrow not belonging to a pointer variable
147   standard function or procedure must not be assigned
148   constant not allowed as variant
149   SET type expected
150   illegal substitution to WORD parameter
151   EXIT only in LOOP
152   RETURN only in PROCEDURE
153   expression expected
154   expression not allowed
155   type of function expected
156   integer constant expected
157   procedure call expected
158   identifier not exported from qualifying module
159   code buffer overflow
160   illegal value for code
161   call of procedure with lower priority not allowed

200   compiler error
201   implementation restriction
202   implementation restriction: for step too large
203   implementation restriction: boolean expression too long
204   implementation restriction: expression stack overflow,
      i.e. expression too complicated or too many parameters
205   implementation restriction: procedure too long
206   implementation restriction: packed element used for var parameter
```

```
207   implementation restriction: illegal type conversion

220   not further specified error
221   division by zero
222   index out of range or conversion error
223   case label defined twice
```

APPENDIX E
EBNF NOTATION FOR *symbol* INPUT

E1. SYNTAX FOR "*.obj*" FILE.

E1.1. Syntax

```
1   Unit =            Header { SymbolModule } ENDFILESS .
2   Header =          SymFile ModuleKey DefModName .
3   SymFile =         Value .                          / symbol file syntax version /
4   Value =           NORMALCONSTSS Number .
5   ModuleKey =       Value Value Value .              / compilation time stamp /
6   DefModName =      Ident .                          / name of compiled definition module /
7   Ident =           ( IDENTSS | SYMBOLSS ) Spix .
8   SymbolModule =    UNITSS ModuleKey Ident
9                      [ IMPORTSS { [UNITSS] QualIdent } ]
10                     [ EXPORTSS { QualIdent } ]
11                     { Definition } ENDUNITSS .
12  Definition =      CONSTSS { ConstDeclaration } |
13                    TYPESS { TypeDeclaration } |
14                    PROCSS ProcedureHeading { Definition } ENDUNITSS |
15                    VARSS { VarDeclaration } |
16                    MODSS QualIdent [ EXPORTSS { QualIdent } ]
17                     { Definition } ENDUNITSS .
18  ConstDeclaration = QualIdent Constant .
19  Constant =        Value QualIdent | RealConst | StringConst .
20  QualIdent =       Ident { PERIODSS Ident } .
21  RealConst =       REALCONSTSS RHigh RLow .
22  RHigh =           Number .                         / upper part of real number /
23  RLow =            Number .                         / lower part of real number /
24  StringConst =     STRINGCONSTSS { Character } '0C' .
25  TypeDeclaration = QualIdent [ OPAQUESS ] Type .
26  Type =            SimpleType | HIDDENTYPSS | ArrayType |
27                    RecordType | SetType | PointerType |
28                    ProcType .
29  SimpleType =      [ STRUCTTYPSS ] QualIdent | Enumeration | Subrange .
30  Enumeration =     LPARENTSS { QualIdent Value } RPARENTSS .
31  Subrange =        LBRACKETSS Constant RANGESS Constant RBRACKETSS .
32  ArrayType =       ARRAYTYPSS SimpleType OFSS Type .
33  RecordType =      RECORDTYPSS Fields { Fields } ENDSS .
34  Fields =          Ident COLONSS Type |
35                    CASESS [ Ident ] COLONSS Type Ident OFSS Variant
36                     { OFSS Variant } [ ELSESS Ident Fields { Fields } ]
37                    ENDSS.
38  Variant =         Ident CaseLabelList COLONSS Fields { Fields }.
39  CaseLabelList=    CaseLabel { CaseLabel }.
40  CaseLabel =       Value.
41  SetType =         SETTYPSS SimpleType .
42  PointerType =     POINTERTYPSS Type .
43  ProcType =        PROCSS LPARENTSS
44                     { [ VARSS ] [ Ident COLONSS ] [ POINTERTYPSS ]
```

```
45                          QualIdent }
46                          RPARENTSS [ COLONSS QualIdent ] .
47  ProcedureHeading =      QualIdent ProcNum ProcType .
48  ProcNum =               Value .
49  VarDeclaration =        QualIdent [ AbsAddr ] COLONSS Type .
50  AbsAddr =               LBRACKETSS Value RBRACKETSS .
51  Spix =                  Pointer in ASC file .
```

Again if Spix < 0 then internally generated name.

The following is a list of **terminal** and **non-terminal** symbols, followed by the line numbers on which the symbol appears. The line number on which the symbol is defined is precided by '-'.

E1.2. Non-terminal Symbols

```
AbsAddr          -50  49
ArrayType        -32  26
CaseLabel        -40  39
CaseLabelList     38
Character         24
ConstDeclaration -18  12
Constant          31  -19  18
DefModName        -6   2
Definition        17   14  -12  11
Enumeration      -30  29
Fields            38   36  -34  33
Header            -2   1
Ident             44   38  36   35  34  20  8   -7  6
ModuleKey         8    -5  2
Number            23   22  4
PointerType      -42  27
ProcNum          -48  47
ProcType          47   -43 28
ProcedureHeading -47  14
QualIdent         49   47  46   45  30  29  25  -20 19  18  16  10  9
RHigh            -22  21
RLow             -23  21
RealConst        -21  19
RecordType       -33  27
SetType          -41  27
SimpleType        41   32  -29  26
Spix             -51  7
StringConst      -24  19
Subrange         -31  29
SymFile           -3   2
SymbolModule      -8   1
Type              49   42  35   34  32  -26 25
TypeDeclaration  -25  13
Unit              -1
Value             50   48  30   19  5   -4  3
VarDeclaration   -49  15
```

Variant	-38	36	35			

E1.3. Terminal Symbols

ARRAYTYPSS	32					
CASESS	35					
COLONSS	49	46	44	38	35	34
CONSTSS	12					
ELSESS	36					
ENDFILESS	1					
ENDSS	33					
ENDUNITSS	17	14	11			
EXPORTSS	16	10				
HIDDENTYPSS	26					
IDENTSS	7					
IMPORTSS	9					
LBRACKETSS	50	31				
LPARENTSS	43	30				
MODSS	16					
NORMALCONSTSS	4					
OFSS	36	35	32			
OPAQUESS	25					
PERIODSS	20					
POINTERTYPSS	44	42				
PROCSS	43	14				
RANGESS	31					
RBRACKETSS	50	31				
REALCONSTSS	21					
RECORDTYPSS	33					
RPARENTSS	46	30				
SETTYPSS	41					
STRINGCONSTSS	24					
STRUCTTYPSS	29					
SYMBOLSS	7					
TYPESS	13					
UNITSS	8					
VARSS	44	15				

APPENDIX F
EBNF NOTATION FOR *modula* INPUT

F1. SYNTAX FOR "*.dfn*" AND "*.mtc*" FILES.

F1.1. Syntax

```
 1  Unit =          [ IMPLEMENTATION ] Module EOP .
 2  Module =        MODULESY NptrList { Definition } ENDBLOCK .
 3  Definition =    Module | Procedure .
 4  Procedure =     PROCEDURESY NptrList Block .
 5  Nptr =          Spix .
 6  NptrList =      Nptr { FROMSY Nptr } .
 7  Block =         { Definition } [BEGINSY StatSequence] ENDBLOCK .
 8  StatSequence =  { Statement } .
 9  Statement =     BECOMES Variable COMMA Expression [ VARSY | CONSTSY ] |
10                  CALL Variable ParamList |
11                  IFSY Expression StatSequence
12                    { ELSIFSY Expression StatSequence }
13                    [ ELSESY StatSequence ] ENDSY |
14                  FORSY Variable COMMA Expression TOSY Expression
15                    [ BYSY Constant ] StatSequence ENDSY |
16                  CASESY Expression
17                    { OFSY { Element } COLON StatSequence }
18                    [ ELSESY StatSequence ] ENDSY |
19                  WHILESY Expression StatSequence ENDSY |
20                  REPEATSY StatSequence UNTILSY Expression |
21                  LOOPSY StatSequence ENDSY |
22                  RETURNSY [ LPARENT Expression RPARENT ] |
23                  EXITSY |
24                  WITHSY Variable COLON Name StatSequence ENDSY .
25  Expression =    SimpleExpr [ RelOp SimpleExpr ] .
26  RelOp =         EQL | NEQ | GRT | GEQ | LSS | LEQ | INSY .
27  SimpleExpr =    [ MINUS ] Term { AddOp Term } .
28  AddOp =         ( PLUS | MINUS | ORSY ) [ SETOP ] .
29  Term =          Factor { MulOp Factor } .
30  MulOp =         ( TIMES | SLASH | DIVSY | MODSY | ANDSY ) [ SETOP ] .
31  Factor =        Constant |
32                  [ ADDRESSY ] Variable [ [ CastOption ] ParamList ] |
33                  LPARENT Expression RPARENT |
34                  NOTSY Factor .
35  Variable =      [ FIELD FieldLevel PERIOD ] Name
36                    { LBRACK Expression RBRACK |
37                      PERIOD Name | ARROW } .
38  FieldLevel =    Number .
39  Constant =      ANYCON TypeStruct .
40  ParamList =     LPARENT [ [VARSY] Expression
41                    { COMMA [VARSY] Expression } ] RPARENT .
42  Element =       Constant .
43  Value =         Number .
44  TypeStruct =    INTCON Value |
```

```
45                      CARDCON Value |
46                      INTCARCON Value |
47                      CHARCON Value |
48                      REALCON RealConst |
49                      STRINGCON StringConst |
50                      SETCON Value .
51  Name =              [ MODULESY Nptr ] [ Union ] ( NAMESY | SYMBOLSY )
52                       NptrList .
53  CastOption =        TYPESY |
54                      RECORDSY |
55                      ARRAYSY .
56  Spix =              Pointer in ASC file .
57  RealConst =         RHigh RLow .
58  RHigh =             Value .
59  RLow =              Value .
60  StringConst =       { Character } '0C' .
61  Union =             RECORDSY Nptr Nptr { RECORDSY Nptr Nptr }.
```

If Spix < 0 then the identifier is internally generated. Use the two bytes as an identifier name.

The following is a list of **terminal** and **non-terminal** symbols, followed by the line numbers on which the symbol appears. The line number on which the symbol is defined is precided by '-'.

F1.2. Non-terminal Symbols

```
AddOp          -28   27
Block           -7    4
CastOption     -53   32
Character       60
Constant        42   -39   31   15
Definition       7    -3    2
Element        -42   17
Expression      41   40    36   33   -25   22   20   19   16   14   12   11   9
Factor          34   -31   29
FieldLevel     -38   35
Module           3    -2    1
MulOp          -30   29
Name           -51   37    35   24
Nptr            61   51    6    -5
NptrList        52   -6    4    2
Number          43   38
ParamList      -40   32    10
Procedure       -4    3
RHigh          -58   57
RLow           -59   57
RealConst      -57   48
RelOp          -26   25
SimpleExpr     -27   25
Spix           -56    5
StatSequence    24   21    20   19   18   17   15   13   12   11   -8
Statement       -9    8
```

StringConst	-60	49						
Term	-29	27						
TypeStruct	-44	39						
Union	-61	51						
Unit	-1							
Value	59	58	50	47	46	45	44	-43
Variable	-35	32	24	14	10	9		

F1.3. Terminal Symbols

ADDRESSY	32						
ANDSY	30						
ANYCON	39						
ARRAYSY	55						
ARROW	37						
BECOMES	9						
BYSY	15						
CALL	10						
CARDCON	45						
CASESY	16						
CHARCON	47						
COLON	24	17					
COMMA	41	14	9				
CONSTSY	9						
DIVSY	30						
ELSESY	18	13					
ELSIFSY	12						
ENDBLOCK	7	2					
ENDSY	24	21	19	18	15	13	
EOP	1						
EQL	26						
EXITSY	23						
FIELD	35						
FORSY	14						
FROMSY	6						
GEQ	26						
GRT	26						
IFSY	11						
IMPLEMENTATION	1						
INSY	26						
INTCARCON	46						
INTCON	44						
LBRACK	36						
LEQ	26						
LOOPSY	21						
LPARENT	40	33	22				
LSS	26						
MINUS	28	27					
MODSY	30						
MODULESY	51	2					
NAMESY	51						

NEQ	26		
NOTSY	34		
OFSY	17		
ORSY	28		
PERIOD	37	35	
PLUS	28		
PROCEDURESY	4		
RBRACK	36		
REALCON	48		
RECORDSY	61	54	
REPEATSY	20		
RETURNSY	22		
RPARENT	41	33	22
SETCON	50		
SETOP	30	28	
SLASH	30		
STRINGCON	49		
SYMBOLSY	51		
TIMES	30		
TOSY	14		
TYPESY	53		
UNTILSY	20		
VARSY	9		
WHILESY	19		
WITHSY	24		

APPENDIX G
INSTALLATION DETAILS

G1. PROCEDURE FOR INSTALLING THE MODULA-2 TO C TRANSLATOR

The source of the Modula-2 translator is distributed on a single file using the UNIX tape archiver utility *tar*.

To install mod2 do the following:

(a) Unpack the distribution file into some directory.

(b) Edit Makefile to suit your system (*i.e.* BIN, LIB, INCLUDE & OWNER - see explanation in Makefile)

(c) In the ucb universe type 'make first'

(d) Test the program (this is a bit difficult as the bits are spread over 4 directories).

(e) type 'make install'

Notes:

i) If you are looking for the makefiles, then they either have the name Makefile or makefile, depending on what is most readable in the particular directory.

ii) There is nothing really universe dependent, except for the support of very long names in both files and identifiers. However, it was originally written in the ucb universe and has not been extensively tested in the att universe.

One suggestion is that you first install the system in some local area to test it before inflicting it on the system. To regenerate the code type 'make all' (or just make).

The distribution contains a number of directories:

Src - the source to the translator. This contains the following directories:

- Mtc - Pass 1-3 of the compiler (written in Modula-2, with equivalent C code).
- Modula - the translator for mtc's object code.
- Symbol - the translator for mtc's symbol table.
- M2C - a program to merge the outputs from modula and symbol.
- Sys - a number of system routines.
- Utils - a couple of useful programs (including a shell script to compile modula programs).

Docs - some documentation. Presently rather scarce. It should contain a man page for mod2 (the shell script in Src/Utils), a copy of a paper for AUUG and documentation from the original compiler (*i.e.* from ETH). Read m2c.auug and user.guide first.

Test - some test programs that are known to work (including the equivalent C source).

Pretty.prt - a program to format Modula listings - as yet untested.

Other files are:

README - this file,

BUGS - a list of known bugs (and possible fixes),

Makefile - a top level make file.

On distribution any translator sources that are written in Modula-2 should also have the equivalent C code.

During installation the translator will make a number of directories if they do not already exist. These are

$BIN	- the place for the programs to go from make install (*e.g.* /usr/bin or /usr/local/bin)
$LIB	- where the library for ld goes (*e.g.* /usr/lib or /usr/local/lib)
$INCLUDE	- a home for some other subdirectories (*e.g.* /usr/lib/modula or /usr/include/modula)
$INCLUDE/Decl	- all the '.decl' equivalent of the Modula-2 '.def' are stored here for system modules. Used by cc.
$INCLUDE/sym	- symbol files for system modules ('.sym' from '.def') Used by mtc.
$INCLUDE/def	- the '.defs' used to generate '.def' and '.sym'. Mainly for ease of reference (nothing uses them).