SRC Modula-3
Version 2.07

Bill Kalsow
Eric Muller

Systems Research Center
Digital Equipment Corporation
130 Lytton Avenue
Palo Alto, CA 94301-1044

released: June 15, 1992
manual update: July 20, 1992

Contents

Chapter 1

Introduction

This document describes SRC Modula-3 and the terms under which it is distributed.

The distribution contains a Modula-3 compiler and runtime, a set of libraries, a coverage analyzer, a Modula-3 pretty printer, and a small test suite of Modula-3 programs. The compiler generates C as intermediate code.

This release is known to work on a variety of machines (see the table on page 6). We have not tested the software in any other configurations. It may function correctly on other versions of Ultrix or on other machines.

The compiler and runtime system was designed and implemented by Bill Kalsow and Eric Muller. Neither of us view this as a finished product. Nonetheless, we thought others might like to use it. The system should be of interest to two camps: those interested in trying out Modula-3 and those interested in compiler hacking.

Other Documents

The bibliography at the end of this document contains some references related to Modula-3.

The Modula-3 language is described in "Systems Programming with Modula-3" [14], edited by Greg Nelson and published by Prentice Hall. It should be available in book stores. Other chapters in this book describe the thread mechanism and readers and writers.

Sam Harsion wrote "Modula-3" [9] and "Safe Programming with Modula-3" [10], overviews of Modula-3 and "Modula-3" [11], a textbook for Modula-3.

To receive a SRC report on paper, contact:

SRC Report Distribution
Digital Equipment Corporation
130 Lytton Avenue
Palo Alto, CA 94301-1044

`src-report@src.dec.com`

Acknowledgments

Many people contributed to SRC Modula-3, and we would like to thank them. Below is a partial list of the contributors.

We use the garbage collector developed by **Joel Bartlett** (DEC-WRL).

John Dillon (DEC-SRC) provided the original C version of thread switching.

Mark R. Brown and **Greg Nelson** (DEC-SRC) designed the readers and writers interfaces.

Jorge Stolfi (DEC-SRC) and **Stephen Harrison** (DEC-WSE) were very patient alpha-testers. They gave us invaluable bug reports and also translated some DEC-SRC Modula-2+ modules to Modula-3.

Jérôme Chailloux (ILOG) developed the X interfaces while visiting DEC-SRC. We also had numerous discussions about the evolution of SRC Modula-3.

The "gatekeepers" (DEC-WRL), in particular **Paul Vixie**, helped with the distribution of SRC Modula-3.

David Goldberg (XEROX PARC) ported SRC Modula-3 to the SPARC machines.

Ray Lischner ported the system to the APOLLO machines.

Richard Orgass (IBM Rochester) ported the system to the IBM machines.

Piet van Oostrum (Utrecht University) ported the system to the HP series 9000/300 computers running HP/UX 7.0.

Pat Lashley (KLA Instruments) contributed the lexer for `pps`.

Régis Crelier (ETH) designed and implemented the pickles modules while he was a summer intern at SRC.

Mick Jordan (DEC-SRC) provided challenging programs to compile.

Norman Ramsey (Princeton University) has pushed the system into obscure corners and found many bugs there.

R.J. Stroud and **Dick Snow** (University of Newcastle upon Tyne) provided the Encore Multimax port.

Dave Nichols (Xerox PARC) fixed and improved the pretty printer.

Greg Nelson and **Mark Manasse** (DEC-SRC) designed and implemented the Trestle window system.

Sam Harbison contributed the `fieldlist` interface.

Steven Pemberton (CWI) wrote the `enquire` program and made it available to the community.

The vbtkit software has been designed and implemented by a large number of people at SRC: **Andrew Birrell**, **Ken Brooks**, **Marc H. Brown**, **Mark R. Brown**, **Pat Chan**, **Luca Cardelli**, **John DeTreville**, **Steve Glassman**, **Mark Manasse**, **Jim Meehan**, **Greg Nelson**, **Jorge Stolfi**, **Mary-Claire van Leunen**.

FormsVBT is due to **Jim Meehan** and **Marc H. Brown** (DEC-SRC).

Thanks also to all the people who used SRC Modula-3 and reported bugs.

The various ports would have been impossible without the work of a number of people, who kindly made their modifications available. However, most of the bugs you may find in these ports were introduced during the final integration of these modifications and we are to be blamed for them.

Chapter 2

License

SRC Modula-3 is distributed under the terms of this license:

SRC Modula-3 Non-commercial License

SRC Modula-3 is distributed by Digital Equipment Corporation ("DIGITAL"), a corporation of the Commonwealth of Massachusetts. DIGITAL hereby grants to you a non-transferable, non-exclusive, royalty free worldwide license to use, copy, modify, prepare integrated and derivative works of and distribute SRC Modula-3 for non-commercial purposes, subject to your agreement to the following terms and conditions:

- The SRC Modula-3 Non-commercial License shall be included in the code and must be retained in all copies of SRC Modula-3 (full or partial; original, modified, derivative, or otherwise):
- You acquire no ownership right, title, or interest in SRC Modula-3 except as provided herein.
- You agree to make available to DIGITAL all improvements, enhancements, extensions, and modifications to SRC Modula-3 which are made by you or your sublicensees and distributed to others and hereby grant to DIGITAL an irrevocable, fully paid, worldwide, and non-exclusive license under your intellectual property rights, including patent and copyright, to use and sublicense, without limititation, these modifications.
- SRC Modula-3 **is a research work which is provided "as is", and DIGITAL disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness of purpose. In no event shall DIGITAL be liable for any special, direct, indirect, or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.**

NOTICE: There is also a commercial license. By signing and returning it, further rights to use and distribute SRC Modula-3 are granted. This license is in `doc/agreement.ps`.

Chapter 3

History

Version 2.0 implements the twelve language changes (i.e. generics, IEEE floating point interfaces, ...) that are included in [14]. Version stamp checking was moved into the `m3` driver, which also supports `-make` mode and generates enough type declarations to make debugging tolerable. The compiler internals were rearranged and many bugs were removed. Better code is produced.

Version 1.6 fixes many bugs that have been reported. It also introduces the `SUN3`, `UMAX` and `ARM` architectures. Some Unix interfaces have been added or modified (`Usocket`, `Udir`, `Uexec`, `Uerror`). The names in the `Rd` and `Wr` interfaces are now more coherent. The new `Pkl` interface allow input/output of binary data structures. The runtime has been rewritten to be mostly in Modula-3; this allows for clean interfaces to the runtime; some limitations have been removed (profiling; scheduling). The driver has been rewritten, so as to support shared libraries (on `IBMR2`, by default); the syntax of some options has changed.

Version 1.5 supports five new architectures (`AP300`, `AIX386`, `IBMR2`, `IBMRT` and `HP300`). The driver has been modified to improve portability of user systems. The SRC Modula-3 libraries have been reorganized, and of course known bugs have been fixed. New demonstration programs and games are included.

Version 1.4 is the second public release of SRC Modula-3. It uses the new features of version 1.3 and was alpha-tested by several SRC clients. This version added `<*UNUSED*>` and `<*OBSOLETE*>` pragmas, simplified coverage profiling by having the compiler directly generate the counters, reduced the number of `#line` directives in the generated C, added "map" procedures so that the garbage collector can efficiently locate global references, packed enumerations into smaller C types, and fixed several bugs.

Version 1.3 is for internal use only. This version serves to snapshot the massive editing that has taken place since 1.2. This version fixed the variable renaming problems, made `TEXT` a `REF ARRAY OF CHAR`, converted the text implementation to Modula-3, passed nested procedures as closures, used C initialization where possible for constants and variables, added warning messages, and fixed many bugs.

Version 1.2 Thanks to the new technology introduced in 1.1, porting the compiler to other machines is much easier. We have ported it to DECstation 3100 running Ultrix 3.1. A few bugs have been fixed. The driver processes the options `-D` and `-B` in a slightly different way.

The installation procedure is new, and we no longer furnish executables as the intermediate C files are present on the release. Because the intermediate C files vary according to the target machine, there are separate `tar` files for each of the supported machines. However, each distribution contains all of the sources; only the intermediate C files differ.

Version 1.1 This version is for internal use only. The main difference with Version 1.0 is the use of RCS and the use of `imake` rather than the standard `make`.

Version 1.0 This version is the first public release of the SRC Modula-3 system. It contains a Modula-3 compiler and runtime, a core library, a coverage analyzer, a dependency checker, a Modula-3 pretty printer, and a small test suite of Modula-3 programs. The compiler generates C as an intermediate code.

It is known to run on VAX Ultrix 3.1. We have not tested the software in any other configurations. The software may function correctly on other versions of Ultrix, and if recompiled, may even work on other machines.

Chapter 4

Installation

This chapter describes how to get and install the SRC Modula-3 system.

4.1 What is available

SRC Modula-3 is distributed via anonymous `ftp` from `gatekeeper.dec.com`. The distribution is in a set of compressed `tar` files in the directory named `pub/DEC/Modula-3/release`. The files are of the form *archive-version*`.tar.Z`; in the rest of the chapter, we will speak of the archive *archive* and forget the version numbers.

The archives `boot.`*architecture* are used to build and install `m3make`, a driver and a compiler. These programs are built from intermediate C files that are architecture specific; you need to get the archive(s) corresponding to the architecture(s) on which you want to install SRC Modula-3. The supported *architecture*s are:

architecture	Hardware	Operating system	Build disk (KB)	Build cpu (usr+sys) (min)	Install (KB)
AIX386		AIX/PS2			
AP3000	Apollo DN4500	Domain/OS 10.2			
ARM	Acorn R260	RISC iX 1.21			
DS3100	DECstation 5000/200	Ultrix 4.2	23170	8 + 7	2758
HP300	HP 9000/300	HP-UX 7.0			
IBMR2	IBM RISC System/6000	AIX 3.1			
IBMRT	IBM RT	IBM/4.3 (AOS 4.3)			
NEXT	NeXT				
SPARC	Sparcstation-1	SunOS 4.1.x			
SUN3	Sun 3/?	SunOS ?			
UMAX	Encore Multimax	UMAX 4.3 (R4.1.1)			
VAX	VAX 8800	Ultrix 4.2			

Each of these archives is about 4000 kilobytes. The column “Build” indicates the resources you need to build the two programs: “disk” is the amount of disk space (in kilobytes), “cpu” is the amount of user and system cpu time (in minutes). The column “Install” indicates the amount of disk space that will be permanently used after the installation is done.

The other archives contain Modula-3 source files for various libraries and programs.

File Name	Size (KB)	Build disk (KB)	Build cpu (usr+sys) (min)	Install (KB)	Contents
`bicycle`	45				bitmaps of cards for games, needs `trestle`
`compiler`	408	11255	12 + 5	0	compiler sources
`data`	28	2693	2 + 1	856	some generic container types
`demos`	74	10446	4 + 1	0	a few demo programs, needs `trestle` and `bicycle`
`driver`	77			0	driver sources
`doc`	424				the documentation for SRC Modula-3
`dpskit`	112				binding interfaces to Display PostScript
`dpsapps`	12				applications using Display PostScript
`formsvbt`	112				FormsVBT, needs `vbtkit`
`libm3`	2160	20374	8 + 4	5742	base library
`m3make`	68			0	make for Modula-3
`tcl`	696				binding interface to Tcl
`tests`	472			0	"validation" tests
`tools`	272	1746	1 + 0	548	development tools, need `trestle`
`trestle`	2400	13245	14 + 3	4048	Trestle window system, needs `X11R4`
`vbtkit`	416				More VBTs, needs `trestle`
`vbtapps`	248				FormsVBT applications, needs `formsvbt`
`X11R4`	128	2906	4 + 1	1057	binding interfaces to the X11R4 libraries

The column "File Size" is the size (in kilobytes) of the compressed tar file. The colum "Build" indicates the resources you need to build and install these pieces of sources: "disk" is the amount of disk space (in kilobytes), "cpu" is the amount of user and system cpu time (in minutes). The column "Install" indicates the amount of disk space that will be permanently used after the installation is done; if you want to keep the sources around, you will need more space. These time and sizes have been measured on a DECstation 5000/200 running Ultrix 4.2; other architectures may have different requirements.

You need to build and install `libm3` to have a useful system, but all the other pieces are optional.

The m3make and doc archives are also contained in the boot archives.

Large archives are available in one piece (`foo.tar.Z`) as well as in pieces of 512 KB each (`foo.tar.Z-01` and so on). If your connection to `gatekeeper` is slow, you may want to get the smaller pieces and reassemble them in a one piece archive at your site (using `cat` for example).

4.2 Getting SRC Modula-3

In the following, `$` is the shell prompt and `ftp>` is the `ftp` prompt. To get SRC Modula-3:

1. Make sure that you have enough disk space using the tables above.

2. Create a fresh directory for the software and go there. Path names below are relative to that directory, and it will be called the top-level directory:

   `$ mkdir` *top-level*
   `$ cd` *top-level*

3. Open an `ftp` connection with `gatekeeper.dec.com` [16.1.0.2]; give `anonymous` for the name and your login id for the password:

```
$ ftp gatekeeper.dec.com
Connected to gatekeeper.dec.com.
...
Name (gatekeeper.dec.com):  anonymous
Password (gatekeeper.dec.com:anonymous):  your name@your machine
...
```

4. Change to the proper directory:

```
ftp> cd pub/DEC/Modula-3/release
```

5. Set the transmission type to binary:

```
ftp> type binary
```

6. Get the distribution bootstrap:

```
ftp> get boot.architecture-version.tar.Z
```

7. Get `libm3` along with any other Modula-3 sources that you want:

```
ftp> get libm3-version.tar.Z
ftp> get ...
```

8. Close the connection:

```
ftp> quit
```

9. Uncompress and extract the files:

```
$ zcat boot.architecture-*.tar.Z | tar xpof -
$ zcat libm3-*.tar.Z | tar xpof -
$ ...
```

The `tar` arguments specify the following options:

`x`	extract the files from the `tar` file to the current directory
`p`	restore files to their original modes
`o`	override the original ownership, this makes you the owner of the files
`f`	use the following argument (e.g `-`) as the input file; `-` as the input file means `stdin`

You can add the `v` option to see what is going on.

10. At this point you may delete the archives to save space (the disk requirements indicated above assume that you do delete these files):

```
$ rm *.tar.Z
```

4.3 Installation procedure

1. Create a description of your system, `m3make/`*architecture*`/config`, using the file `m3make/model-configs/`*architecture* as a model.

2. Build and install the `m3make` system:

   ```
   $ (cd m3make/architecture; make -f ../src/Makefile all install)
   ```

   You may need to tell your shell that new executables (`m3make`) are present after the install step, using `rehash`, for example.

3. You may want to change chapter 7 of this document to describe your installation (see that chapter to know how to proceed).

4. Build and install SRC Modula-3:

   ```
   $ m3make -f m3makefile.boot all install
   ```

   This moves the driver, the compiler and some other files to the directories specified in `m3make/`*architecture*`/config`. Again, you may need to tell your shell that new executables (`m3`) are present, using `rehash`, for example.

5. At this point, you should have successfully installed the Modula-3 compiler and driver. To check, type

   ```
   $ m3 -\?
   ```

   The driver should list its configuration options.

6. You can now delete the bootstrap directories to conserve space:

   ```
   $ rm -f driver compiler
   ```

   (Note: if you are doing a port, don't do that!)

7. Build and install the other libraries and tools. For each of the archives of Modula-3 source that you copied, starting with `libm3`:

   ```
   $ m3make -f m3makefile.libm3 all install
   $ m3make -f m3makefile.archive all install
   ```

   These libraries and tools will be built using the installed system and should help detecting problems in the installation. Note that the `compiler`, `driver`, `doc` and `m3make` packages shouldn't need to be recompiled, they are the same Modula-3 source that produced the C code for bootstrap.

4.4 Running the tests

SRC Modula-3 includes a collection of test programs. While these programs are intended to help the developers of SRC Modula-3, you may want to look at them or run them. The tests are available in the archive `tests`. If you're interested, see the `README` file at the top-level of that archive.

Chapter 5

How to use the system

This section describes each of the tools in the SRC Modula-3 distribution and how to use them. Briefly, the tools include a compiler, a linker, a pretty printer, and a line-based profiler. See also chapter 7 for tools and hints that are local to your installation.

5.1 Compiling

To compile a Modula-3 program, invoke `m3`(1). This driver is much in the style of `cc`(1); the output is an object file or an executable program, according to the options.

`m3` parses the command line and invokes the compiler and linker as required. `m3` tells the compiler where to seek imported interfaces and where to find the Modula-3 runtime library. Arguments ending in `.m3` or `.i3` are assumed to name Modula-3 source files to be compiled. Arguments ending in `.mo`, `.io` or `.o` are assumed to name object files, possibly created by other language processors, that are to be linked with the object files created by `m3`. Arguments ending in `.mc`, `.ic`, or `.c` are assumed to name C source files to be compiled. Arguments ending in `.ms`, `.is`, or `.s` are assumed to name assembly language files to be translated into object files by the assembler. Arguments starting with `-` specify compiler options. Other arguments are assumed to name library files, and are simply passed along to the linker.

The source for a module named `M` is normally in a file named `M.m3`. The source for an interface named `I` must be in a file named `I.i3`. The main program is the module that implements the interface `Main`.

There are options to compile without linking, stop compiling after producing C, emit debugger symbols, generate profiling hooks, retain intermediate files, override search paths, select non-standard executables for the various passes, and pass arguments to individual passes. For the full details, see the `m3`(1) man page.

In a source file, an occurrence of `IMPORT Mumble` causes the compiler to seek an interface named `Mumble`. The compiler will step through a sequence of directories looking for the file `Mumble.i3`. It will parse the first such file that it finds, which is expected to contain an interface named `Mumble`. If no file `Mumble.i3` exists, or if the parse fails, the compiler will generate an error. The particular sequence of directories to be searched is determined by the options passed to `m3`. See the `m3(1)` manual page for full details.

5.2 An example

Here's a simple program composed of a main module, an imported interface and its implementation.

In the file `Main.m3`:

```
MODULE Main;
IMPORT A;
BEGIN
  A.DoIt ();
END Main.
```

In the file `A.i3`:

```
INTERFACE A;
PROCEDURE DoIt ();
END A.
```

In the file `A.m3`:

```
MODULE A;
IMPORT Wr, Stdio;

PROCEDURE DoIt () =
   BEGIN
     Wr.PutText (Stdio.stdout, "Hello world.\n");
     Wr.Close (Stdio.stdout);
   END DoIt;

BEGIN
END A.
```

If SRC Modula-3 is installed correctly, the command

```
m3 -make -why -o hello Main.m3 A.m3 A.i3
```

will compile the three compilation units and link them with the standard libraries. The result will be left in the executable file named `hello`.

5.3 Makefiles

Once installed, SRC Modula-3 provides `m3make`, a slightly enhanced version of plain `make`. The primary benefit provided by `m3make` is that the operational description found in most `makefiles` is replaced by a more declarative one. The result is that `makefiles` are smaller, simpler, and more portable. You're not required to use `m3make`, but we believe you will like it.

The `m3makefile` for the example above would be:

```
implementation (Main)
module (A)
program (hello)
```

See the `m3make` manpage for full details.

5.4 Language restrictions

With a few exceptions, SRC Modula-3 implements the Modula-3 language as defined in "Systems Programming with Modula-3" ([14]).

Arithmetic checking.

SRC Modula-3 does not generate any special checking for integer arithmetic overflow or underflow. You get whatever checking your C compiler gives you. We decided that the runtime checking was too expensive in a compiler that was constrained to produce C. Depending on your machine, the `FloatMode` interface may be used to control floating point exceptions.

Packed types.

Packed types are restricted. `BITS n FOR T` is treated as `T` everywhere except when applied to a field in a record. In that case, the field is implemented by a *bitfield* of width `n` in a C `struct`. Otherwise, a Modula-3 field is implemented as a *member* of a C `struct`. Consequently, Modula-3 types that would require the C field to span word boundaries are not accepted by SRC Modula-3.

Stack overflow checking.

SRC Modula-3 does not reliably detect thread stack overflows. Stacks are only checked for overflow on procedure entry. No checking is done on external procedures. Thread stacks are allocated in fixed size chunks. The required `Thread` interface has been augmented with the `SizedClosure` type to allow arbitrary sized stacks. The default size can be adjusted with `Thread.MinDefaultStackSize` and `Thread.IncDefaultStackSize`.

Exception semantics.

SRC Modula-3 uses C's `setjmp/longjmp` mechanism to unwind the stack when raising an exception. A problem can occur: assignments may appear to be undone. For example, consider

```
TRY
  i := 3;
  P ();
EXCEPT E:
  j := i;
END;
```

where `P` raises exception `E`. The compiler generates a `setjmp` at the beginning of the try statement. If the C compiler allocates variable `i` to a register, the assignment of `3` may be lost during the `longjmp` and branch that gets to the handler.

Method constants.

The language definition says that if `T` is an object type and `m` one of its methods, `T.m` denotes the procedure value that implements that method and that this value is a constant. In SRC Modula-3, `T.m` denotes the correct procedure constant, but since the compiler generates runtime code to locate the method, some uses of the constant that the `C` compiler must resolve at link time will cause `C` errors. For example,

```
CONST P = T.m;  BEGIN P (...) ...
```

will work, since no initialized `C` storage is allocated for `P`. But the following generates initialized storage and will fail

```
CONST X = ARRAY [0..2] OF Proc { T.m, ..};
```

Similarly, although Modula-3 allows it, the following cannot be evaluated at compile time

```
CONST X = (T.m = MyProcedure);
```

5.5 Pragmas

SRC Modula-3 recognizes the pragmas described below.

<*EXTERNAL*>

The pragma `<*EXTERNAL N:L*>` may precede an interface or a procedure or variable declaration in an interface. It asserts that the following entity is named "N" and implemented in language "L". If "N" is omitted, the external name is the Modula-3 name. The default and only recognized value for "L" is `C`. The ":" is only required when specifying "L". "N" and "L" may be Modula-3 identifiers or string literals.

The names of external procedures and variables are passed through to the C compiler unchanged. *The types of external variables, the types of formal parameters, the types of results, and the raises clauses of external procedures are all assumed to be correct and are not checked against their external implementation.* Standard calling conventions are used when calling external procedures.

Beginning an interface with `<*EXTERNAL*>` declares all of the procedures and variables in that interface external.

For example:

```
<*EXTERNAL*> INTERFACE OS;
VAR errno: INTEGER;
PROCEDURE exit (i: INTEGER);
END OS.
```

allows importers of `OS` to access the standard UNIX symbols `errno` and `exit` through the names `OS.errno` and `OS.exit` respectively.

Alternatively, the following interface provides access to the same two symbols, but uses a more conventional Modula-3 name for the procedure:

```
INTERFACE OS;
<*EXTERNAL errno:C *> VAR errno: INTEGER;
<*EXTERNAL exit:C  *> PROCEDURE Exit (i: INTEGER);
END OS.
```

If several variables are declared within a single `<*EXTERNAL*> VAR` declaration, they are all assumed to be external.

The external pragma may optionally specify a name different from the Modula-3 name. For example:

```
INTERFACE Xt;
  <*EXTERNAL "_XtCheckSubclassFlag" *>
  PROCEDURE CheckSubclassFlag (...);
  ...
```

defines a procedure named `Xt.CheckSubclassFlag` in Modula-3 and named `_XtCheckSubclassFlag` in the generated C.

<*INLINE*>

The pragma <*INLINE*> may precede a procedure declaration. The pragma is allowed in interfaces and modules. SRC Modula-3 recognizes but ignores this pragma.

For example:

```
INTERFACE X;
<*INLINE*> PROCEDURE P (i: INTEGER);
<*INLINE*> PROCEDURE Q ();
END X.
```

declares X.P and X.Q to be inlined procedures.

<*ASSERT*>

The pragma <*ASSERT expr*> may appear anywhere a statement may appear. It is a static error if "expr" is not of type BOOLEAN. At runtime "expr" is evaluated. It is a checked runtime error if the result is FALSE. Assertion checking can be disabled with the -a compiler switch.

<*TRACE*>

The pragma <*TRACE expr*> may appear at the end of any variable or formal declaration. This pragma will generate tracing calls whenever the declared variable is modified.

The "expr" must evaluate to a procedure of two arguments. The first argument is the name of the traced variable, a TEXT. The second argument is the traced variable. Note that any of the formal passing modes may be used with the second argument.

For example:

```
MODULE M;
VAR x: Foo <*TRACE MyTrace.FooChanged*>;
```

will cause

```
MyTrace.FooChanged ("M.x", x)
```

to be generated after each statement that modifies x. Variable aliasing is not tracked, so

```
WITH  alias = x DO  INC(alias) END
```

will not generate any tracing.

The pieces of Modula-3 grammar affected by <*TRACE expr*> are:

```
VariableDecl = IdList (":" Type & ":=" Expr) V_TRACE.
Formal       = [Mode] IdList (":" Type & ":=" ConstExpr) V_TRACE.
ForSt        = FOR Id V_TRACE ":=" Expr TO Expr [BY Expr] DO S END.
Handler      = QualId {"," QualId} ["(" Id V_TRACE ")"] "=>" S.
TCase        = Type {"," Type} ["(" Id V_TRACE ")"] "=>" S.
Binding      = Id V_TRACE "=" Expr.
V_TRACE      = [ "<*" TRACE  Expr "*>" ].
```

The pragma `<*TRACE` stmt-list`*>` may appear immediately after any `BEGIN`. The specified "stmt-list" will be inserted after each statement of the block started by the `BEGIN`. For example:

```
BEGIN <* TRACE  INC(cnt); MyTrace(cnt) *>
  i := j;
  j := i;
END;
```

will generate `INC(cnt); MyTrace(cnt)` after each of the assignment statements.

<*FATAL*>

The pragma `<*FATAL` id-list`*>` may appear anywhere a declaration may appear. It asserts that the exceptions named in "id-list" may be raised, but unhandled in the containing scope. If they are, it's fatal and the program should crash. Effectively, the `<*FATAL*>` pragma disables a specific set of "potentially unhandled exception" warnings. If "id-list" is `ANY`, the pragma applies to all exceptions. The effects of the `<*FATAL*>` pragma are limited to its containing scope — they cannot be imported from interfaces.

For example:

```
EXCEPTION InternalError;
<*FATAL InternalError*>
```

at the top-level of a module `M` means that no warnings will be generated for procedures in `M` that raise but don't list `InternalError` in their `RAISES` clauses.

Similarly,

```
PROCEDURE X() RAISES {} =
BEGIN
  ...
  <*FATAL ANY*> BEGIN
     List.Walk (list, proc);
  END;
  ...
END X;
```

specifies that although `X` raises no exceptions and `List.Walk` may, no warnings should be generated.

<*UNUSED*>

The pragma `<*UNUSED*>` may precede any declaration. It asserts that the entity in the following declaration is not used and no warnings should be generated.

For example, the procedures that implement the default methods for an object may not need all of the actual parameters:

```
PROCEDURE DefaultClose (<*UNUSED*> wr: Wr.T) =
   BEGIN (* do nothing *) END DefaultClose;
```

<*OBSOLETE*>

The pragma <*OBSOLETE*> may precede any declaration (e.g. <*OBSOLETE*> PROCEDURE P ();). A warning is emitted in any module that references an obsolete symbol. This feature is used to warn clients of an evolving interface that they are using features that will disappear in the future.

<*NOWARN*>

The pragma <*NOWARN*> may appear anywhere. It prevents warning messages from being issued for the line containing the pragma. It is probably better to use this pragma in a few places and enable all warnings with the -w1 switch than to ignore all warnings.

<*LINE*>

For the benefit of preprocessors that generate Modula-3 programs, the compiler recognizes a <*LINE ... *> pragma, in two forms:

```
<*LINE number filename *>
<*LINE number *>
```

where number is an integer literal and filename is a text literal. This pragma causes the compiler to believe, for purposes of error messages and debugging, that the line number of the following source line is number and that the current input file is filename. If filename is omitted, it is assumed to be unchanged. <*LINE ... *> may appear between any two Modula-3 tokens; it applies to the source line following the line on which it appears. Here's an example: <*LINE 32 "SourceLoc.nw" *>.

<*PRAGMA*>

The pragma <*PRAGMA id-list*> may appear anywhere. It notifies the compiler that pragmas beginning with the identifiers in "id-list" may occur in this compilation unit. Since the compiler is free to ignore any pragma, the real effect of <*PRAGMA*> is to tell the compiler that pragmas it doesn't implement are coming, but they shouldn't cause "unrecognized pragma" warnings.

5.6 Linking

SRC Modula-3 requires a special two-phase linker. You must link Modula-3 programs with m3.

The first phase of the linker checks that all version stamps are consistent, generates flat struct* declarations for all opaque and object types, and builds the initialization code from the collection of objects to be linked. The second phases calls ld to actually link the program.

The information needed by the first phase is generated by the compiler in files ending in .ix and .mx. Libraries containing Modula-3 code must be created using m3 -a. m3 will combine the .ix and .mx files for the objects in the library into a new file ending in .ax. The .ix, .mx, and .ax files must reside in the same directory as their corresponding .io, .mo and .a files. If m3 encounters a library without a .ax file, it assumes that the library contains no Modula-3 code.

For every symbol X.Z exported or imported by a module, the compiler generates a version stamp. These stamps are used to ensure that all modules linked into a single program agree on the type of X.Z. The linker will refuse to link programs with inconsistent version stamps.

5.7 Runtime arguments

Command line arguments given to Modula-3 programs are divided in two groups. Those that start with the characters `@M3` are reserved for the Modula-3 runtime and are accessible via the `RTParams` interface (we call those the *runtime parameters*). The others are accessible via the `Params`, `ParseParams`, and `RTArgs` interfaces (these are the *program arguments*).

Three runtime parameters are recognized today; others are simply ignored.

- `@M3nogc` turns the garbage collector off.
- `@M3showheap=`*name* activates the logging of heap allocation and garbage collection events. The program forks a process running the *name* program, and sends it these events as they occur. If `=`*name* is ommitted, the `showheap` program is forked (it is part of the `tools` archive); this program displays the status of the heap pages. See its man page for more details.
- `@M3showthread=`*name* activates the logging of thread switching events. The program forks a process running the *name* program, and sends it these events as they occur. If `=`*name* is ommitted, the `showthread` program is forked (it is part of the `tools` archive); this program displays the status of the various threads. See its man page for more details.

5.8 Garbage Collection

A crucial fact for clients of the garbage collector to know is that *objects in the heap move*. If all references to a traced heap object are from other traced heap objects, the collector may move the referent. Hence, it is a bad idea to hash pointer values. References from the stack or untraced heap into the traced heap are never modified.

5.9 Debugging

Since an intermediate step of the Modula-3 compiler is to produce C code, you may use any debugger for which your C compiler can produce debugging information; in most cases, this means `dbx` or `gdb`.

However, this mechanism has limitations: the C compiler generates source-level information that relates the executable program to the intermediate C code, not to the Modula-3 source code. We attempted to reflect as much as possible of the source-level Modula-3 information into the intermediate C code. But there are still some shortcomings that you should know about.

Names

Global names (i.e. top-level procedures, constants, types, variables, and exceptions) are prefixed by their module's name and two underscores. For example, in an interface or module named `X`, the C name of a top-level procedure `P` would be `X__P`. Note, there are two underscores between `X` and `P`.

Local names (e.g. of local variables and formal parameters) are preserved.

The compiler will issue a warning and append an underscore to any Modula-3 name that is a C reserved word.

Types

Modula-3 is based on structural type equivalence, C is not. For this reason, the compiler maps all structurally equivalent Modula-3 types into a single C type. These C types have meaningless names like `_t1fc3a882`. The Modula-3 type names are equated to their corresponding C type. Unfortunately variables are declared with the C type names. So, if you ask your debugger "what is the type of `v`?", it will most likely answer, "`_t13e82b97`". But, if you ask "what is `_t13e82b97`?" it will most likely give you a useful type description.

The table 5.1 indicates the C types corresponding to Modula-3 types.

Despite the fact that the compiler turns all object references into `char*`, the linker generates useful type declarations. These declarations are available under the type's global name. For example, to print an object `o` of type `Wr.T`, type `print *(Wr__T)o`. Note that if `o` was really a subtype of `Wr.T`, say `TextWr.T`, then you must use `print *(TextWr__T)o` to see the additional fields. If the same type appears with two names in a program, the linker arbitrarily picks one.

To print the null terminated string in a variable of type `TEXT` (or `Text.T`) named `txt`, type `print *(char**)txt`.

If you don't know the type of a traced reference, you may be able to use the runtime information to discover it. Given a reference `r`, `print *(_refHeader*)(((char*)r)-4)` will print its typecode `x`, and `print *_types[x]` will print the corresponding typecell. A typecell includes a type's Modula-3 name as a C string (`typecell.name`). If the type doesn't have a Modula-3 name, its internal is the concatenation of "`_t`" and `typecell.selfID` in hex.

File names and line numbers

Due to liberal use of the `#line` mechanism of C, the Modula-3 file names and line numbers are preserved. Your debugger should give you the right names and line numbers and display the correct Modula-3 source code (if it includes facilities to display source code).

Note that uses of the `<*LINE*>` pragma are propagated into the intermediate C code.

Debugger quirks

Most debuggers have a few quirks. `dbx` is no exception. We've found that having a `.dbxinit` file in your home directory with the following contents prevents many surprises:

```
ignore SIGVTALRM
set $casesense = 1
```

The first line tells `dbx` to ignore virtual timer signals. They are used by the Modula-3 runtime to trigger thread preemptions. The second line tells `dbx` that your input is case sensitive.

Procedures

Modula-3 procedures are mapped as closely as possible into C procedures. Two differences exist: "large" results and nested procedures.

First, procedures that return structured values (i.e. records, arrays or sets) take an extra parameter. The last parameter is a pointer to the memory that will receive the returned result. This parameter was necessary because some C compilers return structured results by momentarily copying them into global memory. The global memory scheme works fine until it's preempted by the Modula-3 thread scheduler.

Modula-3	C
enumeration	`unsigned char`, `unsigned short` or `unsigned int` depending on the number of elements in the enumeration.
`INTEGER`	`int`
subrange	`char`, `short` or `int`, possibly `unsigned`, depending on the base type of the subrange. Subranges of enumerations are implemented by the same type as the full enumeration. Subranges of `INTEGER` are implemented by the smallest type containing the range. For example, the type `[0..255]` is mapped to `unsigned char` and `[-1000..1000]` is mapped to `short`.
`REAL`	`float`
`LONGREAL`	`double`
`EXTENDED`	`double`
`ARRAY I OF T`	`struct { tT elts[n] }`, where `tT` is the C type of `T` and `n` is `NUMBER(I)`.
`ARRAY`^{n} `OF T`	`struct { tT* elts; int size[`n`] }`, where `tT` is the C type of `T` and `elts` is a pointer to the first element of the array.
`RECORD ... END`	`struct{ ... }` with the same collection of fields as the original record.
`BITS n FOR T`	Usually `tT` where `tT` is the the C type of `T`. When `T` is an ordinal type and the packed type occurs within a record, it generates a C bit field.
`SET OF T`	`struct { int elts[n] }` where `n` is ⌈`NUMBER (T)/sizeof(int)`⌉.
`REF T` `UNTRACED REF T`	`tT*` where `tT` is the C type of `T`.
`OBJECT ... END`	`ADDRESS`, a `typedef` for `char*` or `void*` (depending on the system) defined in `M3Machine.h`. Each use of an object reference is cast into a pointer of the appropriate type at the point of use.
`PROCEDURE (): T`	Usually `tT *(proc)()` where `tT` is the C type of `T`. If `T` is a record or array, an extra `VAR` parameter is passed to the procedure which it uses to store the return result.

Table 5.1: Type implementations

Second, nested procedures are passed an extra parameter. The first parameter to a nested procedure is a pointer to the local variables of the enclosing block. To call a nested procedure from the debugger, pass the address of the enclosing procedure's local variable named `frame`.

When a nested procedure is passed as a parameter, the address of the corresponding C procedure and its extra parameter are packaged into a small closure record. The address of this record is actually passed. Any call through a formal procedure parameter first checks to see whether the parameter is a closure or not and then makes the appropriate call. Likewise, assignments of formal procedure parameters to variables perform runtime checks for closures.

`<*EXTERNAL*>` procedures have no extra parameters. *except if they return large results??*

Threads

There is no support for debugging threads. That is, there is no mechanism to force the debugger to examine a thread other than the one currently executing. Usually you can get into another thread by setting a breakpoint that it will hit. There is no mechanism to run a single thread while keeping all others stopped.

If your debugger allows you to call procedures in a stopped program, as both `dbx` and `gdb` do, then `print Thread__DumpEverybody()` will produce a table listing the status of all threads.

5.10 Thread scheduling

This version of SRC Modula-3 has a more flexible scheduling algorithm than the previous versions. Here is a rough explanation of its behaviour.

All threads are kept in a circular list. This list is modified only when new threads are created or when threads exit; that is, the relative order of threads in this list is never modified.

When the scheduler comes into action, the list of threads is scanned starting with the thread following the one currently running, until a thread that can execute is found:

- if it was preempted by the scheduler, it can execute
- if it is waiting for a condition or a mutex that is still held, it cannot execute
- if it has blocked because of a call to `Time.Pause` (or a similar procedure), it can execute iff the timeout is now expired
- if it has blocked because of a call to `RTScheduler.IOSelect` (or a similar procedure), it can execute iff the timeout is now expired or a polling `select(2)` returns a non-zero value.

If such a thread is found, it becomes active.

If no thread can execute, and there are no threads blocked in a `Time.Pause` or a `RTScheduler.IOSelect`, a deadlock situation is detected and reported. Otherwise, a combination of the file descriptors sets (OR of all the file descriptors sets) and timeouts (MIN of all the timeouts) is formed, `select(2)` is called with those arguments and the whole process of searching for an executable thread is redone. This ensure that the Unix process does not consume CPU resources while waiting.

The scheduler is activated when the running thread tries to acquire a mutex which is locked, waits for a condition, calls `Time.Pause` (or a similar procedure) with a future time, calls `RTScheduler.IOSelect` (or a similar procedure) with a non-zero valued timeout and no files are ready at the time of the call, or the time allocated to the thread has expired (preemption).

Preemption is implemented using the Unix virtual interval timer. `RTScheduler.SetSwitchingInterval` can be used to change the interval between preemptions. SRC Modula-3 no longer uses the real time interval timer nor the profiling interval timer for thread scheduling; these are available to the program.

Because of the preemption implementation, Unix kernel calls will block the process (i.e. the Unix process sleeps even though some threads could run). However, `Time.Pause` and `RTScheduler.IOSelect` provide functional equivalents of `sigpause(2)` and `select(2)` that do not cause the process to block.

5.11 Profiling

In addition to the usual profiling tools (e.g. see `prof(1)`, `gprof(1)` and `pixie(1)`), SRC Modula-3 provides support for line-based profiling.

To enable collection of data during the execution of programs, give the `-Z` option to the `m3` command for the compilation of the modules you want to examine and also for the linking of the program. To interpret the result, run `analyze_coverage(1)`.

Note that because of the extensive data collection performed by this mode of profiling, the execution time of the program can be significantly larger when it is enabled; thus, simultaneous time profiling can produce erroneous results. For the same reason, the profiling data file is rather large; furthermore, as it is augmented by each execution of the program, you may want to compress it from time to time (see `analyze_coverage(1)` for more details).

5.12 Pretty printing

SRC Modula-3 includes a pretty-printer for Modula-3 programs. It is accessible as `m3pp`(1). Read its man page to find out how to use it.

5.13 Gnuemacs support

There is a mode to edit Modula-3 programs. To use it, you need to put in your `.emacs` file the following lines:

```
(autoload 'modula-3-mode "modula3")
   (setq auto-mode-alist
      (append '(("\\.ig$" . modula-3-mode)
                ("\\.mg$" . modula-3-mode)
                ("\\.i3$" . modula-3-mode)
                ("\\.m3$" . modula-3-mode))
              auto-mode-alist))
```

Your system administrator may have inserted these lines in the default macro files for your system.

There is also a program to build tags file for Modula-3 programs: `m3tags`; see the manpage for the details. When the system is installed, a tag file for the public interfaces is built. To access it, you need in your `.emacs` (or in the system initialization file) the line:

```
(visit-tags-table "LIB_USE/FTAGS")
```

where `LIB_USE` is the place where the Modula-3 libraries have been installed.

5.14 Keeping in touch

`comp.lang.modula3` is a Usenet newsgroup devoted to Modula-3. There you will find discussions on the language and how to use it, annoucements of releases (both of SRC Modula-3 and of other systems). Since not everybody has access to Usenet, we also maintain a relay mailing list, to which we resend the articles appearing in `comp.lang.modula3`. To be added to this list, send a message to `m3-request@src.dec.com`. You may post articles to `comp.lang.modula3` by sending them to `m3@src.dec.com`.

Reporting bugs. We prefer that you send bug reports to `m3-request@src.dec.com`. After we have reviewed your report, we may post an article in `comp.lang.modula3`, describing the bug and a workaround or a fix.

Needless to say, this implementation probably has many bugs. We are quite happy to receive bug reports. We can't promise to fix them, but we will try. When reporting a bug, please send us a short program that demonstrates the problem.

Chapter 6

The libraries

SRC Modula-3 includes a large set of libraries, described in this chapter. It is intended that the interfaces within the library be complete and self documenting.

The library `foo` is in the files `LIB/libfoo.a` and `LIB/libfoo.ax`, and the interfaces that are implemented by this library are in the directory `PUB`; `LIB` and `PUB` depend on your local configuration, see chapter 7 for the values of these parameters (by default, they are `/usr/local/lib/m3` and `/usr/local/include/m3`).

Normally, the `m3` driver knows the location of the public interfaces and archives. You just need to pass the `-lfoo` option to `m3` to link with the library `foo`. Also, the driver automatically links with the `m3` library.

The key to making Modula-3 successful requires designing, building and sharing libraries. You are encouraged to send us useful modules or programs and we will include them in the next release as contributed software. You can also announce the availability of your work on `comp.lang.modula3`.

Your system may have additional libraries; see chapter 7 or ask your system administrator.

6.1 The m3 library

The `m3` library contains some basic interfaces and modules. This library is always included when linking Modula-3 programs, and its interfaces are accessible using the default search path.

Conversion of representation:

`Convert`	Basic binary/ASCII conversion of numbers
`Fmt`	Formatting to `Text.T`
`Scan`	Parsing from `Text.T`

Input/output is achieved using readers and writers:

`Rd`	Basic operations on readers
`UnsafeRd`	Faster version for non-concurrent access
`RdClass`	To implement new classes of readers
`Wr`	Basic operations on writers
`UnsafeWr`	Faster version for non-concurrent access
`WrClass`	To implement new classes of writers
`TextRd`	Readers that are connected to `Text.T`s
`TextWr`	Writers that are connected to `Text.T`s
`Stdio`	Readers and writers for standard files
`FileStream`	Readers and writers connected to named files
`UFileRdWr`	Readers and writers connected to file descriptors

Higher-level input/output:

`AutoFlushWr`	buffered writers that flush automatically
`Pkl`	reading and writing binary data structures

There is also a very primitive equivalent of `stdio`, which is needed by the low-levels of the runtime: `SmallIO`.

Fingerprints (64 bits CRC's are built using polynomial arithmetic:

`FPrint`	Compute the fingerprint of a `Text.T`
`PolyBasis`	support for `FPrint`
`Poly`	support for `FPrint`

There is a set of interfaces to provide standard access to and operations on basic types: `Char`, `Boolean`, `Cardinal`, `Integer`, `Real`, `LongReal`, `Address`, `Refany`, `Root`, and `Cast`.

The m3 library has a few basic data structures:

`List`	Lists of `REFANY`s
`IntTable`	Tables of `INTEGER`s
`RefTable`	Tables of `REFANY`s
`STable`	Sorted tables, implemented by 2-3-4 trees
`SIntTable`	`STable` applied to `INTEGER`
`STextTable`	`STable` applied to `Text.T`

There is a set of interfaces that give access to the ANSI-C libraries. This collection is under construction.

`M3toC`	support for Modula-3/C communication
`Ctypes`	C-like names for types
`Cstdarg`	obsolete
`Cstdlib`	`stdlib.h`
`Cstring`	`string.h`

There is a set of interfaces that give access to the runtime system. The `Rep` interfaces depend heavily on the runtime implementation; other interfaces are more likely to be present (at least, similar functionalities) in other systems.

`RTException`	exception mechanism
`RTMath`	basic math functions
`RTLink`	program initialization
`RTScheduler`	low-level access to the thread scheduler
`RTType`	type manipulation
`RTTypeRep`	more type manipulation
`RTProc`	procedure manipulation
`RTProcRep`	more procedure manipulation
`RTHeap`	heap allocation and garbage collection
`RTHeapRep`	additional control over the heap
`RTMisc`	miscellaneous support functions; runtime errors
`RTStack`	low-level thread stack allocation
`RTThread`	low-level thread switching

There is a set of interfaces giving access to the Unix system. These interfaces are machine-dependent, but we tried to use the same names in all versions to make programs easier to port. Thus, it should be no more difficult to port Modula-3 programs that use these interfaces than it is to port C programs.

In general, an interface regroups the definitions given by a system include file and the related functions. Eventually, all of sections 2 and 3 should be available. Currently, we have the following interfaces:

`Utypes`	Declarations of types name (`sys/types.h`)
`Uerror`	Declarations of error codes (`errno.h`)
`Uipc`	Inter-process communication (`sys/ipc.h`)
`Umsg`	Inter-process messages (`sys/msg.h`)
`Unetdb`	Network database manipulation (`netdb.h`)
`Uprocess`	Process ids
`Uresource`	Resources utilization (`sys/resource.h`)
`Usem`	Semaphores (`sys/sem.h`)
`Ushm`	Shared memory (`sys/shm.h`)
`Usignal`	Signals (`signal.h`)
`Utime`	Time manipulation (`sys/time.h`)
`Uugid`	User and group ids
`Uutmp`	Login names (`utmp.h`)
`Unix`	Other functions (not yet organized in separate interfaces)

Some math-oriented interfaces:

`Math`	sin, cos and friends
`Point`	2-D integral points
`Interval`	Open integral intervals
`Axis`	horizontal/vertical
`Rect`	2-D integral rectangles
`Transform`	2-D transformations
`Stat`	simple statistics

Finally, various interfaces, including the mandatory ones.

`Main`	Main program interface
`Text`	Character strings
`TextF`	Reveals to our friends what a `Text.T` is
`Thread`	Control of concurrency
`ThreadF`	Additional control for our friends
`Time`	Time manipulation
`Word`	Unsigned integer manipulation
`Random`	Random numbers
`RandomPerm`	
`RandomReal`	
`UID`	Generate unique identifiers
`ParseParams`	Parsing of UNIX-style command lines
`ParseShell`	Lower level support
`Formatter`	Formatting of text, for example for pretty-printers
`Filename`	File names manipulation

6.2 The data structures library

The library `m3data` provides generic data structures. The interfaces in that library are currently being designed and the implementations need more testing. Your comments are welcome.

6.3 The X11R4 library

The library `m3X11R4` contains binding interfaces for the X11R4 system. The interfaces are:

`X11`	Xlib-level functionalities
`Xt`	X Toolkit Intrinsics
`XtC`	
`XtE`	
`XtN`	
`XtR`	
`Xrm`	
`Xmu`	
`Xct`	
`Xaw`	Athena Widget set

6.4 The Trestle library

The library `m3ui` contains the Trestle toolkit. It's a powerful set of tools for building windowing applications. A full description of Trestle can be found in the "Trestle Reference Manual" [12].

6.5 The FormsVBT library

The library `m3vbtkit` adds another layer of window building tools. A full description of FormsVBT can be found in "The FormsVBT Reference Manual" [5].

6.6 The TclTk library

`Tcl` is an embeddable tool command language. `Tk` is an X11 toolkit based on the `Tcl` language. Both have been developed by John Ousterhout at UC Berkeley.

For an introduction to Tcl and Tk you may wish to read two papers: "Tcl: An Embeddable Command Language", in the Proceedings of the 1990 Winter USENIX Conference, and "An X11 Toolkit Based on the Tcl Language", in the Proceedings of the 1991 Winter USENIX Conference.

The library `TclTk` gives access to the `Tcl` and `Tk` libraries from Modula-3 programs, via the interfaces `TclC` and `TkC` respectively.

Chapter 7

Local Guide

This chapter describes how SRC Modula-3 is installed at SRC, how to use it and how to contribute to it.

7.1 Your Environment

Find a DECstation (Pmax or 3max). Modula-3 runs on Vaxen, but almost nobody bothers. Modula-3 doesn't run on Alphas, yet.

Before you work in Modula-3, login and make sure that your home directory contains the files listed below. It's important that you have these files in at least minimal working order. Beware! Getting everything perfect can be a *huge* time sink.

`.xsession` Your `.xsession` file is run first, it selects your window manager and gets your X server initialized. After the comments, around line 38, there's a line that looks like:

```
set WINMGR = "tvtwm"
```

The window managers that are allowed are `tvtwm`, `dxwm`, and `mwm`. Today most people are using `tvtwm`. Most window managers can be configured with a `rc` file. I have a file named `.tvtwmrc` that `tvtwm` reads during startup. I don't know exactly what the file does, nor what it could do. I suggest you copy a version of the file from a friend and read the `tvtwm` man page.

`.login` Your `.login` file should begin by reading the system-defined `.login` file using the shell command:

```
source /proj/local/lib/system.login
```

This command will work on all our machines (Fireflies, VAX mainframes, DECstations). The file it refers to figures out what kind of machine you are running on and sets your basic environment accordingly. In particular, the programs `m3`, `m3pp`, `m3make` and their man pages should be on your search paths.

Put your personal customizations after the `source` command given above.

`.cshrc` Similarly, your `.cshrc` file should begin with the line

```
source /proj/mips/lib/system.cshrc
```

and conclude with your personal customizations.

.X11Startup Your .X11Startup file is used to start a set of X applications each time you login. My .X11Startup contains:

```
xmodmap /udir/kalsow/.xmodmaprc
xload  -geometry 130x70-0+135 &
xclock -geometry 130x130-0+0 &
xterm  -iconic -geometry "80x55" &
xterm  -iconic -geometry "80x55" &
xmh    -iconic -geometry "685x750" &
xrn    -iconic -geometry "685x750" &
```

`xmodmap` is a program that can redefine the mapping of your keyboard. My `.xmodmaprc` file fixes the brain damaged DECstation keyboard so that the escape key is in the right place and the shift lock is disabled. If you do nothing, the mystery key labelled "F11" is your escape key. Here's my `.xmodmaprc` file:

```
!
! Caps_Lock -> Control
! F1        -> Caps_Lock
!
remove Lock      = Caps_Lock
keysym F1        = Caps_Lock
keysym Caps_Lock = Control_R
add Lock         = Caps_Lock
add Control      = Control_R

!
! key cap      character   (first unshifted, second shifted)
!
!   , ,          ,   <
!   . .          ,   >
!   < >          ‘   ~
!   ‘ ~          ESC ESC
!
keysym less      = quoteleft asciitilde
keysym comma     = comma     less
keysym period    = period    greater
keysym quoteleft = Escape    Escape
```

I also start a program that displays my system's load average – `xload`, a clock, a few X terminals, my mail reader – `xmh`, and my news reader – `xrn`.

.Xdefaults Your `.Xdefaults` file defines some of the ten bazillion options that make X applications so much fun. My advice is to steal a copy from someone who's screen looks OK. When you're really bored, diddle with your `.Xdefaults`. (You need to login again or run `xdb` to reload your `.Xdefaults` file.)

7.2 Editing

Several people at SRC have spent some time doing various things to try to make Modula-3 programming in `gnuemacs` and `epoch` more pleasant and productive. Unfortunately, it is more difficult than it should be to find out about these efforts; you have to ask the right people, look at comments in various source files, read the right bulletin boards, etc. The purpose of this section is to describe these packages as something like a coherent whole.

Our editing czar says, “use `epoch`”. I don’t see any reason to disobey. `Epoch` is a version of `gnuemacs` that’s been feature-ified to fit better with X. Otherwise, the two editors are very similar. They can even share the same elisp code. As always, you should probably read its man page.

You’ll need a `.emacs` file to hold your personal configuration. The best way to get started is to copy this file from someone you trust.

You may also have a `.epoch` file. It’s intended to hold `epoch`-specific elisp. Note, this file is not read automatically. You must arrange to read it. Here’s the recommended recipe for invoking your `.epoch` file:

```
(if (boundp 'epoch::version)
    (progn
      (load "dot.emacs" nil t)
      (setq auto-raise-screen nil)))
```

The following `gnuemacs` elisp packages are in `/proj/m3/pkg/gnuemacs/src` and described below:

- `modula3.el`

  This package defines `modula-3-mode`, an emacs “major mode” for editting Modula-3 source code. This package has grown by accretion over a number of years, by a number of hands. It provides mechanisms for formatting code and for inserting keywords or whole syntactic constructs.

- `m3tags`

  Eric Muller has adapted the emacs `tags` facility to work for Modula-3. This allows one to quickly go to the definition of a sytactic unit when the cursor is pointing at a use of that unit.

- `lightbrite`

  If you want a little dash of color in your programming life, the lightbrite package can provide it, by highlighting certain keywords and comments in different colors and/or fonts.

modula-3-mode

`Gnuemacs` Modula-3 mode has grown by accretion, by a number of hands over a number of years. It often provides a couple of ways of accomplishing the same goal. People who modified the code to add new features tried not to change the behavior observed by current users. In short, the code is something of a mess, and there are a lot of variables you can set to get different behaviors. It might be a good idea to go for someone to make editorial decisions so that there is only one way of doing each thing, and maybe call the result a new mode.

Here is a list of the key things Modula-3 mode provides:

- Avoidance of typing:

  If you don’t like typing a lot of uppercase keywords, there are two methods you can use to automatically insert keywords or entire syntactic constructs. One is termed “aggressive”, the other “polite”. In aggressive mode, various keystrokes starting with `^C` are bound to functions that insert entire syntactic constructs into your buffer; for instance, `^C-b` gives you a BEGIN/END pair, both at the current indentation. In polite mode, there is ubiquitous completion of keywords, bound to the `<TAB>` key. For example `b<TAB>` expands the `b` to `BEGIN`, provided the `b` appears in a context where `BEGIN` may be a valid keyword. There are some fairly extensive rules governing the contexts in which a given keyword is a valid completion; the net result is that it is seldom necessary to type more than one letter to get the correct completion. If you get specify a non-unique prefix of a set of keywords, it chooses the first in an ordering intended to capture frequency of use; it also presents the other choices, and typing `<TAB>` repeatedly cycles through these choices.

- Indenting/pretty-printing

  There are also two methods for pretty-printing code. The first is via invocation of `m3pp`. `M-x m3pp-region` will pretty-print the code between mark and point. `M-x m3pp-unit` pretty prints the "unit" containing the cursor. A unit starts with a blank line followed by `CONST`, `TYPE`, `VAR`, `PROCEDURE`, `EXCEPTION`, `IMPORT`, `FROM`, `MODULE`, or `BEGIN`, and it extends to the start of the next unit. If there is no such unit around the cursor, the entire file is pretty-printed. Unfortunately, the `m3pp-region` and `m3pp-unit` commands are not bound to keys. You can add the conventional binding for these commands by adding:

  ```
  ?? help ??
  ```

  to your `.emacs` file.

  The other method of pretty-printing is a gnuemacs "electric" mode, where a key (`<TAB>` again; it serves double duty) immediately indents the current line with respect to the previous line. Another pair of features of the electric mode are "`END`-matching" and "`END`-completion"; if enabled, `END`-matching blinks the cursor briefly at the construct matching an `END`, and `END`-completion fills in the name of the procedure or module an `END` completes, or a comment with the name of the construct completed.

  The two methods are not mutually exclusive; perhaps you like the way `m3pp` lines up columns in declarations, but you also like to keep things indented while you type. You can use the electric mode to get things close, then invoke `m3pp` when you're done. Personally, I just use the electric mode.

- Finding files.

  There are (you guessed it) two methods for quickly finding an interface. Both expect the point to be on an interface name, and find the file for that interface by probing a search path. `m3-path-find` file, bound to `^C-^O-v`, assumes the presence of an `m3path` file in the current directory, and finds the file in another window. `m3::show-interface` (not bound to any key by default) uses the variable `*m3::defpath*` as the search path, and, if epoch is being used, displays the found file in a new screen (i.e., window.)

  Apparently, Steve Glassman has yet another variation on this theme, which knows enough about our symbolic link conventions to allow finding the implementation of an interface, as well. It will be a good thing to merge the good features of these three into one.

To use modula-3-mode, put the following lines in your `.emacs` file:

```
(setq auto-mode-alist
    (append (list '("\\.m3$\\|\\.i3$\\|\\.ig$\\|\\.mg$" .
                    modula-3-mode))
    auto-mode-alist))
```

The `auto-mode-alist` says what mode you should enter when a file extension matches the given regular expression.

```
(autoload 'modula-3-mode "modula3"
"A special mode for M3." t)
```

The autoload command tells emacs what elisp file to load to get the definition for a given function.

```
(setq completion-ignored-extensions
    (append '(".mo" ".mx" ".mc" ".io" ".ix")
            completion-ignored-extensions))
```

When you find files, you would rather not be offered files with these extensions as possible completions.

```
(defun m3-mode-hook-function ()
  (setq m3-abbrev-enabled 'polite)
  (setq m3-electric-end 'all)
  (setq m3-blink-end-matchers t))

(setq m3-mode-hook 'm3-mode-hook-function)
```

The `m3-mode-hook` variable specifies a function to run when you enter `modula-3-mode`.

The `m3-mode-hook-function` given above customizes the behavior of `modula3-mode` to:

- use `polite` abbrev mode. `'aggressive` is the default and `nil` is legal.
- do all possible `END`-completion. `'proc-mod` will match procedure and module names, `nil` is the default and matches nothing.
- blink `END`-matchers. `nil` is the default value.

m3-tags

"Tags" is supposed to allow you to quickly find the definition of a given construct when the point is at a use. Eric Muller is the local master of tags. The program `m3tags` builds tags for the public interfaces. To use the tag database from `epoch` you need to add the line

```
(visit-tags-table "/proj/m3/lib.mips/FTAGS")
```

to your `.emacs` file.

INSERT how to use Steve Glassman's "mpindex".

lightbright

This is the stuff Dave Detlefs demonstrated at the 6/17/92 Center Meeting, which put Modula-3 keywords in different colors and fonts, and comments in proportionally-spaced italics.

We don't recommend using this package at yet: it requires a newer version of Epoch than the default, and it applies uqiquitously across modes. However, if you must :-), run `/udir/detlefs/bin/epoch`, and add the following lines in your .emacs:

```
;;;------- lightbright.el ---------------------------

(defun make-style-with-font (font)
  "Make a style with font FONT."
  (let ((s (epoch::make-style)))
    (set-style-font s font)
```

```
    s))

(defun make-style-with-font-and-color (font color)
  "Make a style with font FONT."
  (let ((s (epoch::make-style)))
    (set-style-font s font)
    (set-style-foreground s color)
    s))

(if (boundp 'epoch::version)
    (progn
      (load "lightbrite")
      (setq brite::touchup-threshold 200000)
      (setq brite::comment-threshold 200000)
      (setq-default brite::change-interval 6)

      (setq red-style (make-style-with-font-and-color
                       "*helvetica-medium-r-normal--10*"
                       "red"))
      (setq green-style (make-style-with-font-and-color
                         "*courier-medium-r-normal--12*"
                         "blue"))
      (setq yellow-style (make-style-with-font
                          "*helvetica-bold-r-normal--10*"))
      (setq magenta-style (make-style-with-font
                           "*helvetica-medium-r-normal--10*"))
      (setq comment-style (make-style-with-font
                           "*times-medium-i-normal--12*"))
      (setq yellow-underline-style (make-style-with-font
                                    "*helvetica-medium-o-normal--12*"))
      (setq red-underline-style (make-style-with-font
                                 "*helvetica-bold-o-normal--12*"))
      ))

;;;------- end stuff for lightbright.el ---------------------------
```

to do (i.e. Dave Detlefs' wish list)

- General cleanup. `Modula3.el` needs a general pass of cleanup, elimination of redundant code, better allocation of the scarse resource of keybindings, etc.
- Identifier completion. Dave has an experimental `m3-complete-identifier` command, that constructs the complete list of identifiers declared in the current file, and then uses the current word as a prefix to pick out a set of possibilites. It is buggy, and too slow to use on even medium-sized files. He may try to make it robust, fast enough, and more ambitious: completing record field names or second parts of qualified names, etc.
- Epoch display of the call stack in debugging. It would be kind of neat if, while debugging, the buffers on the source files for the call stack appeared as some sort of stack of windows.

7.3 Compiling

You should use `m3make`. The best advice for beginners is to copy an existing package that's similar to the one you want to create. You'll probably find the following files:

```
./README          - a top-level description created by m3create
./src/m3makefile  - the input to m3make
./src/*.[im][3g]  - the Modula-3 sources of the program
./mips/*.[im][ox] - the compiled objects for a DECstation
./vax/*           - the compiled objects for a VAX
```

There's a man page for `m3make`, at some point you should read it.

You can use `epoch`'s `compile` command to run `m3make` and then use the `next-error` facitility to quickly move to the source lines containing errors.

The standard `epoch` `M-x compile` command will work with Modula-3 and `m3make`. Invoke this command in a buffer whose current directory is the one containing your derived files (i.e. if your cursor is in a source file then first visit `../mips`). When you invoke this command, it presents the current `compile command` in the minibuffer for your approval. The first time you run this, the command will be `make -k`. If you modify this to `m3make`, that will become the new current command on subsequent makes. The results of the `m3make` will be displayed in a buffer called `*compilation*`.

When a compile is finished, `next-error` (`^X-‘`) will parse the `*compilation*` buffer and find the first error in the file, moving the current error to the top of the `*compilation*` buffer, and moving the point to the line containing the error in the appropriate source buffer. When you invoke `next-error` again, it will go to the start of the next line containing an error. The big advantage of using `next-error` is that uses emacs "marker" facility to keep things straight if you edit the file to fix errors, changing the line numbers. Barring drastic edits, `next-error` will still get you to the right line.

Mick Jordan has also defined some commands that let you use `m3check` in a shell window, and get about the same `next-error` behavior. *INSERT the details...*

7.4 Debugging

Today you get your choice of debugger – either `gdb` or `dbx`. At the moment most SRCers prefer `gdb`, although it can't read DECstation core files.

with `dbx`

Like everything else, `dbx` will read a start-up file. It reads the file named `.dbxinit` in your home directory. In that file you place any commands that you'd like executed each time you start the debugger. Here's a suggested `.dbxinit` file:

```
ignore 26
set $casesense = 1
set $printwide = 1
stop in RTException__NoHandler
alias typeof(r) "print (_types[(*(int *)(r-4))//2]).name"
alias wide "set $printwide = (1-$printwide)"
alias threads "call Thread__DumpEverybody()"
use src ../src /udir/XYZ/pkg/A/src /udir/XYZ/pkg/B/src
```

The first line tells dbx to ignore signal 26 – the thread switching timer. The second line tells dbx that all input is case-sensitive. The third line tells dbx to print folded lines of output for large structures. The fourth line sets a break point in the runtime routine that's called for unhandled exceptions. The fifth line defines a new command that examines runtime data structures and prints the type of a reference. The fifth line defines a new command that toggles the folded output behavior. The sixth line defines a new command that produces a listing of all threads. The last line gives dbx a list of directories to search when it's looking for source files.

with gdb

Like dbx and everything else, gdb will read a start-up file. It reads the file named .gdbinit in your home directory. In that file you place any commands that you'd like executed each time you start the debugger. Here's a suggested .gdbinit file:

```
dir
dir /udir/steveg/b.e/mg/src
dir /udir/steveg/b.e/zeus/src
dir /udir/steveg/b.e/lego/src

define ss
nexti
x/i $pc
end

define breaks
info breakpoints
end

break RTException__NoHandler

define threads
call Thread__DumpEverybody()
end
```

The first set of lines gives gdb a search path for locating source files. The next set defines a single step command. The third set defines a simple command to list the active breakpoints. The fourth sets a break point in the runtime routine that catches unhandled exceptions. And the last set of lines defines a threads command to list all threads.

from epoch

There are some advantages to debugging Modula-3 programs under epoch. You get

- Automatic display of the source file with the current line indicated when you stop at a breakpoint or move around in the stack while the program is stopped.
- You can set a breakpoint at the current line in a source file using the ^X-<space> command.
- Special commands take REF or OBJECT variables (or their pointer values) and print their types and the values of the referents. (This feature depends on intimate knowledge of the SRC Modula-3 implementation.) The command ESC-p takes the word containing the current point as the variable or value to print. ESC-r does the same thing with a record value.

These functions are built from `gnuemacs`' `gdb-mode` and `dbx-mode` and are available in the `m3-debug.el` elisp package. To use them, put these lines in your `.emacs` file:

```
(autoload 'run-m3-gdb "m3-debug" "" t)
(autoload 'run-m3-dbx "m3-debug" "" t)
```

You can then use `M-x run-m3-gdb` or `M-x run-m3-dbx` to start gdb or dbx, respectively, in inferior shells in the appropriate modes.

7.5 Packages

We use the same package tools that everyone else at SRC is using. There are special variants of the commands that make it a little simpler for Modula-3 users.

To get the Modula-3 package tools to work on your DECstation named `foobaz`, you must have a `.rhosts` file in your home directory on `bigtop`. It should contain the line:

```
foobaz.pa.dec.com
```

To manipulate a Modula-3 package named `foo`, the last component of your working directory must be named `foo`.

Briefly, here's the available commands and what they do when issued in a directory named `XYZ`.

m3create – creates a new package named `XYZ`.

m3delete – deletes the package named `XYZ`.

m3get – acquires the lock and updates the current directory with the contents of the package `XYZ`.

m3setlock – acquires the lock on package `XYZ`.

m3unlock – unlocks the package `XYZ`.

m3ship – ships a copy of the current directory as the new contents of package `XYZ`.

m3compare – compares the current directory with the existing version of package `XYZ`.

For more details, see the man pages

7.6 Public Directories

All the basic Modula-3 software lives in packages. These packages live in subdirectories of `/proj/m3/pkg`. The public files are exported to public directories:

symbol	public directory	decription
`PUB`	`/proj/m3/pub.`*cpu_type*	interfaces
`LIB`	`/proj/m3/lib.`*cpu_type*	libraries
`BIN`	`/proj/{mips,ultrix}/bin`	programs
`CAT`*n*	`/proj/man/`*cpu_type*`/man/cat`*n*	plain-text man pages

where *cpu_type* is either `vax` or `mips`, and *n* can vary from `1` to `8`.

The Modula-3 compiler (`m3`) knows about the public directories. By default, it will search the current directory and `PUB` for interfaces. It will also try to locate libraries specified in the `-l` syntax in `LIB`. It will systematically link your programs with `-lm3` and `-lm`.

7.7 Package Organization

We have four kinds of packages:

- source packages: They contain Modula-3 sources and produce no derived objects. For an example, see the `text` package.
- library packages: They usually contain a small number of source files, export a few interfaces to `PUB` and export a single library containing all the objects to `LIB`. For example, the `tcl` package contains the source files for the binding to `Tcl`. It exports `TclC.i3` to `PUB` and `libm3tcl.a` to `LIB`.
- umbrella packages: It is inconvenient to work with a large number of small libraries. An umbrella library collects a number of smaller libraries; it is essentially a list of source packages and it exports a library containing all the objects of the smaller packages. An example is `libm3.a`. This library collects the contents of several source packages into a single large library.
- program packages: They contain a single program, exported to `BIN`, with its man page exported to `CAT1`. For an example, see the `solitaire` or `calculator` package.

Chapter 8

Internals

This section contains a brief introduction to the internal structure of the compiler and runtime system. This introduction is neither comprehensive nor tutorial; it is merely intended as a stepping stone for the courageous.

8.1 A tour of the compiler

The compiler has undergone much evolution. It started as a project to build a simple and easy to maintain compiler. Somewhere along the way we decided to compile Modula-3. Much later we decided to generate C. In hindsight, Modula-3 was a good choice, C was at best mediocre.

The initial observation was that most compilers' data structures were visible and complex. This situation makes it necessary to understand a compiler in its entirety before attempting non-trivial enhancements or bug fixes. By keeping most of the compiler's primary data structures hidden behind opaque interfaces, we hoped to avoid this pitfall. So far, bugs have been easy to find. During early development, it was relatively easy to track the weekly language changes.

The compiler is decomposed by language feature rather than the more traditional compiler passes. We attempted to confine each language feature to a single module. For example, the parsing, name binding, type checking and code production for each statement is in its own module. This separation means that only the `CaseStmt` module needs to know what data structures exist to implement `CASE` statements. Other parts of the compiler need only know that the `CASE` statement is a statement. This fact is captured by the object subtype hierarchy. A `CaseStmt.T` is a subtype of a `Stmt.T`.

The main object types within the compiler are: values, statements, expressions, and types. "Values" is a misnomer; "bindings" would be better. This object class include anything that can be named: modules, procedures, variables, exceptions, constants, types, enumeration elements, record fields, methods, and procedure formals. Statements include all of the Modula-3 statements. Expressions include all the Modula-3 expression forms that have a special syntax. And finally, types include the Modula-3 types.

The compiler retains the traditional separation of input streams, scanner, symbol table, and output stream.

The compilation process retains the usual phases. Symbols are scanned as needed by the parser. A recursive descent parser reads the entire source and builds the internal syntax tree. All remaining passes simply add decorations to this tree. The next phase binds all identifiers to values in scopes. Modula-3 allows arbitrary forward references so it is necessary to accumulate all names within a scope before binding any identifiers to values. The next phase divides the types into structurally equivalent classes. This phase actually occurs in two steps. First, the types are divided into classes such that each class will have a unique C representation.

Then, those classes are refined into what Modula-3 defines as structurally equivalent types. After the types have been partitioned, the entire tree is checked for type errors. Finally, the C code is emitted. C's requirement that declarations precede uses means that the code is generated in several passes. First, the types are generated during type checking. Then, the procedure headers are produced. And finally, the procedure bodies are generated.

The compiler implementation is in the `compiler` directory. Within that directory the following directories exist:

builtinOps	`ABS`, `ADR`, `BITSIZE`, ...
builtinTypes	`INTEGER`, `CHAR`, `REFANY`, ...
builtinWord	`Word.And`, `Word.Or`, ...
exprs	`+`, `-`, `[]`, `^`, `AND`, `OR`, ...
misc	main program, scanner, symbol tables, ...
stmts	`:=`, `IF`, `TRY`, `WHILE`, ...
types	`ARRAY`, `BITS FOR`, `RECORD`, ...
values	`MODULE`, `PROCEDURE`, `VAR`, ...

8.2 A tour of the runtime

The runtime itself implements the garbage collector, Modula-3 startup code and a few miscellaneous functions. The runtime exists in the `libm3/runtime` directory.

The interface between the compiler and runtime system is embodied (and very sparsely documented) in `M3Runtime.h`, `M3Machine.h` (an architecture-dependent file) and `M3RuntimeDecls.h`. Every C file generated by the compiler includes these files.

The allocator and garbage collector are based on Joel Bartlett's "mostly copying collector". The best description of his collector is in [1]. Since that paper, we've made a few modifications to support a growing heap and to use extra information that the Modula-3 compiler generates.

Exceptions are implemented with `setjmp` and `longjmp`. The jump buffers and scope descriptors are chained together to form a stack. The head of the chain is kept in `ThreadSupport.handlers`. There is a distinct chain for each thread. When an exception is raised, the chain is searched. If a handler for the exception is found, the exception is allowed to unwind the stack, otherwise a runtime error is signaled. To unwind the stack, a `longjmp` is done to the first handler on the stack. It does whatever cleanup is necessary and passes control on up the stack to the next handler until the exception is actually handled.

Reference types are represented at runtime by a "typecell". Due to separate compilation, opaque types and revelations, it is not possible to fully initialize typecells at compile time. Typecell initialization is finished at link time. A typecell contains a type's typecode, a pointer to its parent typecell, the size of the types referent and method list if any, the type's brand, the number of open array dimensions, the type's fingerprint, and procedures to initialize the typecell, initialize new instances of the type, print instances of the type and trace the type for garbage collection.

Link time type elaboration occurs in several steps. First, all types are registered. That is, a global array that points to all typecells is built. Next, the runtime verifies that all opaque types have been given concrete representations. Then, the initialization of typecells is finished. Then, all types with the same brand and fingerprint are identified with the same typecode. Finally, a check is made to ensure that distinct types have distinct brands.

At the beginning of the execution of the program, all global variables are initialized, and the main bodies of the modules are invoked. The skeletal code that ensures that every module is initialized is generated by the linker part of the driver.

Other parts of the runtime, such as threads, are actually implemented in the base library.

Thread switching is implemented with `setjmp` and `longjmp`. A timer interrupt (signal) is used to cause preemptive thread switching. The global variable `ThreadSupport.self` points to the currently running thread. The integer `ThreadSupport.inCritical` is used by the runtime to prevent thread switching during garbage collection and other "atomic" runtime operations.

8.3 Porting to another machine

Anyone who is interested in porting this compiler is encouraged! We would like to know how it goes. The primary concerns when doing a port will be the size and alignment constraints of the target machine and the runtime. We tried to avoid suspicious C constructs, but we doubt that we were completely successful.

The directions in this section are somewhat sparse. We tried to make the installation of SRC Modula-3 smooth, but it is another story to make the development of ports smooth. Please bear with us and tell us what we can do to improve this section.

If you want to a port to an unsupported system you should:

- get the `compiler` and `driver` source archives (in addition to `m3make` that came with the boot archive and `libm3` which you had to install anyway).
- decide on the name of the new architecture; in the rest, we assume that it is *new*
- describe the target machine for the compiler
- implement the machine-specific part of the base libraries for the new machine
- build a cross-compiler on a supported machine
- cross-compile (to C) the driver and the compiler
- finish the compilation of the driver and the compiler on the target machine

In the following, all the paths are relative to the directory in which you unpacked the archives (also known as the top-level directory).

Describing the target machine The compiler has a small number of parameters that are used to describe the target machine. These parameters are expressed in the interface `compiler/src/`*new*`/Target.i3`. Create the directory `compiler/src/`*new* and build the file `Target.i3`, using the descriptions for the other machines as models. In `compiler/src/m3makefile`, add the lines:

```
#if defined (TARGET_new)
source_dir (../src/new)
#endif
```

Porting the runtime and base libraries Some of the Modula-3 code (as well as very little pieces of C) are machine-dependent. Of course, it may be that some code we thought to be machine-independent will turn to have to be changed for your *new* architecture, so we cannot guarantee that the list below is exhaustive. In general, look at what is done for the other machines, and find the most similar as a starting point.

In `libm3/Csupport/src`, add a directory *new* and put in it the files:

- `m3makefile` to describe the contents of the directory

- `M3Machine.h` which is included in every C file generated by the SRC Modula-3 compiler
- `dtoa.c` to configure `../generic/dtoa.h`; look at that file for the things to configure.
- `float.h` if your system does not have one. You can build it using a program called `enquire`, which can be found on the net.

In `libm3/C/src`, add a directory *new* and put in it the files:

- `m3makefile` to describe the contents of the directory
- `Csetjmp.i3` to describe the interface to `setjmp`, `longjmp`, `_setjmp` and `_longjmp`. Be careful to get the size of the `jmp_buf` right.
- `Cstdio.i3`, essentially a Modula-3 translation of `stdio.h`

In `libm3/thread/src`, add a directory *new* and put in it the files:

- `m3makefile` to describe the contents of the directory
- `WildJmp.i3`: this interface provides functions that are similar to `_setjmp` and `_longjmp`, but that do not impose any restriction on what is possible. The `DS3100` version is the simplest, because on that architecture, `_setjmp` and `_longjmp` are just fine. The `VAX` version is the most complex, we had to rewrite our own versions because `longjmp` requires that the stack be popped (remember the `longjmp botch` message ?).

In `libm3/runtime/src`, you can either reuse one of the `StackInc-`n component, or you may have to create a new one (i.e., you need a different value of n). The dependency described there is for the benefit of the garbage collector. At the beginning of a collection, the collector must find all the roots, that is, all the heap objects that are referenced from outside the heap itself. The stacks contain such pointers, and the collector scans them to find roots. However, the collector does not know the full structure of the stacks (frames, argument lists and so on); rather, it just looks at the values that are there and make conservative decisions by interpreting these values as possible pointers. For each stack, the collector initializes a pointer P to the bottom of the stack; then it repeatedly tries to interpret the bits pointed by P as a pointer in the heap, marks the root if the interpretation is successfull and advances P. The question is by how much P should be advanced; if all entries in the stack are aligned at n-bytes boundaries, it is sufficient to increment P by n bytes; a smaller value would be an overkill. We have found that some machines require n to be 2, and that 4 is enough for others.

In `libm3/float/src`, add a directory *new* and copy the files that are in `MODEL` in that directory. The routines in these modules provide access the floating point control (to set exceptions and so on). The version in `MODEL` is a template, and most of the routines will fail (because of an <*ASSERT FALSE*>) if executed. The `DS3100` and `SPARC` directories are examples of implementations for IEEE machines, the `VAX` directory is an example for non-IEEE machines. It is not essential that you implement the proper procedures right now: the versions in `MODEL` are good enough for the compiler, the driver and simple test programs. But you will have to take care of that at some point.

In `libm3/random/src`, you can either reuse one of the directories `VAX`, `IEEE-le` or `IEEE-be`, or create your own on those models. The goal is to describe enough of the floating point representation for the random number generator. There is probably some overlap with the `libm3/float` stuff, we will take care of that at some point.

In `libm3/unix/src`, you will find a bunch of interfaces to the procedures of sections 2 and 3 of U**X. Not everything is there, but we sometime dream to have a complete set; in other words, it's quite a bit of work to make sure that you have the proper descriptions, and of course, there is nothing from which these interfaces could be mechanically derived. Fortunately, the driver and the compiler rely on very few of these procedures, and any version is probably good enough for your machine. We suggest that you do the work only when you find some problems (at least, wait until you get a basic port running).

The last thing is to reflect all these changes in `libm3/src/m3makefile`. Add a bunch of lines:

```
#if defined (TARGET_new)
...
#endif
```

similar to those that are already there. You need to make a similar modification to `compiler/src/m3makefile` and `driver/src/m3makefile` (sorry for the duplication, but it is difficult to avoid).

Creating a cross-compiler At the top level, type to the shell:

```
$ m3make cross NEW=new
```

After a while, you should get an executable `compiler/cross-`*new*`/m3compiler`.

Cross-compiling the driver and the compiler At the top level, type to the shell:

```
$ m3make bootstrap_driver NEW=new
$ m3make bootstrap_compiler NEW=new
```

At this point, you should in the same state as if we had built a boot archive for *new* and you had grabbed it. If you cannot mount the file system that contains all the files on the new machine, create a boot archive:

```
$ m3make pack NEW=new
```

This creates a file `boot_files/boot.`*new-version.tar.Z*, which you need to unpack on the *new* machine.

You can then proceed as for the first installation of SRC Modula-3(see the top level `README`).

Good Luck!

Bibliography

[1] Joel F. Bartlett.
Compacting garbage collection with ambiguous roots.
WRL Research Report 88/2, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, February 1988.

[2] A. D. Birrell, J. V. Guttag, J. J. Horning, and R. Levin.
Synchronization primitives for a multiprocessor: A formal specification.
SRC report 20, System Research Center, Digital Equipment Corporation, Palo Alto, August 1987.

[3] Andrew D. Birrell.
An introduction to programming with threads.
SRC report 35, System Research Center, Digital Equipment Corporation, Palo Alto, January 1989.

[4] Gilad Bracha and William Cook.
Mixin-based inheritance.
In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications; European Conference on Object-Oriented Programming*, pages 303–311, October 1990.

[5] Marc H. Brown and James R. Meehan.
The formsvbt reference manual.
SRC report ??, System Research Center, Digital Equipment Corporation, Palo Alto, July 1992.

[6] Mark R. Brown and Greg Nelson.
IO streams: Abstract types, real programs.
SRC report 53, System Research Center, Digital Equipment Corporation, Palo Alto, November 1989.

[7] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson.
Modula-3 report (revised).
SRC report 52, System Research Center, Digital Equipment Corporation, Palo Alto, November 1989.

[8] Luca Cardelli, James Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson.
The modula-3 type system.
In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 202–212, January 1989.

[9] Sam Harbison.
Modula-3.
Byte, 15(12):385–392, November 1990.

[10] Sam Harbison.
Safe programming with modula-3.
Dr. Dobb's Journal, 17(10):88–96, October 1992.

[11] Samuel P. Harbison.
Modula-3.
Prentice Hall, 1992.

[12] Mark S. Manasse and Greg Nelson.
Trestle reference manual.
SRC report 68, System Research Center, Digital Equipment Corporation, Palo Alto, December 1991.

[13] Mark S. Manasse and Greg Nelson.
Trestle tutorial.
SRC report 69, System Research Center, Digital Equipment Corporation, Palo Alto, May 1992.

[14] Greg Nelson, editor.
System Programming with Modula-3.
Prentice Hall, 1991.