The Mercury User's Guide

Version 22.01.9

Fergus Henderson
Thomas Conway
Zoltan Somogyi
Peter Ross
Tyson Dowd
Mark Brown
Ian MacLarty
Paul Bone

Copyright © 1995–2012 The University of Melbourne.
Copyright © 2013–2026 The Mercury team.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

Table of Contents

1 Introduction 1

2 File naming conventions 1

3 Using the Mercury compiler 2

4 Running programs 5

5 Using Mmake 5

6 Libraries 11
- 6.1 Writing libraries 11
- 6.2 Building with mmc –make 12
  - 6.2.1 Building and installing libraries with mmc –make 12
  - 6.2.2 Using installed libraries with mmc –make 12
  - 6.2.3 Using non-installed libraries with mmc –make 13
- 6.3 Building with Mmake 13
  - 6.3.1 Building libraries with Mmake 13
  - 6.3.2 Installing libraries with Mmake 14
  - 6.3.3 Using libraries with Mmake 15
- 6.4 Libraries and the Java grade 16
- 6.5 Libraries and the C# grade 16

7 Debugging 16
- 7.1 Quick overview 16
- 7.2 GNU Emacs interface 17
- 7.3 Tracing of Mercury programs 19
- 7.4 Preparing a program for debugging 21
- 7.5 Tracing optimized code 24
- 7.6 Mercury debugger invocation 24
- 7.7 Mercury debugger concepts 25
- 7.8 User defined events 28
- 7.9 I/O tabling 30
- 7.10 Debugger commands 31
  - 7.10.1 Interactive query commands 31
  - 7.10.2 Forward movement commands 32
  - 7.10.3 Backward movement commands 35
  - 7.10.4 Browsing commands 37
  - 7.10.5 Breakpoint commands 44
  - 7.10.6 I/O tabling commands 50
  - 7.10.7 Parameter commands 51

7.10.8 Help commands ... 56
7.10.9 Declarative debugging mdb commands ... 57
7.10.10 Miscellaneous commands ... 58
7.10.11 Experimental commands ... 59
7.10.12 Developer commands ... 62
7.11 Declarative debugging ... 66
7.11.1 Overview ... 66
7.11.2 Concepts ... 66
7.11.3 Oracle questions ... 67
7.11.4 Commands ... 68
7.11.5 Diagnoses ... 71
7.11.6 Search modes ... 72
7.11.6.1 Top-down mode ... 72
7.11.6.2 Divide and query mode ... 72
7.11.6.3 Suspicion divide and query mode ... 73
7.11.6.4 Binary search mode ... 73
7.11.7 Improving the search ... 73
7.11.7.1 Tracking suspicious subterms ... 73
7.11.7.2 Trusting predicates, functions and modules ... 74
7.11.7.3 When different search modes are used ... 75
7.12 Trace counts ... 75
7.12.1 Generating trace counts ... 75
7.12.2 Combining trace counts ... 76
7.12.3 Slicing ... 76
7.12.4 Dicing ... 78
7.12.5 Coverage testing ... 80

8 Profiling ... 80

8.1 Profiling introduction ... 80
8.2 Building profiled applications ... 81
8.3 Creating profiles ... 82
8.4 Using mprof for time profiling ... 82
8.5 Using mprof for profiling memory allocation ... 85
8.6 Using mprof for profiling memory retention ... 86
8.7 Using mdprof ... 87
8.8 Using threadscope ... 88
8.9 Profiling and shared libraries ... 88

9 Invocation ... 89

9.1 Invocation overview ... 89
9.2 Warning options ... 89
9.3 Verbosity options ... 95
9.4 Output options ... 98
9.5 Auxiliary output options ... 101
9.6 Language semantics options ... 106
9.7 Termination analysis options ... 107
9.8 Compilation model options ... 108
9.8.1 Grades and grade components ... 108

9.8.2 Target options ... 112
9.8.3 LLDS back-end compilation model options ... 113
9.8.4 MLDS back-end compilation model option ... 113
9.8.5 Optional features compilation model options ... 113
9.8.6 Developer compilation model options ... 115
9.9 Code generation options ... 116
9.9.1 Code generation target options ... 117
9.10 Optimization options ... 117
9.10.1 Overall optimization options ... 117
9.10.2 High-level (HLDS -> HLDS) optimization options ... 119
9.10.3 MLDS back-end (MLDS -> MLDS) optimization options ... 122
9.10.4 Medium-level (HLDS -> LLDS) optimization options ... 123
9.10.5 Low-level (LLDS -> LLDS) optimization options ... 124
9.10.6 Output-level (LLDS -> C) optimization options ... 125
9.11 Target code compilation options ... 126
9.12 Link options ... 127
9.13 Build system options ... 131
9.14 Miscellaneous options ... 135

10 Environment variables ... 136

11 Using a different C compiler ... 142

12 Foreign language interface ... 142

13 Stand-alone interfaces ... 143

Index ... 145

1 Introduction

This document describes the compilation environment of Mercury. It describes

- how to use '`mmc`', the Mercury compiler;
- how to use '`mmc --make`', a build tool integrated into '`mmc`';
- how to use an older build tool, '`mmake`', built on top of GNU make;
- how to use '`mdb`', the Mercury debugger;
- how to use '`mprof`', a '`gprof`'-style profiler for Mercury; and
- how to use '`mdprof`', a more detailed profiler for Mercury that uses a web interface.

Mercury programs can be compiled to executables via any one of three target languages: C, Java and C#. The most frequently used target language is C, for two main reasons. The first is that several important parts of the Mercury system, such as '`mdb`', '`mprof`' and '`mdprof`', are implemented only when targeting C. The second is that targeting C yields the fastest executables.

2 File naming conventions

Mercury source files must be named '`*.m`'. Each Mercury source file should contain a single Mercury module whose module name should be (although it does not have to be) the same as the filename without the '`.m`' extension.

The Mercury implementation uses a variety of intermediate files, which are described below. But all you really need to know is how to name source files. For historical reasons, the default behaviour is for intermediate files to be created in the current directory, but if you use the '`--use-subdirs`' option to '`mmc`' or '`mmake`', all these intermediate files will be created in a '`Mercury`' subdirectory, where you can happily ignore them. Thus you may wish to skip the rest of this chapter.

In cases where the source file name and module name don't match, the names for intermediate files are based on the name of the module from which they are derived, not on the source file name.

Files whose names end in '`.int`', '`.int2`', '`.int0`' and '`.int3`' are interface files; these are generated automatically by the compiler, using the '`--make-interface`' (or '`--make-int`'), '`--make-private-interface`' (or '`--make-priv-int`') and '`--make-short-interface`' (or '`--make-short-int`') options. (Invoking the compiler with '`--make-interface`' generates both '`.int`' and '`.int2`' files.) Files whose names end in '`.opt`' are used for inter-module optimization, and are created using the '`--make-optimization-interface`' (or '`--make-opt-int`') option. Similarly, files whose name ends in '`.trans_opt`' are used for transitive inter-module optimization, and are created using the '`--make-transitive-optimization-interface`' (or '`--make-trans-opt`') option.

Since the interface of a module changes less often than its implementation, the '`.int`', '`.int0`', '`.int2`', '`.int3`', '`.opt`', and '`.trans_opt`' files will often remain unchanged when they are rebuilt. To avoid unnecessary recompilations of the clients of the module, the timestamps on these files are updated only if their contents change. Files with the suffixes

‘`.date`’, ‘`.date0`’, ‘`.date3`’, ‘`.optdate`’, and ‘`.trans_opt_date`’ serve as timestamp files, to be used when deciding whether the interface files need to be regenerated: they each record the last time when the file or files it represents has last been checked for being up to date.

Files ending in ‘`.d`’ are automatically generated Makefile fragments which contain the dependencies for a module. Files ending in ‘`.dep`’ are automatically generated Makefile fragments which contain the rules for an entire program. Files ending in ‘`.dv`’ are automatically generated Makefile fragments which contain variable definitions for an entire program.

As usual, ‘`.c`’ files are C source code, and ‘`.o`’ files are object code. In addition, ‘`.pic_o`’ files are object code files that contain position-independent code (PIC). ‘`.lpic_o`’ files are object code files that can be linked with shared libraries, but don’t necessarily contain position-independent code themselves. ‘`.mh`’ and ‘`.mih`’ files are C header files generated by the Mercury compiler. Their non-standard extensions are necessary to avoid conflicts with system header files.

‘`.java`’ files are Java source code, ‘`.class`’ files are Java bytecode, and ‘`.jar`’ files are Java archives.

‘`.cs`’ files are C# source code, ‘`.dll`’ files are library assemblies and, when compiling to C#, ‘`.exe`’ files are process assemblies.

When smart recompilation (see Section 9.5 [Auxiliary output options], page 101) is enabled, ‘`.c_date`’, ‘`.cs_date`’ and ‘`.java_date`’ files perform the same function for ‘`.c`’, ‘`.cs`’ and ‘`.java`’ files respectively as e.g. ‘`.date3`’ files do for ‘`.int3`’ files: when smart recompilation works out that a module does not need to be recompiled, it updates the timestamp of the *timestamp file* for the target file, but leaves the timestamp of the target file itself unchanged.

‘`.used`’ files contain dependency information for smart recompilation (see Section 9.5 [Auxiliary output options], page 101).

3 Using the Mercury compiler

The Mercury compiler is called ‘`mmc`’ (for “Melbourne Mercury Compiler”). Some of its options (e.g. ‘`-c`’, ‘`-o`’, and ‘`-I`’) have a similar meaning to that in other compilers.

(Note that on Microsoft Windows systems, the name ‘`mmc`’ is also used by the executable for the Microsoft Management Console. To avoid the name clash on these systems, you can either invoke the Mercury compiler by the alternative name ‘`mercury`’, or adjust your `PATH` to ensure that the ‘`bin`’ directory precedes the Windows system directory.)

Arguments to ‘`mmc`’ may be either file names (ending in ‘`.m`’), or module names, with ‘`.`’ (rather than ‘`__`’) as the module qualifier. For a module name such as ‘`foo.bar.baz`’, the compiler will look for the source in the file named ‘`foo.bar.baz.m`’. To make the compiler look for a module in another file, use ‘`mmc -f` *`sources-files`*’ to generate a mapping from module name to file name, where *sources-files* is the list of source files in the directory (see Section 9.4 [Output options], page 98).

Arguments to ‘`mmc`’ may also be in ‘`@file`’. The ‘`@file`’ argument is replaced with arguments representing the contents of the file. This argument processing is done recursively. The contents of the ‘`@file`’ is split into arguments one per line in the file.

There are two ways to compile a program that consists of a single source file. The first way, which works only when targeting C, is to issue the command

```
mmc filename.m
```

The second way, which works regardless of target language, is to issue the command

```
mmc --make filename
```

There are also two ways to build programs that consist of more than one source file: using Mmake (see Chapter 5 [Using Mmake], page 5), which only works when targeting C, or using '`mmc --make`'.

We strongly recommend that programmers use '`mmc --make`' rather than invoking '`mmc`' directly, because '`mmc --make`' is generally easier to use and avoids unnecessary recompilation.

When targeting C on systems that do not require an executable file extension, '`mmc`' will put the executable into a file called '*`filename`*'; on systems (such as Windows) that use '`.exe`' as the file extension for executables, '`mmc`' will put the executable into a file called '*`filename`*`.exe`'.

When targeting C#, '`mmc`' will generate a process assembly called '*`filename`*`.exe`'. On Windows, this process assembly can be run directly. On non-Windows systems, '`mmc`' will also generate a shell script called '*`filename`*' that invokes the CLI execution environment on the process assembly. (See the file '`README.CSharp`' included in the Mercury distribution for further details.)

When targeting Java, '`mmc`' will package up all of the class files for the executable into a Java archive (JAR) named '*`filename`*`.jar`'. If you are using the Windows command line interpreter '`cmd.exe`', '`mmc`' will generate a batch file called '*`filename`*`.bat`' that invokes the program using the Java interpreter. Otherwise, it will generate a shell script called '*`filename`*' that also invokes the program using the Java interpreter.

If you use Mmake or '`mmc --make`', then you do not need to understand the details of how the Mercury implementation goes about building programs. Thus you may wish to skip the rest of this chapter.

To compile a source file to object code via C without creating an executable, use the command

```
mmc -c filename.m
```

Since targeting C is the default, this tells '`mmc`' to generate C code, and then invoke the configured C compiler to translate that to object code. '`mmc`' puts the generated C code into a file called '*`module`*`.c`' and the generated object code into a file called '*`module`*`.o`', where *module* is the name of the Mercury module defined in '*`filename`*`.m`'. If the source file contains nested modules, then each submodule will get compiled to separate C and object files.

Before you can compile a module, you must make the interface files for the modules that it imports (directly or indirectly). You can create the interface files for one or more source files using the following commands:

```
mmc --make-short-int filename1.m filename2.m ...
mmc --make-priv-int filename1.m filename2.m ...
mmc --make-int filename1.m filename2.m ...
```

The first command builds (or rebuilds) the '`.int3`' file of each module contained in the named source files. The second command builds (or rebuilds) the '`.int0`' file of each module contained in the named source files. (Note that only modules that have submodules need '`.int0`' files.) The third command builds (or rebuilds) both the '`.int`' and '`.int2`' file of each module contained in the named source files.

If you are going to compile with '`--intermodule-optimization`' enabled, then you also need to create the optimization interface files.

```
mmc --make-opt-int filename1.m filename2.m ...
```

If you are going to compile with '`--transitive-intermodule-optimization`' enabled, then you also need to create the transitive optimization files.

```
mmc --make-trans-opt filename1.m filename2.m ...
```

After you have made all the interface files, one way to create an executable for a multi-module program is to compile all the modules at the same time using the command

```
mmc filename1.m filename2.m ...
```

This will by default put the resulting executable in '*filename1*', though you can use the '`-o` *filename*' option to specify a different name for the output file, if you so desire.

The other way to create an executable for a multi-module program is to compile each module separately using '`mmc -c`', and then link the resulting object files together. The linking is a two stage process.

First, you must create and compile an *initialization file*, which is a C source file containing calls to automatically generated initialization functions contained in the C code of the modules of the program:

```
c2init module1.c module2.c ... > main-module_init.c,
mgnuc -c main-module_init.c
```

The '`c2init`' command line must contain the name of the C file of every module in the program. The order of the arguments is not important. The '`mgnuc`' command is the Mercury GNU C compiler; it is a shell script that invokes the configured C compiler with the options appropriate for compiling the C programs generated by Mercury. (In the early days of the Mercury project, the configured C compiler was almost always GCC, which is why the name of the script is what it is, but the script itself will work with clang or MSVC as well.)

You then link the object code of each module with the object code of the initialization file to yield the executable:

```
ml -o main-module module1.o module2.o ... main_module_init.o
```

'`ml`', the Mercury linker, is another shell script that invokes a C compiler with options appropriate for Mercury, this time for linking. '`ml`' also pipes any error messages from the linker through '`mdemangle`', the Mercury symbol demangler, so that any error messages refer to predicate and function names from the Mercury source code rather than to the names used in the intermediate C code.

The above command puts the executable in the file '*main-module*'. The same command line without the '`-o`' option would put the executable into the file '`a.out`'.

'`mmc`' and '`ml`' both accept a '`-v`' (verbose) option. You can use that option to see what is actually going on. For the full set of options of '`mmc`', see Chapter 9 [Invocation], page 89.

4 Running programs

Once you have created an executable for a Mercury program, you can go ahead and execute it. You may however wish to specify certain options to the Mercury runtime system. The Mercury runtime accepts options via the `MERCURY_OPTIONS` environment variable. The most useful of these are the options that set the size of the stacks. (For the full list of available options, see Chapter 10 [Environment], page 136.)

When targeting Java or C#, and when targeting high-level C ('`mmc --high-level-code`'), stack management is the responsibility of the target language's compiler. When targeting low-level C ('`mmc`' without –high-level-code), stack management is the responsibility of '`mmc`' and of the C runtime system. This backend uses two stacks, the det stack and the nondet stack. With '`mmc --stack-segments`', both of these stacks will grow and shrink automatically as needed. Without '`--stack-segments`', their size is fixed at program start-up. The default size is 4096k times the word size (in bytes) for the det stack and 64k times the word size (in bytes) for the nondet stack, but these can be overridden with the '`--detstack-size`' and '`--nondetstack-size`' options, whose arguments are the desired sizes of the det and nondet stacks respectively, in units of kilobytes. On operating systems that provide the appropriate support, the Mercury runtime will ensure that stack overflow is trapped by the virtual memory system.

With conservative garbage collection (the default), the heap will start out with a zero size, and will be dynamically expanded as needed. When not using conservative garbage collection, the heap has a fixed size like the stacks. The default size is 8Mb times the word size (in bytes), but this can be overridden with the '`--heap-size`' option.

5 Using Mmake

Mmake, short for "Mercury Make", is a tool for building Mercury programs. The same functionality is now provided in '`mmc`' directly by using the '`--make`' option:

```
mmc --make main-module
```

The usage of Mmake is discouraged, not least because it works only when targeting C.

Mmake is built on top of GNU Make. With Mmake, building even a complicated Mercury program consisting of a number of modules is as simple as

```
mmc -f source-files
mmake main-module.depend
mmake main-module
```

Mmake only recompiles those files that need to be recompiled, based on automatically generated dependency information. Most of the dependencies are stored in '`.d`' files that are automatically recomputed every time you recompile, so they are never out-of-date. A little bit of the dependency information is stored in '`.dep`' and '`.dv`' files which are more expensive to recompute. The '`mmake main-module.depend`' command which recreates the '`main-module.dep`' and '`main-module.dv`' files needs to be repeated only when you either add a module to your program or remove a module from it. There is no danger of getting an inconsistent executable if you forget this step — instead you will get a compile or link error.

The '`mmc -f`' step above is only required if there are any source files for which the file name does not match the module name. '`mmc -f`' generates a file named '`Mercury.modules`' containing a mapping from module name to source file. The '`Mercury.modules`' file must be updated when a source file for which the file name does not match the module name is added to or removed from the directory.

'`mmake`' allows you to build more than one program in the same directory. Each program must have its own '`.dep`' and '`.dv`' files, and therefore you must run '`mmake` *`program`*`.depend`' for each program. The '`Mercury.modules`' file is used for all programs in the directory.

If there is a file called '`Mmake`' or '`Mmakefile`' in the current directory, Mmake will include that file in its automatically generated Makefile. The '`Mmake`' file can override the default values of various variables used by Mmake's builtin rules, or it can add additional rules, dependencies, and actions.

Mmake's builtin rules are defined by the file '*`prefix`*`/lib/mercury/mmake/Mmake.rules`' (where *prefix* is '`/usr/local/mercury-`*`version`*' by default, and *version* is the version number, e.g. '`0.6`'), as well as the rules and variables in the automatically generated '`.dep`' and '`.dv`' files. These rules define the following targets:

'*`main-module`*`.depend`'
: Creates the files '*`main-module`*`.dep`' and '*`main-module`*`.dv`' from '*`main-module`*`.m`' and the modules it imports. This step must be performed first. It is also required whenever you wish to change the level of inter-module optimization performed (see Section 9.10.1 [Overall optimization options], page 117).

'*`main-module`*`.all_ints`'
: Ensure that the interface files for *main-module* and its imported modules are up-to-date. (If the underlying '`make`' program does not handle transitive dependencies, this step may be necessary before attempting to make '*`main-module`*' or '*`main-module`*`.check`'; if the underlying '`make`' is GNU Make, this step should not be necessary.)

'*`main-module`*`.check`'
: Perform semantic checking on *main-module* and its imported modules. Error messages are placed in '`.err`' files.

'*`main-module`*'
: Compiles and links *main-module* using the Mercury compiler. Error messages are placed in '`.err`' files.

'`lib`*`main-module`*'
: Builds a library whose top-level module is *main-module*. This will build a static object library, a shared object library (for platforms that support it), and the necessary interface files. For more information, see Chapter 6 [Libraries], page 11.

'`lib`*`main-module`*`.install`'
: Builds and installs a library whose top-level module is *main-module*. This target will build and install a static object library and (for platforms that support it) a shared object library, for the default grade and also for the additional grades

specified in the `LIBGRADES` variable. It will also build and install the necessary interface files. The variable `INSTALL` specifies the name of the command to use to install each file, by default '`cp`'. The variable `INSTALL_MKDIR` specifies the command to use to create directories, by default '`mkdir -p`'.

'*main-module*`.clean`'
: Removes the automatically generated files that contain the compiled code of the program and the error messages produced by the compiler. Specifically, this will remove all the '`.c`', '`.o`', '`.pic_o`', '`.prof`', '`.used`', '`.mih`', and '`.err`' files belonging to the named *main-module* or its imported modules. Use this target whenever you wish to change compilation model (see Section 9.8 [Compilation model options], page 108). This target is also recommended whenever you wish to change the level of inter-module optimization performed (see Section 9.10.1 [Overall optimization options], page 117) in addition to the mandatory *main-module*.depend.

'*main-module*`.realclean`'
: Removes all the automatically generated files. In addition to the files removed by *main-module*.clean, this removes the '`.int`', '`.int0`', '`.int2`', '`.int3`', '`.opt`', '`.trans_opt`', '`.date`', '`.date0`', '`.date3`', '`.optdate`', '`.trans_opt_date`', '`.mh`' and '`.d`' files belonging to one of the modules of the program, and also the various possible executables, libraries and dependency files for the program as a whole — '*main-module*', 'lib*main-module*`.a`', 'lib*main-module*`.so`', 'lib*main-module*`.dylib`', '*main-module*`.init`', '*main-module*`.dep`' and '*main-module*`.dv`'.

'`clean`'
: This makes '*main-module*`.clean`' for every *main-module* for which there is a '*main-module*`.dep`' file in the current directory, as well as deleting the profiling files '`Prof.CallPair`', '`Prof.Counts`', '`Prof.Decl`', '`Prof.MemoryWords`' and '`Prof.MemoryCells`'.

'`realclean`'
: This makes '*main-module*`.realclean`' for every *main-module* for which there is a '*main-module*`.dep`' file in the current directory, as well as deleting the profiling files as per the '`clean`' target.

The variables used by the builtin rules (and their default values) are defined in the file '*prefix*`/lib/mercury/mmake/Mmake.vars`', however these may be overridden by user '`Mmake`' files. Some of the more useful variables are:

`MAIN_TARGET`
: The name of the default target to create if '`mmake`' is invoked with any target explicitly named on the command line.

`MC`
: The executable that invokes the Mercury compiler.

`GRADEFLAGS and EXTRA_GRADEFLAGS`
: Compilation model options (see Section 9.8 [Compilation model options], page 108) to pass to the Mercury compiler, linker, and other tools (in particular `mmc`, `mgnuc`, `ml`, and `c2init`).

`MCFLAGS and EXTRA_MCFLAGS`
: Options to pass to the Mercury compiler. (Note that compilation model options should be specified in `GRADEFLAGS`, not in `MCFLAGS`.)

`MGNUC`
: The executable that invokes the C compiler.

`MGNUCFLAGS and EXTRA_MGNUCFLAGS`
: Options to pass to the mgnuc script.

`CFLAGS and EXTRA_CFLAGS`
: Options to pass to the C compiler.

`JAVACFLAGS and EXTRA_JAVACFLAGS`
: Options to pass to the Java compiler (if you are using it).

`ML`
: The executable that invokes the linker.

`LINKAGE`
: Can be set to either '`shared`' to link with shared libraries, or '`static`' to always link statically. The default is '`shared`'. This variable only has an effect with '`mmc --make`'.

`MERCURY_LINKAGE`
: Can be set to either '`shared`' to link with shared Mercury libraries, or '`static`' to always link with the static versions of Mercury libraries. The default is system dependent. This variable only has an effect with '`mmc --make`'. See Section 6.2.2 [Using installed libraries with mmc –make], page 12.

`MLFLAGS and EXTRA_MLFLAGS`
: Options to pass to the ml and c2init scripts. (Note that compilation model options should be specified in `GRADEFLAGS`, not in `MLFLAGS`.) These variables have no effect with '`mmc --make`'.

`LDFLAGS and EXTRA_LDFLAGS`
: Options to pass to the command used by the ml script to link executables (use `ml --print-link-command` to find out what command is used, usually the C compiler).

`LD_LIBFLAGS and EXTRA_LD_LIBFLAGS`
: Options to pass to the command used to by the ml script to link shared libraries (use `ml --print-shared-lib-link-command` to find out what command is used, usually the C compiler or the system linker, depending on the platform).

`MLLIBS and EXTRA_MLLIBS`
: A list of '`-l`' options specifying libraries used by the program (or library) that you are building. See Section 6.3.3 [Using libraries with Mmake], page 15. See Section 6.2.2 [Using installed libraries with mmc –make], page 12.

`MLOBJS and EXTRA_MLOBJS`
: A list of extra object files or archives to link into the program or library that you are building.

`C2INITFLAGS and EXTRA_C2INITFLAGS`
: Options to pass to the linker and the c2init program. `C2INITFLAGS` and `EXTRA_C2INITFLAGS` are obsolete synonyms for `MLFLAGS` and `EXTRA_MLFLAGS` (`ml` and

`c2init` take the same set of options). (Note that compilation model options and extra files to be processed by c2init should not be specified in `C2INITFLAGS` — they should be specified in `GRADEFLAGS` and `C2INITARGS`, respectively.)

`C2INITARGS and EXTRA_C2INITARGS`
: Extra files to be processed by c2init. These variables should not be used for specifying flags to c2init (those should be specified in `MLFLAGS`) since they are also used to derive extra dependency information.

`EXTRA_LIBRARIES`
: A list of extra Mercury libraries to link into any programs or libraries that you are building. Libraries should be specified using their base name; that is, without any '`lib`' prefix or extension. For example, the library including the files '`libfoo.a`' and '`foo.init`' would be referred to as just '`foo`'. See Section 6.3.3 [Using libraries with Mmake], page 15. See Section 6.2.2 [Using installed libraries with mmc –make], page 12.

`EXTRA_LIB_DIRS`
: A list of extra Mercury library directory hierarchies to search when looking for extra libraries. See Section 6.3.3 [Using libraries with Mmake], page 15. See Section 6.2.2 [Using installed libraries with mmc –make], page 12.

`INSTALL_PREFIX`
: The path to the root of the directory hierarchy where the libraries, etc. you are building should be installed. The default is to install in the same location as the Mercury compiler being used to do the install.

`INSTALL` The command used to install each file in a library. The command should take a list of files to install and the location to install them. The default command is '`cp`'.

`INSTALL_MKDIR`
: The command used to create each directory in the directory hierarchy where the libraries are to be installed. The default command is '`mkdir -p`'.

`LIBGRADES`
: A list of additional grades which should be built when installing libraries. The default is to install the Mercury compiler's default set of grades. Note that this may not be the set of grades in which the standard libraries were actually installed. Note also that any `GRADEFLAGS` settings will also be applied when the library is built in each of the listed grades, so you may not get what you expect if those options are not subsumed by each of the grades listed.

`LIB_LINKAGES`
: A list of linkage styles ('`shared`' or '`static`') for which libraries should be built and installed. The default is to install libraries for both static and shared linking. This variable only has an effect with '`mmc --make`'.

Other variables also exist — see '*`prefix`*`/lib/mercury/mmake/Mmake.vars`' for a complete list.

If you wish to temporarily change the flags passed to an executable, rather than setting the various '`FLAGS`' variables directly, you can set an '`EXTRA_`' variable. This is particularly

intended for use where a shell script needs to call mmake and add an extra parameter, without interfering with the flag settings in the 'Mmakefile'.

For each of the variables for which there is a version with an 'EXTRA_' prefix, there is also a version with an 'ALL_' prefix that is defined to include both the ordinary and the 'EXTRA_' version. If you wish to *use* the values of any of these variables in your Mmakefile (as opposed to *setting* the values), then you should use the 'ALL_' version.

It is also possible to override these variables on a per-file basis. For example, if you have a module called say 'bad_style.m' which triggers lots of compiler warnings, and you want to disable the warnings just for that file, but keep them for all the other modules, then you can override MCFLAGS just for that file. This is done by setting the variable 'MCFLAGS-bad_style', as shown here:

```
MCFLAGS-bad_style = --inhibit-warnings
```

Mmake has a few options, including '--use-subdirs', '--use-mmc-make', '--save-makefile', '--verbose', and '--no-warn-undefined-vars'. For details about these options, see the man page or type 'mmake --help'.

Finally, since Mmake is built on top of GNU Make, you can also make use of the features and options supported by the underlying Make. In particular, GNU Make has support for running jobs in parallel, which is very useful if you have a machine with more than one CPU.

As an alternative to Mmake, the Mercury compiler now contains a significant part of the functionality of Mmake, using 'mmc''s '--make' option.

The advantages of the 'mmc --make' over Mmake are that there is no 'mmake depend' step and the dependencies are more accurate.

Note that '--use-subdirs' is automatically enabled if you specify 'mmc --make'.

The Mmake variables above can be used by 'mmc --make' if they are set in a file called 'Mercury.options'. The 'Mercury.options' file has the same syntax as an Mmakefile, but only variable assignments and 'include' directives are allowed. All variables in 'Mercury.options' are treated as if they are assigned using ':='. Variables may also be set in the environment, overriding settings in options files.

'mmc --make' can be used in conjunction with Mmake. This is useful for projects which include source code written in languages other than Mercury. The '--use-mmc-make' Mmake option disables Mmake's Mercury-specific rules. Mmake will then process source files written in other languages, but all Mercury compilation will be done by 'mmc --make'. The following variables can be set in the Mmakefile to control the use of 'mmc --make'.

MERCURY_MAIN_MODULES
: The top-level modules of the programs or libraries being built in the directory. This must be set to tell Mmake to use 'mmc --make' to rebuild the targets for the main modules even if those files already exist.

MC_BUILD_FILES
: Other files which should be built with 'mmc --make'. This should only be necessary for header files generated by the Mercury compiler which are included by the user's C source files.

MC_MAKE_FLAGS and EXTRA_MC_MAKE_FLAGS
: Options to pass to the Mercury compiler only when using 'mmc --make'.

The following variables can also appear in options files but are *only* supported by '`mmc --make`'.

`GCC_FLAGS`
: Options to pass to the C compiler, but only if the C compiler is GCC. If the C compiler is not GCC then this variable is ignored. These options will be passed *after* any options given by the '`CFLAGS`' variable.

`CLANG_FLAGS`
: Options to pass to the C compiler, but only if the C compiler is clang. If the C compiler is not clang then this variable is ignored. These options will be passed *after* any options given by the '`CFLAGS`' variable.

`MSVC_FLAGS`
: Options to pass to the C compiler, but only if the C compiler is Microsoft Visual C. If the C compiler is not Visual C then this variable is ignored. These options will be passed *after* any options given by the '`CFLAGS`' variable.

6 Libraries

Often you will want to use a particular set of Mercury modules in more than one program. The Mercury implementation includes support for developing libraries, i.e. sets of Mercury modules intended for reuse. It allows separate compilation of libraries and, on many platforms, it supports shared object libraries.

6.1 Writing libraries

A Mercury library is identified by a top-level module, which should contain all of the modules in that library as submodules. It may be as simple as this '`mypackage.m`' file:

```
:- module mypackage.
:- interface.
:- include_module foo.
:- include_module bar.
:- include_module baz.
```

This defines a module '`mypackage`' containing submodules '`mypackage.foo`', '`mypackage.bar`', and '`mypackage.baz`'.

It is also possible to build libraries of unrelated modules, so long as the top-level module imports all the necessary modules. For example:

```
:- module blah.
:- implementation.
:- import_module fee.
:- import_module fie.
:- import_module foe.
:- import_module fum.
```

This example defines a module '`blah`', which has no functionality of its own, and which is just used for grouping the unrelated modules '`fee`', '`fie`', '`foe`', and '`fum`'. To avoid a warning about the interface of this module being empty, this module would have to be

compiled with '`--no-warn-nothing-exported`'. Alternatively, the library could of course just export something, such as a predicate that returns its version number.

Generally it is better style for each library to consist of a single module which encapsulates its submodules, as in the first example, rather than just a group of unrelated modules, as in the second example.

6.2 Building with mmc –make

6.2.1 Building and installing libraries with mmc –make

To build a library from the source '`mypackage.m`' (and other included modules), run '`mmc`' with the following arguments:

```
mmc --make libmypackage
```

'`mmc`' will create static (non-shared) object libraries and, on most platforms, shared object libraries; however, we do not yet support the creation of dynamic link libraries (DLLs) on Windows. Use the '`mmc`' option '`--lib-linkage`' to specify which versions of the library should be created: '`shared`' or '`static`'. The '`--lib-linkage`' option can be specified multiple times. In our example, the files '`libmypackage.a`' and '`libmypackage.so`' should appear in the current directory. (On macOS '`libmypackage.dylib`' will appear instead of '`libmypackage.so`'.)

Other programs can more easily use a library that is installed. To install the library, issue the following command:

```
mmc --make --install-prefix <dir> libmypackage.install
```

'`mmc`' will create the directory '`<dir>/lib/mercury`' and install the library there. The library will be compiled in all valid grades and with all interface files. Because several grades are usually compiled, installing the library can be a lengthy process. You can specify the set of installed grades using the option '`--no-libgrade`' followed by '`--libgrade <grade>`' for all grades you wish to install.

If no '`--install-prefix <dir>`' is specified, the library will be installed in the standard location, next to the Mercury standard library.

6.2.2 Using installed libraries with mmc –make

Once a library is installed, it can be used by running '`mmc`' with the following options:

```
mmc ... --ml mypackage ... --ml myotherlib ... --ml my_yet_another_lib ...
```

If a library was installed in a different place (using '`--install-prefix <dir>`'), you will also need to add this option:

```
mmc ... --mld <dir>/lib/mercury ...
```

Note that '`/lib/mercury`' has to be added to the searched path. The '`--mld`' option can be used several times to add more directories to the library search path.

You can also specify whether to link executables with the shared or static versions of Mercury libraries using '`--mercury-linkage shared`' or '`--mercury-linkage static`'.

6.2.3 Using non-installed libraries with mmc –make

Suppose the user wants to link against library '`mypackage`' without installing the library. The source of the library is stored in the directory '`<dir>`' and that the library has been properly built using '`mmc --make libmypackage`'. To link against the library, the following options have to be added to '`mmc`':

```
mmc ... --search-lib-files-dir <dir> \
        --init-file <dir>/mypackage.init \
        --link-object <dir>/libmypackage.a \
    ...
```

Note that the option '`--ml`' is not used.

You need to make sure the library '`libmypackage.a`' and the main program were compiled in the same grade.

If you need to experiment with more grades, be sure to build the library in all the grades (building several times using '`mmc --grade <grade> --make libmypackage`') and use the '`libmypackage.a`' that is compatible with your main program's grade:

```
mmc ... --use-grade-subdirs \
        --grade <grade> \
        --search-lib-files-dir <dir> \
        --init-file <dir>/mypackage.init \
        --link-object <dir>/Mercury/<grade>/*/Mercury/lib/libmypackage.a \
    ...
```

6.3 Building with Mmake

6.3.1 Building libraries with Mmake

Generally Mmake will do most of the work of building libraries automatically. Here is a sample `Mmakefile` for creating a library.

```
MAIN_TARGET = libmypackage
depend: mypackage.depend
```

The Mmake target '`lib`*`foo`*' is a builtin target for creating a library whose top-level module is '*`foo`*`.m`'. The automatically generated Mmake rules for the target '`lib`*`foo`*' will create all the files needed to use the library. (You will need to run '`mmake `*`foo`*`.depend`' first to generate the module dependency information.)

Mmake will create static (non-shared) object libraries and, on most platforms, shared object libraries; however, we do not yet support the creation of dynamic link libraries (DLLs) on Windows. Static libraries are created using the standard tools `ar` and `ranlib`. Shared libraries are created using the '`--make-shared-lib`' option to '`ml`'. The automatically generated Make rules for '`libmypackage`' will look something like this:

```
libmypackage: libmypackage.a libmypackage.so \
                $(mypackage.ints) $(mypackage.int3s) \
                $(mypackage.opts) $(mypackage.trans_opts) mypackage.init

libmypackage.a: $(mypackage.os)
        rm -f libmypackage.a
```

```
        $(AR) $(ARFLAGS) libmypackage.a $(mypackage.os) $(MLOBJS)
        $(RANLIB) $(RANLIBFLAGS) libmypackage.a

libmypackage.so: $(mypackage.pic_os)
        $(ML) $(MLFLAGS) --make-shared-lib -o libmypackage.so \
                $(mypackage.pic_os) $(MLPICOBJS) $(MLLIBS)

libmypackage.init:
        ...

clean:
        rm -f libmypackage.a libmypackage.so
```

If necessary, you can override the default definitions of the variables such as '`ML`', '`MLFLAGS`', '`MLPICOBJS`', and '`MLLIBS`' to customize the way shared libraries are built. Similarly '`AR`', '`ARFLAGS`', '`MLOBJS`', '`RANLIB`', and '`RANLIBFLAGS`' control the way static libraries are built. (The '`MLOBJS`' variable is supposed to contain a list of additional object files to link into the library, while the '`MLLIBS`' variable should contain a list of '`-l`' options naming other libraries used by this library. '`MLPICOBJS`' is described below.)

Note that to use a library, as well as the shared or static object library, you also need the interface files. That is why the '`libmypackage`' target builds '`$(mypackage.ints)`' and '`$(mypackage.int3s)`'. If the people using the library are going to use intermodule optimization, you will also need the intermodule optimization interfaces. The '`libmypackage`' target will build '`$(mypackage.opts)`' if '`--intermodule-optimization`' is specified in your '`MCFLAGS`' variable (this is recommended). Similarly, if the people using the library are going to use transitive intermodule optimization, you will also need the transitive intermodule optimization interfaces ('`$(mypackage.trans_opt)`'). These will be built if '`--trans-intermod-opt`' is specified in your '`MCFLAGS`' variable.

In addition, with certain compilation grades, programs will need to execute some startup code to initialize the library; the '`mypackage.init`' file contains information about initialization code for the library. The '`libmypackage`' target will build this file.

On some platforms, shared objects must be created using position independent code (PIC), which requires passing some special options to the C compiler. On these platforms, `Mmake` will create '`.pic_o`' files, and '`$(mypackage.pic_os)`' will contain a list of the '`.pic_o`' files for the library whose top-level module is '`mypackage`'. In addition, '`$(MLPICOBJS)`' will be set to '`$MLOBJS`' with all occurrences of '`.o`' replaced with '`.pic_o`'. On other platforms, position independent code is the default, so '`$(mypackage.pic_os)`' will just be the same as '`$(mypackage.os)`', which contains a list of the '`.o`' files for that module, and '`$(MLPICOBJS)`' will be the same as '`$(MLOBJS)`'.

6.3.2 Installing libraries with Mmake

'`mmake`' has support for alternative library directory hierarchies. These have the same structure as the '*`prefix`*`/lib/mercury`' tree, including the different subdirectories for different grades and different machine architectures.

In order to support the installation of a library into such a tree, you simply need to specify (e.g. in your '`Mmakefile`') the path prefix and the list of grades to install:

```
INSTALL_PREFIX = /my/install/dir
LIBGRADES = asm_fast asm_fast.gc.tr.debug
```

This specifies that libraries should be installed in '`/my/install/dir/lib/mercury`', in the default grade plus '`asm_fast`' and '`asm_fast.gc.tr.debug`'. If '`INSTALL_PREFIX`' is not specified, '`mmake`' will attempt to install the library in the same place as the standard Mercury libraries. If '`LIBGRADES`' is not specified, '`mmake`' will use the Mercury compiler's default set of grades, which may or may not correspond to the actual set of grades in which the standard Mercury libraries were installed.

To actually install a library '`lib`*foo*', use the '`mmake`' target '`lib`*foo*`.install`'. This also installs all the needed interface files, and (if intermodule optimisation is enabled) the relevant intermodule optimisation files.

One can override the list of grades to install for a given library '`lib`*foo*' by setting the '`LIBGRADES-`*foo*' variable, or add to it by setting '`EXTRA_LIBGRADES-`*foo*'.

The command used to install each file is specified by '`INSTALL`'. If '`INSTALL`' is not specified, '`cp`' will be used.

The command used to create directories is specified by '`INSTALL_MKDIR`'. If '`INSTALL_MKDIR`' is not specified, '`mkdir -p`' will be used.

Note that currently it is not possible to set the installation prefix on a library-by-library basis.

6.3.3 Using libraries with Mmake

Once a library is installed, using it is easy. Suppose the user wishes to use the library '`mypackage`' (installed in the tree rooted at '`/some/directory/mypackage`') and the library '`myotherlib`' (installed in the tree rooted at '`/some/directory/myotherlib`'). The user need only set the following Mmake variables:

```
EXTRA_LIB_DIRS = /some/directory/mypackage/lib/mercury \
                 /some/directory/myotherlib/lib/mercury
EXTRA_LIBRARIES = mypackage myotherlib
```

When using '`--intermodule-optimization`' with a library which uses the C interface, it may be necessary to add '`-I`' options to '`MGNUCFLAGS`' so that the C compiler can find any header files used by the library's C code.

Mmake will ensure that the appropriate directories are searched for the relevant interface files, module initialisation files, compiled libraries, etc.

Beware that the directory name that you must use in '`EXTRA_LIB_DIRS`' or as the argument of the '`--mld`' option is not quite the same as the name that was specified in the '`INSTALL_PREFIX`' when the library was installed — the name needs to have '`/lib/mercury`' appended.

One can specify extra libraries to be used on a program-by-program basis. For instance, if the program '`foo`' also uses the library '`mylib4foo`', but the other programs governed by the Mmakefile don't, then one can declare:

```
EXTRA_LIBRARIES-foo = mylib4foo
```

6.4 Libraries and the Java grade

To create or install a library in the Java grade, specify that you want to use the Java grade and use '`mmc --make`'. Mmake does *not* support Java targets.

Libraries are compiled to class files that are added to a Java archive (JAR) file whose name has the form '*`library-name`*`.jar`'.

6.5 Libraries and the C# grade

To create or install a library in the C# grade, specify that you want to use the C# grade and use '`mmc --make`'. Mmake does *not* support C# targets.

Libraries are complied to a dynamic link library assembly whose name has the form '*`library-name`*`.dll`'.

7 Debugging

7.1 Quick overview

This section gives a quick and simple guide to getting started with the debugger. The remainder of this chapter contains more detailed documentation.

To use the debugger, you must first compile your program with debugging enabled. You can do this by using one of the '`--debug`' or '`--decl-debug`' options when invoking '`mmc`', or by including '`GRADEFLAGS = --debug`' or '`GRADEFLAGS = --decl-debug`' in your '`Mmakefile`'.

```
bash$ mmc --debug hello.m
```

Once you have compiled with debugging enabled, you can use the '`mdb`' command to invoke your program under the debugger:

```
bash$ mdb ./hello arg1 arg2 ...
```

Any arguments (such as '`arg1 arg2 ...`' in this example) that you pass after the program name will be given as arguments to the program.

The debugger will print a start-up message and will then show you the first trace event, namely the call to `main/2`:

```
       1:      1  1 CALL pred hello:main/2-0 (det)
                        hello.m:13
mdb>
```

By hitting enter at the '`mdb>`' prompt, you can step through the execution of your program to the next trace event:

```
       2:      2  2 CALL pred io:write_string/3-0 (det)
                        io.m:2837 (hello.m:14)
mdb>
Hello, world
       3:      2  2 EXIT pred io:write_string/3-0 (det)
                        io.m:2837 (hello.m:14)
mdb>
```

For each trace event, the debugger prints out several pieces of information. The three numbers at the start of the display are the event number, the call sequence number, and the call depth. (You don't really need to pay too much attention to those.) They are followed by the event type (e.g. '`CALL`' or '`EXIT`'). After that comes the identification of the procedure in which the event occurred, consisting of the module-qualified name of the predicate or function to which the procedure belongs, followed by its arity, mode number and determinism. This may sometimes be followed by a "path" (see Section 7.3 [Tracing of Mercury programs], page 19). At the end is the file name and line number of the called procedure and (if available) also the file name and line number of the call.

The most useful `mdb` commands have single-letter abbreviations. The '`alias`' command will show these abbreviations:

```
mdb> alias
?       =>    help
EMPTY   =>    step
NUMBER  =>    step
P       =>    print *
b       =>    break
c       =>    continue
d       =>    stack
f       =>    finish
g       =>    goto
h       =>    help
p       =>    print
r       =>    retry
s       =>    step
v       =>    vars
```

The '`P`' or '`print *`' command will display the values of any live variables in scope. The '`f`' or '`finish`' command can be used if you want to skip over a call. The '`b`' or '`break`' command can be used to set breakpoints. The '`d`' or '`stack`' command will display the call stack. The '`quit`' command will exit the debugger.

That should be enough to get you started. But if you have GNU Emacs installed, you should strongly consider using the Emacs interface to '`mdb`' — see the following section.

For more information about the available commands, use the '`?`' or '`help`' command, or see Section 7.10 [Debugger commands], page 31.

7.2 GNU Emacs interface

As well as the command-line debugger, mdb, there is also an Emacs interface to this debugger. Note that the Emacs interface only works with GNU Emacs, not with XEmacs.

With the Emacs interface, the debugger will display your source code as you trace through it, marking the line that is currently being executed, and allowing you to easily set breakpoints on particular lines in your source code. You can have separate windows for the debugger prompt, the source code being executed, and for the output of the program being executed. In addition, most of the mdb commands are accessible via menus.

To start the Emacs interface, you first need to put the following text in the file '`.emacs`' in your home directory, replacing "/usr/local/mercury-1.0" with the directory that your Mercury implementation was installed in.

```
(setq load-path (cons (expand-file-name
  "/usr/local/mercury-1.0/lib/mercury/elisp")
  load-path))
(autoload 'mdb "gud" "Invoke the Mercury debugger" t)
```

Build your program with debugging enabled, as described in Section 7.1 [Quick overview], page 16 or Section 7.4 [Preparing a program for debugging], page 21. Then start up Emacs, e.g. using the command '`emacs`', and type *`M-x mdb RET`*. Emacs will then prompt you for the mdb command to invoke

```
Run mdb (like this): mdb
```

and you should type in the name of the program that you want to debug and any arguments that you want to pass to it:

```
Run mdb (like this): mdb ./hello arg1 arg2 ...
```

Emacs will then create several "buffers": one for the debugger prompt, one for the input and output of the program being executed, and one or more for the source files. By default, Emacs will split the display into two parts, called "windows", so that two of these buffers will be visible. You can use the command *`C-x o`* to switch between windows, and you can use the command *`C-x 2`* to split a window into two windows. You can use the "Buffers" menu to select which buffer is displayed in each window.

If you are using X-Windows, then it is a good idea to set the Emacs variable '`pop-up-frames`' to '`t`' before starting mdb, since this will cause each buffer to be displayed in a new "frame" (i.e. a new X window). You can set this variable interactively using the '`set-variable`' command, i.e. *`M-x set-variable RET pop-up-frames RET t RET`*. Or you can put '`(setq pop-up-frames t)`' in the '`.emacs`' file in your home directory.

For more information on buffers, windows, and frames, see the Emacs documentation.

Another useful Emacs variable is '`gud-mdb-directories`'. This specifies the list of directories to search for source files. You can use a command such as

```
M-x set-variable RET
gud-mdb-directories RET
(list "/foo/bar" "../other" "/home/guest") RET
```

to set it interactively, or you can put a command like

```
(setq gud-mdb-directories
  (list "/foo/bar" "../other" "/home/guest"))
```

in your '`.emacs`' file.

At each trace event, the debugger will search for the source file corresponding to that event, first in the same directory as the program, and then in the directories specified by the '`gud-mdb-directories`' variable. It will display the source file, with the line number corresponding to that trace event marked by an arrow ('`=>`') at the start of the line.

Several of the debugger features can be accessed by moving the cursor to the relevant part of the source code and then selecting a command from the menu. You can set a break point on a line by moving the cursor to the appropriate line in your source code (e.g. with

the arrow keys, or by clicking the mouse there), and then selecting the "Set breakpoint on line" command from the "Breakpoints" sub-menu of the "MDB" menu. You can set a breakpoint on a procedure by moving the cursor over the procedure name and then selecting the "Set breakpoint on procedure" command from the same menu. And you can display the value of a variable by moving the cursor over the variable name and then selecting the "Print variable" command from the "Data browsing" sub-menu of the "MDB" menu. Most of the menu commands also have keyboard short-cuts, which are displayed on the menu.

Note that '`mdb`''s '`context`' and '`user_event_context`' commands should not be used if you are using the Emacs interface, otherwise the Emacs interface won't be able to parse the file names and line numbers that '`mdb`' outputs, and so it won't be able to highlight the correct location in the source code.

7.3 Tracing of Mercury programs

The Mercury debugger is based on a modified version of the box model on which the four-port debuggers of most Prolog systems are based. Such debuggers abstract the execution of a program into a sequence, also called a *trace*, of execution events of various kinds. The four kinds of events supported by most Prolog systems (their *ports*) are

call A call event occurs just after a procedure has been called, and control has just reached the start of the body of the procedure.

exit An exit event occurs when a procedure call has succeeded, and control is about to return to its caller.

redo A redo event occurs when all computations to the right of a procedure call have failed, and control is about to return to this call to try to find alternative solutions.

fail A fail event occurs when a procedure call has run out of alternatives, and control is about to return to the rightmost computation to its left that still has possibly successful alternatives left.

Mercury also supports these four kinds of events, but not all events can occur for every procedure call. Which events can occur for a procedure call, and in what order, depend on the determinism of the procedure. The possible event sequences for procedures of the various determinisms are as follows.

nondet procedures
: a call event, zero or more repeats of (exit event, redo event), and a fail event

multi procedures
: a call event, one or more repeats of (exit event, redo event), and a fail event

semidet and cc_nondet procedures
: a call event, and either an exit event or a fail event

det and cc_multi procedures
: a call event and an exit event

failure procedures
: a call event and a fail event

erroneous procedures
: a call event

In addition to these four event types, Mercury supports *exception* events. An exception event occurs when an exception has been thrown inside a procedure, and control is about to propagate this exception to the caller. An exception event can replace the final exit or fail event in the event sequences above or, in the case of erroneous procedures, can come after the call event.

Besides the event types call, exit, redo, fail and exception, which describe the *interface* of a call, Mercury also supports several types of events that report on what is happening *internal* to a call. Each of these internal event types has an associated parameter called a path. The internal event types are:

cond
: A cond event occurs when execution reaches the start of the condition of an if-then-else. The path associated with the event specifies which if-then-else this is.

then
: A then event occurs when execution reaches the start of the then part of an if-then-else. The path associated with the event specifies which if-then-else this is.

else
: An else event occurs when execution reaches the start of the else part of an if-then-else. The path associated with the event specifies which if-then-else this is.

disj
: A disj event occurs when execution reaches the start of a disjunct in a disjunction. The path associated with the event specifies which disjunct of which disjunction this is.

switch
: A switch event occurs when execution reaches the start of one arm of a switch (a disjunction in which each disjunct unifies a bound variable with a different function symbol). The path associated with the event specifies which arm of which switch this is.

neg_enter
: A neg_enter event occurs when execution reaches the start of a negated goal. The path associated with the event specifies which negation goal this is.

neg_fail
: A neg_fail event occurs when a goal inside a negation succeeds, which means that its negation fails. The path associated with the event specifies which negation goal this is.

neg_success
: A neg_success event occurs when a goal inside a negation fails, which means that its negation succeeds. The path associated with the event specifies which negation goal this is.

A goal path is a sequence of path components separated by semicolons. Each path component is one of the following:

`c`*num*
: The *num*'th conjunct of a conjunction.

`d`*num*
: The *num*'th disjunct of a disjunction.

`s`*num*
: The *num*'th arm of a switch.

`?` The condition of an if-then-else.

`t` The then part of an if-then-else.

`e` The else part of an if-then-else.

`~` The goal inside a negation.

`q!` The goal inside an existential quantification or other scope that changes the determinism of the goal.

`q` The goal inside an existential quantification or other scope that doesn't change the determinism of the goal.

A goal path describes the position of a goal inside the body of a procedure definition. For example, if the procedure body is a disjunction in which each disjunct is a conjunction, then the goal path '`d2;c3;`' denotes the third conjunct within the second disjunct. If the third conjunct within the second disjunct is an atomic goal such as a call or a unification, then this will be the only goal with whose path has '`d2;c3;`' as a prefix. If it is a compound goal, then its components will all have paths that have '`d2;c3;`' as a prefix, e.g. if it is an if-then-else, then its three components will have the paths '`d2;c3;?;`', '`d2;c3;t;`' and '`d2;c3;e;`'.

Goal paths refer to the internal form of the procedure definition. When debugging is enabled (and the option '`--trace-optimized`' is not given), the compiler will try to keep this form as close as possible to the source form of the procedure, in order to make event paths as useful as possible to the programmer. Due to the compiler's flattening of terms, and its introduction of extra unifications to implement calls in implied modes, the number of conjuncts in a conjunction will frequently differ between the source and internal form of a procedure. This is rarely a problem, however, as long as you know about it. Mode reordering can be a bit more of a problem, but it can be avoided by writing single-mode predicates and functions so that producers come before consumers. The compiler transformation that potentially causes the most trouble in the interpretation of goal paths is the conversion of disjunctions into switches. In most cases, a disjunction is transformed into a single switch, and it is usually easy to guess, just from the events within a switch arm, just which disjunct the switch arm corresponds to. Some cases are more complex; for example, it is possible for a single disjunction to be transformed into several switches, possibly with other, smaller disjunctions inside them. In such cases, making sense of goal paths may require a look at the internal form of the procedure. You can ask the compiler to generate a file with the internal forms of the procedures in a given module by including the options '`-dfinal -Dpaths`' on the command line when compiling that module.

7.4 Preparing a program for debugging

When you compile a Mercury program, you can specify whether you want to be able to run the Mercury debugger on the program or not. If you do, the compiler embeds calls to the Mercury debugging system into the executable code of the program, at the execution points that represent trace events. At each event, the debugging system decides whether to give control back to the executable immediately, or whether to first give control to you, allowing you to examine the state of the computation and issue commands.

Mercury supports two broad ways of preparing a program for debugging. The simpler way is to compile a program in a debugging grade, which you can do directly by specifying

a grade that includes the word "debug" or "decldebug" (e.g. '`asm_fast.gc.debug`', or '`asm_fast.gc.decldebug`'), or indirectly by specifying one of the '`--debug`' or '`--decl-debug`' grade options to the compiler, linker, and other tools (in particular `mmc`, `mgnuc`, `ml`, and `c2init`). If you follow this way, and accept the default settings of the various compiler options that control the selection of trace events (which are described below), you will be assured of being able to get control at every execution point that represents a potential trace event, which is very convenient.

The "decldebug" grades improve declarative debugging by allowing the user to track the source of subterms (see Section 7.11.7 [Improving the search], page 73). Doing this increases the size of executables, so these grades should only be used when you need the subterm dependency tracking feature of the declarative debugger. Note that declarative debugging, with the exception of the subterm dependency tracking features, also works in the .debug grades.

The two drawbacks of using a debugging grade are the large size of the resulting executables, and the fact that often you discover that you need to debug a big program only after having built it in a non-debugging grade. This is why Mercury also supports another way to prepare a program for debugging, one that does not require the use of a debugging grade. With this way, you can decide, individually for each module, which of four trace levels, '`none`', '`shallow`', '`deep`', and '`rep`' you want to compile them with:

'`none`' A procedure compiled with trace level '`none`' will never generate any events.

'`deep`' A procedure compiled with trace level '`deep`' will always generate all the events requested by the user. By default, this is all possible events, but you can tell the compiler that you are not interested in some kinds of events via compiler options (see below). However, declarative debugging requires all events to be generated if it is to operate properly, so do not disable the generation of any event types if you want to use declarative debugging. For more details see Section 7.11 [Declarative debugging], page 66.

'`rep`' This trace level is the same as trace level '`deep`', except that a representation of the module is stored in the executable along with the usual debugging information. The declarative debugger can use this extra information to help it avoid asking unnecessary questions, so this trace level has the effect of better declarative debugging at the cost of increased executable size. For more details see Section 7.11 [Declarative debugging], page 66.

'`shallow`' A procedure compiled with trace level '`shallow`' will generate interface events if it is called from a procedure compiled with trace level '`deep`', but it will never generate any internal events, and it will not generate any interface events either if it is called from a procedure compiled with trace level '`shallow`'. If it is called from a procedure compiled with trace level '`none`', the way it will behave is dictated by whether its nearest ancestor whose trace level is not '`none`' has trace level '`deep`' or '`shallow`'.

The intended uses of these trace levels are as follows.

'`deep`' You should compile a module with trace level '`deep`' if you suspect there may be a bug in the module, or if you think that being able to examine what happens inside that module can help you locate a bug.

‘`rep`’ You should compile a module with trace level ‘`rep`’ if you suspect there may be a bug in the module, you wish to use the full power of the declarative debugger, and you are not concerned about the size of the executable.

‘`shallow`’ You should compile a module with trace level ‘`shallow`’ if you believe the code of the module is reliable and unlikely to have bugs, but you still want to be able to get control at calls to and returns from any predicates and functions defined in the module, and if you want to be able to see the arguments of those calls.

‘`none`’ You should compile a module with trace level ‘`none`’ only if you are reasonably confident that the module is reliable, and if you believe that knowing what calls other modules make to this module would not significantly benefit you in your debugging.

In general, it is a good idea for most or all modules that can be called from modules compiled with trace level ‘`deep`’ or ‘`rep`’ to be compiled with at least trace level ‘`shallow`’.

You can control what trace level a module is compiled with by giving one of the following compiler options:

‘`--trace shallow`’
This always sets the trace level to ‘`shallow`’.

‘`--trace deep`’
This always sets the trace level to ‘`deep`’.

‘`--trace rep`’
This always sets the trace level to ‘`rep`’.

‘`--trace minimum`’
In debugging grades, this sets the trace level to ‘`shallow`’; in non-debugging grades, it sets the trace level to ‘`none`’.

‘`--trace default`’
In debugging grades, this sets the trace level to ‘`deep`’; in non-debugging grades, it sets the trace level to ‘`none`’.

As the name implies, the last alternative is the default, which is why by default you get no debugging capability in non-debugging grades and full debugging capability in debugging grades. The table also shows that in a debugging grade, no module can be compiled with trace level ‘`none`’.

Important note: If you are not using a debugging grade, but you compile some modules with a trace level other than none, then you must also pass the ‘`--trace`’ (or ‘`-t`’) option to c2init and to the Mercury linker. If you are using Mmake, then you can do this by including ‘`--trace`’ in the ‘`MLFLAGS`’ variable.

If you are using Mmake, then you can also set the compilation options for a single module named *Module* by setting the Mmake variable ‘`MCFLAGS-`*Module*’. For example, to compile the file ‘`foo.m`’ with deep tracing, ‘`bar.m`’ with shallow tracing, and everything else with no tracing, you could use the following:

```
MLFLAGS     = --trace
MCFLAGS-foo = --trace deep
MCFLAGS-bar = --trace shallow
```

7.5 Tracing optimized code

By default, all trace levels other than '`none`' turn off all compiler optimizations that can affect the sequence of trace events generated by the program, such as inlining. If you are specifically interested in how the compiler's optimizations affect the trace event sequence, you can specify the option '`--trace-optimized`', which tells the compiler that it does not have to disable those optimizations. (A small number of low-level optimizations have not yet been enhanced to work properly in the presence of tracing, so the compiler disables these even if '`--trace-optimized`' is given.)

7.6 Mercury debugger invocation

The executables of Mercury programs by default do not invoke the Mercury debugger even if some or all of their modules were compiled with some form of tracing, and even if the grade of the executable is a debugging grade. This is similar to the behaviour of executables created by the implementations of other languages; for example the executable of a C program compiled with '`-g`' does not automatically invoke gdb or dbx etc when it is executed.

Unlike those other language implementations, when you invoke the Mercury debugger '`mdb`', you invoke it not just with the name of an executable but with the command line you want to debug. If something goes wrong when you execute the command

```
prog arg1 arg2 ...
```

and you want to find the cause of the problem, you must execute the command

```
mdb [mdb-options] prog arg1 arg2 ...
```

because you do not get a chance to specify the command line of the program later.

When the debugger starts up, as part of its initialization it executes commands from the following three sources, in order:

1. The file named by the `MERCURY_DEBUGGER_INIT` environment variable. Usually, '`mdb`' sets this variable to point to a file that provides documentation for all the debugger commands and defines a small set of aliases. However, if `MERCURY_DEBUGGER_INIT` is already defined when '`mdb`' is invoked, it will leave its value unchanged. You can use this override ability to provide alternate documentation. If the file named by `MERCURY_DEBUGGER_INIT` cannot be read, '`mdb`' will print a warning, since in that case, the usual online documentation will not be available.
2. The file named '`.mdbrc`' in your home directory. You can put your usual aliases and settings here.
3. The file named '`.mdbrc`' in the current working directory. You can put program-specific aliases and settings here.

mdb will ignore any lines starting with the character '`#`' in any of the above mentioned files.

mdb accepts the following options from the command line. The options should be given to mdb before the name of the executable to be debugged.

`-t` *file-name*, `--tty` *file-name*
: Redirect all of the I/O for the debugger to the device specified by *file-name*. The I/O for the program being debugged will not be redirected. This option

allows the contents of a file to be piped to the program being debugged and not to mdb. For example, on Linux the command

```
mdb -t /dev/tty ./myprog < myinput
```

will cause the contents of '`myinput`' to be piped to the program '`myprog`', but mdb will read its input from the terminal.

`-w, --window, --mdb-in-window`
Run mdb in a new window, with mdb's I/O going to that window, but with the program's I/O going to the current terminal. Note that this will not work on all systems.

`--program-in-window`
Run the program in a new window, with the program's I/O going to that window, but with mdb's I/O going to the current terminal. Note that input and output redirection will not work with the '`--program-in-window`' option. '`--program-in-window`' will work on most Unix systems running the X Window System, even those for which '`--mdb-in-window`' is not supported.

`-c` *`window-command`*`, --window-command` *`window-command`*
Specify the command used by the '`--program-in-window`' option for executing a command in a new window. The default such command is '`xterm -e`'.

7.7 Mercury debugger concepts

The operation of the Mercury debugger '`mdb`' is based on the following concepts.

break points
The user may associate a break point with some events that occur inside a procedure; the invocation condition of the break point says which events these are. The four possible invocation conditions (also called scopes) are:

- the call event,
- all interface events,
- all events, and
- the event at a specific point in the procedure.

The effect of a break point depends on the state of the break point.

- If the state of the break point is '`stop`', execution will stop and user interaction will start at any event within the procedure that matches the invocation conditions, unless the current debugger command has specifically disabled this behaviour (see the concept '`strict commands`' below).

- If the state of the break point is '`print`', the debugger will print any event within the procedure that matches the invocation conditions, unless the

current debugger command has specifically disabled this behaviour (see the concept '`print level`' below).

Neither of these will happen if the break point is disabled.

Every break point has a print list. Every time execution stops at an event that matches the breakpoint, mdb implicitly executes a print command for each element in the breakpoint's print list. A print list element can be the word '`goal`', which causes the goal to be printed as if by '`print goal`'; it can be the word '`*`', which causes all the variables to be printed as if by '`print *`'; or it can be the name or number of a variable, possibly followed (without white space) by a term path, which causes the specified variable or part thereof to be printed as if the element were given as an argument to the '`print`' command.

strict commands

When a debugger command steps over some events without user interaction at those events, the *strictness* of the command controls whether the debugger will stop execution and resume user interaction at events to which a break point with state '`stop`' applies. By default, the debugger will stop at such events. However, if the debugger is executing a strict command, it will not stop at an event just because a break point in the stop state applies to it.

If the debugger receives an interrupt (e.g. if the user presses control-C), it will stop at the next event regardless of what command it is executing at the time.

print level When a debugger command steps over some events without user interaction at those events, the *print level* controls under what circumstances the stepped over events will be printed.

- When the print level is '`none`', none of the stepped over events will be printed.
- When the print level is '`all`', all the stepped over events will be printed.
- When the print level is '`some`', the debugger will print the event only if a break point applies to the event.

Regardless of the print level, the debugger will print any event that causes execution to stop and user interaction to start.

default print level

The debugger maintains a default print level. The initial value of this variable is '`some`', but this value can be overridden by the user.

current environment
: Whenever execution stops at an event, the current environment is reset to refer to the stack frame of the call specified by the event. However, the '`up`', '`down`' and '`level`' commands can set the current environment to refer to one of the ancestors of the current call. This will then be the current environment until another of these commands changes the environment yet again or execution continues to another event.

paths in terms
: When browsing or printing a term, you can use `"^`'`n`'`"` to refer to the *n*th subterm of that term. If the term's type has named fields, you can use `"^`'`fname`'`"` to refer to the subterm of the field named '`fname`'. You can use several of these subterm specifications in a row to refer to subterms deep within the original term. For example, when applied to a list, `"^2"` refers to the tail of the list (the second argument of the list constructor), `"^2^2"` refers to the tail of the tail of the list, and `"^2^2^1"` refers to the head of the tail of the tail, i.e. to the third element of the list. You can think of terms as Unix directories, with constants (function symbols of arity zero) being plain files and function symbols of arity greater than zero being directories themselves. Each subterm specification such as `"^2"` goes one level down in the hierarchy. The exception is the subterm specification `"^.."`, which goes one level up, to the parent of the current directory.

held variables
: Normally, the only variables from the program accessible in the debugger are the variables in the current environment at the current program point. However, the user can *hold* variables, causing their values -or selected parts of their values- to stay available for the rest of the debugger session. All the commands that accept variable names also accept the names of held variables; users can ask for a held variable by prefixing the name of the held variable with a dollar sign.

user defined events
: Besides the builtin set of events, the Mercury debugger also supports events defined by the user. Each event appears in the source code of the Mercury program as a call prefixed by the keyword '`event`', with each argument of the call giving the value of an event *attribute*. Users can specify the set of user defined events that can appear in a program, and the names, types and order of the attributes of each kind of user defined event, by giving the name of an event set specification file to the compiler when compiling that program. For more details, see Section 7.8 [User defined events], page 28.

user defined event attributes
: Normally, the only variables from the program accessible in the debugger are the variables in the current environment at the current program point. However, if the current event is a user defined event, then the attributes of that event are also available. All the commands that accept variable names also accept the

names of attributes; users can ask for an attribute by prefixing the name of the attribute with an exclamation point.

procedure specification
Some debugger commands, e.g. '`break`', require a parameter that specifies a procedure. The procedure may or may not be a compiler-generated unify, compare or index procedure of a type constructor. If it is, the procedure specification has the following components in the following order:

- An optional prefix of the form '`unif*`', '`comp*`', '`indx*`' or '`init*`', that specifies whether the procedure belongs to a unify, compare, index or init predicate.
- An optional prefix of the form '*`module.`*' or '*`module__`*' that specifies the name of the module that defines the predicate or function to which the procedure belongs.
- The name of the type constructor.
- An optional suffix of the form '*`/arity`*' that specifies the arity of the type constructor.
- An optional suffix of the form '*`-modenum`*' that specifies the mode number of the procedure within the predicate or function to which the procedure belongs.

For other procedures, the procedure specification has the following components in the following order:

- An optional prefix of the form '`pred*`' or '`func*`' that specifies whether the procedure belongs to a predicate or a function.
- An optional prefix of the form '*`module`*`:`', '*`module.`*' or '*`module__`*' that specifies the name of the module that defines the predicate or function to which the procedure belongs.
- The name of the predicate or function to which the procedure belongs.
- An optional suffix of the form '*`/arity`*' that specifies the arity of the predicate or function to which the procedure belongs.
- An optional suffix of the form '*`-modenum`*' that specifies the mode number of the procedure within the predicate or function to which the procedure belongs.

7.8 User defined events

Besides the builtin set of events, the Mercury debugger also supports events defined by the user. The intention is that users can define one kind of event for each semantically important event in the program that is not captured by the standard builtin events, and can then generate those events at the appropriate point in the source code. Each event appears in the source code as a call prefixed by the keyword '`event`', with each argument of the call giving the value of an event *attribute*.

Users can specify the set of user defined events that can appear in a program, and the names, types and order of the attributes of each kind of user defined event, by giving the

name of an event set specification file to the compiler when compiling that program as the argument of the '`event-set-file-name`' option. This file should contain a header giving the event set's name, followed by a sequence of one or more event specifications, like this:

```
event set queens

event nodiag_fail(
    test_failed:    string,
    arg_b:          int,
    arg_d:          int,
    arg_list_len:   int synthesized by list_len_func(sorted_list),
    sorted_list:    list(int) synthesized by list_sort_func(arg_list),
    list_len_func:  function,
    list_sort_func: function,
    arg_list:       list(int)
)

event safe_test(
    test_list:      list(int)
)

event noargs
```

The header consists of the keywords '`event set`' and an identifier giving the event set name. Each event specification consists of the keyword '`event`', the name of the event, and, if the event has any attributes, a parenthesized list of those attributes. Each attribute's specification consists of a name, a colon and information about the attribute.

There are three kinds of attributes.

- For ordinary attributes, like '`arg_b`', the information about the attribute is the Mercury type of that attribute.
- For function attributes, like '`list_sort_func`', the information about the attribute is just the keyword '`function`'.
- For synthesized attributes, like '`sorted_list`', the information about the attribute is the type of the attribute, the keywords '`synthesized by`', and a description of the Mercury function call required to synthesize the value of the attribute. The synthesis call consists of the name of a function attribute and a list of the names of one or more argument attributes. Argument attributes cannot be function attributes; they may be either ordinary attributes, or previously synthesized attributes. A synthesized attribute is not allowed to depend on itself directly or indirectly, but there are no restrictions on the positions of synthesized attributes compared to the positions of the function attributes computing them or of the argument attributes of the synthesis functions.

The result types of function attributes are given by the types of the synthesized attributes they compute. The argument types of function attributes (and the number of those arguments) are given by the types of the arguments they are applied to. Each function attribute must be used to compute at least one synthesized attribute, otherwise there would be no way to compute its type. If it is used to compute more than one synthesized attribute, the result and argument types must be consistent.

Each event goal in the program must use the name of one of the events defined here as the predicate name of the call, and the call's arguments must match the types of that event's non-synthesized attributes. Given that B and N are integers and L is a list of integers, these event goals are fine,

```
event nodiag_fail("N - B", B, N, list.length, list.sort, [N | L]),
event safe_test([1, 2, 3])
```

but these goals

```
event nodiag_fail("N - B", B, N, list.sort, list.length, [N | L]),
event nodiag_fail("N - B", B, list.length, N, list.sort, [N | L]),
event safe_test([1], [2])
event safe_test(42)
event nonexistent_event(42)
```

will all generate errors.

The attributes of event calls are always input, and the event goal is always '`det`'.

7.9 I/O tabling

In Mercury, predicates that want to do I/O must take a di/uo pair of I/O state arguments. Some of these predicates call other predicates to do I/O for them, but some are *I/O primitives*, i.e. they perform the I/O themselves. The Mercury standard library provides a large set of these primitives, and programmers can write their own through the foreign language interface. An I/O action is the execution of one call to an I/O primitive.

In debugging grades, the Mercury implementation has the ability to automatically record, for every I/O action, the identity of the I/O primitive involved in the action and the values of all its arguments. The size of the table storing this information is proportional to the number of *tabled* I/O actions, which are the I/O actions whose details are entered into the table. Therefore the tabling of I/O actions is never turned on automatically; instead, users must ask for I/O tabling to start with the '`table_io start`' command in mdb.

The purpose of I/O tabling is to enable transparent retries across I/O actions. (The mdb '`retry`' command restores the computation to a state it had earlier, allowing the programmer to explore code that the program has already executed; see its documentation in the Section 7.10 [Debugger commands], page 31 section below.) In the absence of I/O tabling, retries across I/O actions can have bad consequences. Retry of a goal that reads some input requires that input to be provided twice; retry of a goal that writes some output generates duplicate output. Retry of a goal that opens a file leads to a file descriptor leak; retry of a goal that closes a file can lead to errors (duplicate closes, reads from and writes to closed files).

I/O tabling avoids these problems by making I/O primitives *idempotent*. This means that they will generate their desired effect when they are first executed, but reexecuting them after a retry won't have any further effect. The Mercury implementation achieves this by looking up the action (which is identified by an I/O action number) in the table and returning the output arguments stored in the table for the given action *without* executing the code of the primitive.

Starting I/O tabling when the program starts execution and leaving it enabled for the entire program run will work well for program runs that don't do lots of I/O. For program

runs that *do* lots of I/O, the table can fill up all available memory. In such cases, the programmer may enable I/O tabling with '`table_io start`' just before the program enters the part they wish to debug and in which they wish to be able to perform transparent retries across I/O actions, and turn it off with '`table_io stop`' after execution leaves that part.

The commands '`table_io start`' and '`table_io stop`' can each be given only once during an mdb session. They divide the execution of the program into three phases: before '`table_io start`', between '`table_io start`' and '`table_io stop`', and after '`table_io stop`'. Retries across I/O will be transparent only in the middle phase.

7.10 Debugger commands

When the debugger (as opposed to the program being debugged) is interacting with the user, the debugger prints a prompt and reads in a line of text, which it will interpret as its next command line. A command line consists of a single command, or several commands separated by semicolons. Each command consists of several words separated by white space. The first word is the name of the command, while any other words give options and/or parameters to the command.

A word may itself contain semicolons or whitespace if it is enclosed in single quotes (''). This is useful for commands that have other commands as parameters, for example '`view -w 'xterm -e'`'. Characters that have special meaning to '`mdb`' will be treated like ordinary characters if they are escaped with a backslash ('\'). It is possible to escape single quotes, whitespace, semicolons, newlines and the escape character itself.

Some commands take a number as their first parameter. For such commands, users can type '*number command*' as well as '*command number*'. The debugger will treat the former as the latter, even if the number and the command are not separated by white space.

7.10.1 Interactive query commands

`query` *`module1 module2 ...`*
`cc_query` *`module1 module2 ...`*
`io_query` *`module1 module2 ...`*

These commands allow you to type in queries (goals) interactively in the debugger. When you use one of these commands, the debugger will respond with a query prompt ('`?-`' or '`run <--`'), at which you can type in a goal; the debugger will then compile and execute the goal and display the answer(s). You can return from the query prompt to the '`mdb>`' prompt by typing the end-of-file indicator (typically control-D or control-Z), or by typing '`quit.`'.

The module names *module1*, *module2*, ... specify which modules will be imported. Note that you can also add new modules to the list of imports directly at the query prompt, by using a command of the form '`[`*`module`*`]`', e.g. '`[int]`'. You need to import all the modules that define symbols used in your query. Queries can only use symbols that are exported from a module; entities which are declared in a module's implementation section only cannot be used.

The three variants differ in what kind of goals they allow. For goals which perform I/O, you need to use '`io_query`'; this lets you type in the goal using DCG syntax. For goals which don't do I/O, but which have determinism '`cc_nondet`' or '`cc_multi`', you need to use '`cc_query`'; this finds only one solution to the specified goal. For all other goals, you can use plain '`query`', which finds all the solutions to the goal.

Goals can refer to variables in the current environment, which will be treated as inputs to the query. Any variables in the goal that do not exist in the current environment, and that do not start with an underscore, will be treated as outputs. For '`query`' and '`cc_query`', the debugger will print the bindings of output variables in the goal using '`io.write_cc`'. The goal must bind all of its output variables to ground terms, otherwise you will get a mode error.

The current implementation works by compiling the queries on-the-fly and then dynamically linking them into the program being debugged. Thus it may take a little while for your query to be executed. Each query will be written to a file named '`mdb_query.m`' in the current directory, so make sure you don't name your source file '`mdb_query.m`'. Note that dynamic linking may not be supported on some systems; if you are using a system for which dynamic linking is not supported, you will get an error message when you try to run these commands.

You may also need to build your program using shared libraries for interactive queries to work. See Chapter 6 [Libraries], page 11 for details of how to build with shared libraries.

7.10.2 Forward movement commands

`step [-NSans] [`*num*`]`

Steps forward *num* events. If this command is given at event *cur*, continues execution until event *cur* + *num*. The default value of *num* is 1.

The options '`-n`' or '`--none`', '`-s`' or '`--some`', '`-a`' or '`--all`' specify the print level to use for the duration of the command, while the options '`-S`' or '`--strict`' and '`-N`' or '`--nostrict`' specify the strictness of the command.

By default, this command is not strict, and it uses the default print level.

A command line containing only a number *num* is interpreted as if it were 'step *num*'.

An empty command line is interpreted as 'step 1'.

`goto [-NSans] num`

Continues execution until the program reaches event number *num*. If the current event number is larger than *num*, it reports an error.

The options '`-n`' or '`--none`', '`-s`' or '`--some`', '`-a`' or '`--all`' specify the print level to use for the duration of the command, while the options '`-S`' or '`--strict`' and '`-N`' or '`--nostrict`' specify the strictness of the command.

By default, this command is strict, and it uses the default print level.

`next [-NSans] [num]`

Continues execution until it reaches the next event of the *num*'th ancestor of the call to which the current event refers. The default value of *num* is zero, which means skipping to the next event of the current call. Reports an error if execution is already at the end of the specified call.

The options '`-n`' or '`--none`', '`-s`' or '`--some`', '`-a`' or '`--all`' specify the print level to use for the duration of the command, while the options '`-S`' or '`--strict`' and '`-N`' or '`--nostrict`' specify the strictness of the command.

By default, this command is strict, and it uses the default print level.

`finish [-NSans]`
`finish [-NSans] num`
`finish [-NSans] ('clentry'|'clique')`
`finish [-NSans] 'clparent'`

If invoked without arguments, continues execution until it reaches a final (EXIT, FAIL or EXCP) port of the current call. If invoked with the number *num* as argument, continues execution until it reaches a final port of the *num*'th ancestor of the call to which the current event refers. If invoked with the argument '`clentry`' or '`clique`', continues execution until it reaches a final port of the call that first entered into the clique of recursive calls of which the current call is a part. (If the current call is not recursive or mutually recursive with any other currently active call, it will skip to the end of the current call.) If the command is given the argument '`clparent`', it skips to the end of the first call outside the current call's clique. This will be the parent of the call that '`finish clentry`' would finish.

If invoked as '`finish clentry`', '`finish clique`' or '`finish clparent`', this command will report an error unless we have stack trace information about all of the current call's ancestors.

Also reports an error if execution is already at the desired port.

The options '`-n`' or '`--none`', '`-s`' or '`--some`', '`-a`' or '`--all`' specify the print level to use for the duration of the command, while the options '`-S`' or '`--strict`' and '`-N`' or '`--nostrict`' specify the strictness of the command.

By default, this command is strict, and it uses the default print level.

`exception [-NSans]`
Continues the program until execution reaches an exception event. Reports an error if the current event is already an exception event.

The options '`-n`' or '`--none`', '`-s`' or '`--some`', '`-a`' or '`--all`' specify the print level to use for the duration of the command, while the options '`-S`' or '`--strict`' and '`-N`' or '`--nostrict`' specify the strictness of the command.

By default, this command is strict, and it uses the default print level.

`return [-NSans]`
Continues the program until the program finished returning, i.e. until it reaches a port other than EXIT. Reports an error if the current event already refers to such a port.

The options '`-n`' or '`--none`', '`-s`' or '`--some`', '`-a`' or '`--all`' specify the print level to use for the duration of the command, while the options '`-S`' or '`--strict`' and '`-N`' or '`--nostrict`' specify the strictness of the command.

By default, this command is strict, and it uses the default print level.

`user [-NSans]`
Continues the program until the next user defined event.

The options '`-n`' or '`--none`', '`-s`' or '`--some`', '`-a`' or '`--all`' specify the print level to use for the duration of the command, while the options '`-S`' or '`--strict`' and '`-N`' or '`--nostrict`' specify the strictness of the command.

By default, this command is strict, and it uses the default print level.

`forward [-NSans]`
Continues the program until the program resumes forward execution, i.e. until it reaches a port other than REDO or FAIL. Reports an error if the current event already refers to such a port.

The options '`-n`' or '`--none`', '`-s`' or '`--some`', '`-a`' or '`--all`' specify the print level to use for the duration of the command, while the options '`-S`' or '`--strict`' and '`-N`' or '`--nostrict`' specify the strictness of the command.

By default, this command is strict, and it uses the default print level.

`mindepth [-NSans] depth`
: Continues the program until the program reaches an event whose depth is at least *depth*. Reports an error if the current event already refers to such a port.

  The options '`-n`' or '`--none`', '`-s`' or '`--some`', '`-a`' or '`--all`' specify the print level to use for the duration of the command, while the options '`-S`' or '`--strict`' and '`-N`' or '`--nostrict`' specify the strictness of the command.

  By default, this command is strict, and it uses the default print level.

`maxdepth [-NSans] depth`
: Continues the program until the program reaches an event whose depth is at most *depth*. Reports an error if the current event already refers to such a port.

  The options '`-n`' or '`--none`', '`-s`' or '`--some`', '`-a`' or '`--all`' specify the print level to use for the duration of the command, while the options '`-S`' or '`--strict`' and '`-N`' or '`--nostrict`' specify the strictness of the command.

  By default, this command is strict, and it uses the default print level.

`continue [-NSans]`
: Continues execution until it reaches the end of the program.

  The options '`-n`' or '`--none`', '`-s`' or '`--some`', '`-a`' or '`--all`' specify the print level to use for the duration of the command, while the options '`-S`' or '`--strict`' and '`-N`' or '`--nostrict`' specify the strictness of the command.

  By default, this command is not strict. The print level used by the command by default depends on the final strictness level: if the command is strict, it is '`none`', otherwise it is '`some`'.

7.10.3 Backward movement commands

`retry [-fio]`
`retry [-fio] num`
`retry [-fio] ('clentry'|'clique')`
`retry [-fio] 'clparent'`
: If the command is given no arguments, restarts execution at the call port of the call corresponding to the current event. If the command is given the number *num* as argument, restarts execution at the call port of the call corresponding to the *num*'th ancestor of the call to which the current event belongs. For example, if *num* is 1, it restarts the parent of the current call. If the command

is given the argument '`clentry`' or '`clique`', restarts execution at the call port of the call that first entered into the clique of recursive calls of which the current call is a part. (If the current call is not (mutually) recursive with any other currently active call, the restarted call will be the current call.) If the command is given the argument '`clparent`', restarts execution at the call port of the first call outside the current call's clique. This will be the parent of the call that '`retry clentry`' would restart.

If invoked as '`retry clentry`', '`retry clique`' or '`retry clparent`', this command will report an error unless we have stack trace information about all of the current call's ancestors.

The command will also report an error unless the values of all the input arguments of the selected call are available at the return site at which control would reenter the selected call. (The compiler will keep the values of the input arguments of traced predicates as long as possible, but it cannot keep them beyond the point where they are destructively updated.) The exception is values of type 'io.state'; the debugger can perform a retry if the only missing value is of type 'io.state' (there can be only one io.state at any given time).

Retries over I/O actions are guaranteed to be safe only if the events at which the retry starts and ends are both within the I/O tabled region of the program's execution. If the retry is not guaranteed to be safe, the debugger will normally ask the user if they really want to do this. The option '`-f`' or '`--force`' suppresses the question, telling the debugger that retrying over I/O is OK; the option '`-o`' or '`--only-if-safe`' suppresses the question, telling the debugger that retrying over I/O is not OK; the option '`-i`' or '`--interactive`' restores the question if a previous option suppressed it.

`track` *num* [*termpath*]
Goto the EXIT event of the procedure in which the subterm in argument *num* at term path *termpath* was bound, and display information about where the term was bound.

Note that this command just invokes a script that is equivalent to running the following sequence of commands:

```
dd
browse num
cd termpath
track
info
pd
```

7.10.4 Browsing commands

`vars` Prints the names of all the known variables in the current environment, together with an ordinal number for each variable.

`held_vars`
Prints the names of all the held variables.

`print [-fpv]` *`name`*`[`*`termpath`*`]`
`print [-fpv]` *`num`*`[`*`termpath`*`]`
Prints the value of the variable in the current environment with the given name, or with the given ordinal number. If the name or number is followed by a term path such as `"^2"`, then only the specified subterm of the given variable is printed. This is a non-interactive version of the '`browse`' command (see below). Various settings which affect the way that terms are printed out (including e.g. the maximum term depth) can be set using the '`format_param`' command.

The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for printing.

`print [-fpv] *`
Prints the values of all the known variables in the current environment.

The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for printing.

`print [-fpv]`
`print [-fpv] goal`
Prints the goal of the current call in its present state of instantiation.

The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for printing.

`print [-fpv] exception`
Prints the value of the exception at an EXCP port. Reports an error if the current event does not refer to such a port.

The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for printing.

`print io limits`
`print action limits`
Prints the numbers of the lowest and highest numbered I/O actions executed and recorded by the program, if any.

`print [-fpv] io` *num*
`print [-fpv] action` *num*
: Prints a representation of the *num*'th I/O action executed by the program, if there was such an action and if it was recorded.

  The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for printing.

`print [-fpv] io` *min*`-`*max*
`print [-fpv] action` *min*`-`*max*
: Prints a representation of the I/O actions executed by the program from the *min*'th to the *max*'th, if there were such actions and if they were recorded.

  The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for printing.

`print [-fpv] [-m` *max*`] io`
`print [-fpv] [-m` *max*`] action`
: If no I/O actions have been printed yet, or if the last mdb command to print I/O actions was not successful, prints a representation of the first *max* I/O actions executed and recorded by the program. If there was an mdb command to print I/O actions and it was successful, prints a representation of the next *max* I/O actions executed and recorded by the program.

  The value of *max* is given by the '`-m`' option. If the option is not specified, the default is 20.

  The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for printing.

`print [-fpv] [-m` *max*`] io *`
`print [-fpv] [-m` *max*`] action *`
: Prints a representation of the first *max* I/O actions executed and recorded by the program.

  The value of *max* is given by the '`-m`' option. If the option is not specified, the default is 500.

  The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for printing.

`browse [-fpvw]` *name*`[`*termpath*`]`
`browse [-fpvw]` *num*`[`*termpath*`]`
: Invokes an interactive term browser to browse the value of the variable in the current environment with the given ordinal number or with the given name. If

the name or number is followed by a term path such as `"^2"`, then only the specified subterm of the given variable is given to the browser.

The interactive term browser allows you to selectively examine particular subterms. The depth and size of printed terms may be controlled. The displayed terms may also be clipped to fit within a single screen.

The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for browsing. The '`-w`' or '`--web`' option tells mdb to dump the value of the variable to an HTML file and then invoke a web browser on that file.

For further documentation on the interactive term browser, invoke the '`browse`' command from within '`mdb`' and then type '`help`' at the '`browser>`' prompt.

`browse [-fpvw]`
`browse [-fpvw] goal`
Invokes the interactive term browser to browse the goal of the current call in its present state of instantiation.

The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for browsing. The '`-w`' or '`--web`' option tells mdb to dump the goal to an HTML file and then invoke a web browser on that file.

`browse [-fpvw] exception`
Invokes the interactive term browser to browse the value of the exception at an EXCP port. Reports an error if the current event does not refer to such a port.

The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for browsing. The '`-w`' or '`--web`' option tells mdb to dump the exception to an HTML file and then invoke a web browser on that file.

`browse [-fpvw] io` *num*
`browse [-fpvw] action` *num*
Invokes an interactive term browser to browse a representation of the *num*'th I/O action executed by the program, if there was such an action and if it was recorded.

The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`' specify the format to use for browsing. The '`-w`' or '`--web`' option tells mdb to dump the I/O action representation to an HTML file, and then invoke a web browser on that file.

`stack [-a] [-d] [-c`*cliquelines*`] [-f`*numframes*`] [`*numlines*`]`
: Prints the names of the ancestors of the call specified by the current event. If two or more consecutive ancestor calls are for the same procedure, the procedure identification will be printed once with the appropriate multiplicity annotation.

  The option '`-d`' or '`--detailed`' specifies that for each ancestor call, the call's event number, sequence number and depth should also be printed if the call is to a procedure that is being execution traced.

  The the '`-f`' option, if present, specifies that only the topmost *numframes* stack frames should be printed.

  The optional number *numlines*, if present, specifies that only the topmost *numlines* lines should be printed. The default value is 100; the special value 0 asks for all the lines to be printed.

  By default, this command will look for cliques of mutually recursive ancestors. It will identify them as such in the output, and it will print at most 10 lines from any clique. The '`-c`' option can be used to specify the maximum number of lines to print for a clique, with the special value 0 asking for all of them to be printed. The option '`-a`' asks for all lines to be printed *without* cliques being detected or marked.

  This command will report an error if there is no stack trace information available about any ancestor.

`up [-d] [`*num*`]`
: Sets the current environment to the stack frame of the *num*'th level ancestor of the current environment (the immediate caller is the first-level ancestor).

  If *num* is not specified, the default value is one.

  This command will report an error if the current environment doesn't have the required number of ancestors, or if there is no execution trace information about the requested ancestor, or if there is no stack trace information about any of the ancestors between the current environment and the requested ancestor.

  The option '`-d`' or '`--detailed`' specifies that for each ancestor call, the call's event number, sequence number and depth should also be printed if the call is to a procedure that is being execution traced.

`down [-d] [`*num*`]`
: Sets the current environment to the stack frame of the *num*'th level descendant of the current environment (the procedure called by the current environment is the first-level descendant).

If *num* is not specified, the default value is one.

This command will report an error if there is no execution trace information about the requested descendant.

The option '`-d`' or '`--detailed`' specifies that for each ancestor call, the call's event number, sequence number and depth should also be printed if the call is to a procedure that is being execution traced.

`level [-d]`
`level [-d] num`
`level [-d] ('clentry'|'clique')`
`level [-d] 'clparent'`
: If the command is given no arguments, it sets the current environment to the stack frame that belongs to the current event. If invoked with the number *num* as argument, it sets the current environment to the stack frame of the *num*'th level ancestor of the call to which the current event belongs. If invoked with the argument '`clentry`' or '`clique`', it sets the current environment to the stack frame of the call that first entered into the clique of recursive calls of which the current call is a part. (If the current call is not (mutually) recursive with any other currently active call, it sets the current environment to the stack frame of the current event.) If the command is given the argument '`clparent`', it sets the current environment to the stack frame of the first call outside the current call's clique. This will be the parent of the stack frame that '`level clentry`' would set the current environment to.

  This command will report an error if the current environment doesn't have the required number of ancestors, or if there is no execution trace information about the requested ancestor, or if there is no stack trace information about any of the ancestors between the current environment and the requested ancestor.

  The option '`-d`' or '`--detailed`' specifies that for each ancestor call, the call's event number, sequence number and depth should also be printed if the call is to a procedure that is being execution traced.

`current`
: Prints the current event. This is useful if the details of the event, which were printed when control arrived at the event, have since scrolled off the screen.

`view [-vf2] [-w window-cmd] [-s server-cmd] [-n server-name] [-t timeout]`
`view -c [-v] [-s server-cmd] [-n server-name]`
: Opens a new window displaying the source code, at the location of the current event. As mdb stops at new events, the window is updated to track through the source code. This requires X11 and a version of '`vim`' compiled with the client/server option enabled.

The debugger only updates one window at a time. If you try to open a new source window when there is already one open, this command aborts with an error message.

The variant with '`-c`' (or '`--close`') does not open a new window but instead attempts to close a currently open source window. The attempt may fail if, for example, the user has modified the source file without saving.

The option '`-v`' (or '`--verbose`') prints the underlying system calls before running them, and prints any output the calls produced. This is useful to find out what is wrong if the server does not start.

The option '`-f`' (or '`--force`') stops the command from aborting if there is already a window open. Instead it attempts to close that window first.

The option '`-2`' (or '`--split-screen`') starts the vim server with two windows, which allows both the callee as well as the caller to be displayed at interface events. The lower window shows what would normally be seen if the split-screen option was not used, which at interface events is the caller. At these events, the upper window shows the callee definition. At internal events, the lower window shows the associated source, and the view in the upper window (which is not interesting at these events) remains unchanged.

The option '`-w`' (or '`--window-command`') specifies the command to open a new window. The default is '`xterm -e`'.

The option '`-s`' (or '`--server-command`') specifies the command to start the server. The default is '`vim`'.

The option '`-n`' (or '`--server-name`') specifies the name of an existing server. Instead of starting up a new server, mdb will attempt to connect to the existing one.

The option '`-t`' (or '`--timeout`') specifies the maximum number of seconds to wait for the server to start.

`hold` *`name`*`[`*`termpath`*`] [`*`heldname`*`]`

Holds on to the variable *name* of the current event, or the part of the specified by *termpath*, even after execution leaves the current event. The held value will stay accessible via the name *$heldname*. If *heldname* is not specified, it defaults to *name*. There must not already be a held variable named *heldname*.

`diff [-s` *start*`] [-m` *max*`]` *name1*`[`*termpath1*`]` *name2*`[`*termpath2*`]`
: Prints a list of some of the term paths at which the (specified parts of) the specified terms differ. Normally this command prints the term paths of the first 20 differences.

  The option '`-s`' (or '`--start`'), if present, specifies how many of the initial differences to skip.

  The option '`-m`' (or '`--max`'), if present, specifies how many differences to print.

`dump [-qx] goal` *filename*
: Writes the goal of the current call in its present state of instantiation to the specified file, and outputs a message announcing this fact unless the option '`-q`' (or '`--quiet`') was given. The option '`-x`' (or '`--xml`') causes the output to be in XML.

`dump [-qx] exception` *filename*
: Writes the value of the exception at an EXCP port to the specified file, and outputs a message announcing this fact unless the option '`-q`' (or '`--quiet`') was given. Reports an error if the current event does not refer to such a port. The option '`-x`' (or '`--xml`') causes the output to be in XML.

`dump [-qx]` *name filename*
`dump [-qx]` *num filename*
: Writes the value of the variable in the current environment with the given ordinal number or with the given name to the specified file, and outputs a message announcing this fact unless the option '`-q`' (or '`--quiet`') was given. The option '`-x`' (or '`--xml`') causes the output to be in XML.

`open` *term*
: Save *term* to a temporary file and open the file in an editor. The environment variable `EDITOR` is consulted to determine what editor to use. If this environment variable is not set then '`vi`' is used. *term* may be any term that can be saved to a file with the '`save_to_file`' command.

`grep` *pattern term*
: Saves the given term to a temporary file and invokes grep on the file using *pattern*. *term* may be any term that can be saved to a file with the '`save_to_file`' command. The Unix '`grep`' command must be available from the shell for this command to work.

`list [`*num*`]`
: Lists the source code text for the current environment, including *num* preceding and following lines. If *num* is not provided then the default of two is used.

7.10.5 Breakpoint commands

`break [-PS] [-E`*`ignore-count`*`] [-I`*`ignore-count`*`] [-n] [-p`*`print-spec`*`]*`
`filename``:`*`linenumber`*

Puts a break point on the specified line of the specified source file, if there is an event or a call at that position. If the filename is omitted, it defaults to the filename from the context of the current event.

The options '`-P`' or '`--print`', and '`-S`' or '`--stop`' specify the action to be taken at the break point.

The options '`-E`*`ignore-count`*' and '`--ignore-entry` *`ignore-count`*' tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of a call event that matches the breakpoint. The options '`-I`*`ignore-count`*' and '`--ignore-interface` *`ignore-count`*' tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of interface events that match the breakpoint.

Each occurrence of the options '`-p`*`printspec`*' and '`--print-list` *`printspec`*' tells the debugger to include the specified entity in the breakpoint's print list.

Normally, if a variable with the given name or number doesn't exist when execution reaches the breakpoint, mdb will issue a warning. The option '`-n`' or '`--no-warn`', if present, suppresses this warning. This can be useful if e.g. the name is the name of an output variable, which of course won't be present at call events.

By default, the action of the break point is '`stop`', the ignore count is zero, and the print list is empty.

`break [-AOPSaei] [-E`*`ignore-count`*`] [-I`*`ignore-count`*`] [-n] [-p`*`print-spec`*`]*`
`proc-spec`

Puts a break point on the specified procedure.

The options '`-A`' or '`--select-all`', and '`-O`' or '`--select-one`' select the action to be taken if the specification matches more than one procedure. If you have specified option '`-A`' or '`--select-all`', mdb will put a breakpoint on all matched procedures, whereas if you have specified option '`-O`' or '`--select-one`', mdb will report an error. By default, mdb will ask you whether you want to put a breakpoint on all matched procedures or just one, and if so, which one.

The options '`-P`' or '`--print`', and '`-S`' or '`--stop`' specify the action to be taken at the break point.

The options '`-a`' or '`--all`', '`-e`' or '`--entry`', and '`-i`' or '`--interface`' specify the invocation conditions of the break point. If none of these options are specified, the default is the one indicated by the current scope (see the '`scope`' command below). The initial scope is '`interface`'.

The options '`-E`*ignore-count*' and '`--ignore-entry` *ignore-count*' tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of a call event that matches the breakpoint. The options '`-I`*ignore-count*' and '`--ignore-interface` *ignore-count*' tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of interface events that match the breakpoint.

Each occurrence of the options '`-p`*printspec*' and '`--print-list` *printspec*' tells the debugger to include the specified entity in the breakpoint's print list.

Normally, if a variable with the given name or number doesn't exist when execution reaches the breakpoint, mdb will issue a warning. The option '`-n`' or '`--no-warn`', if present, suppresses this warning. This can be useful if e.g. the name is the name of an output variable, which of course won't be present at call events.

By default, the action of the break point is '`stop`', its invocation condition is '`interface`', the ignore count is zero, and the print list is empty.

`break [-OPS] [-E`*ignore-count*`] [-I`*ignore-count*`] [-n] [-p`*print-spec*`]*`
proc-spec portname

Puts a break point on one or more events of the specified type in the specified procedure. Port names should be specified as they are printed at events, e.g. '`CALL`', '`EXIT`', '`DISJ`', '`SWTC`', etc.

The option '`-O`' or '`--select-one`' selects the action to be taken if the specification matches more than one procedure. If you have specified option '`-O`' or '`--select-one`', mdb will report an error; otherwise, mdb will ask you which of the matched procedures you want to select.

If there is only one event of the given type in the specified procedure, mdb will put the breakpoint on it; otherwise, it will ask you whether you want to put a breakpoint on all matched events or just one, and if so, which one.

The options '`-P`' or '`--print`', and '`-S`' or '`--stop`' specify the action to be taken at the break point.

The options '`-E`*ignore-count*' and '`--ignore-entry` *ignore-count*' tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of a call event that matches the breakpoint. The options '`-I`*ignore-count*' and

‘`--ignore-interface` *`ignore-count`*’ tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of interface events that match the breakpoint.

Each occurrence of the options ‘`-p`*`printspec`*’ and ‘`--print-list` *`printspec`*’ tells the debugger to include the specified entity in the breakpoint’s print list.

Normally, if a variable with the given name or number doesn’t exist when execution reaches the breakpoint, mdb will issue a warning. The option ‘`-n`’ or ‘`--no-warn`’, if present, suppresses this warning. This can be useful if e.g. the name is the name of an output variable, which of course won’t be present at call events.

By default, the action of the break point is ‘`stop`’, the ignore count is zero, and the print list is empty.

`break [-PS] [-E`*`ignore-count`*`] [-I`*`ignore-count`*`] [-n] [-p`*`print-spec`*`]* here`
Puts a break point on the procedure referred to by the current event, with the invocation condition being the event at the current location in the procedure body.

The options ‘`-P`’ or ‘`--print`’, and ‘`-S`’ or ‘`--stop`’ specify the action to be taken at the break point.

The options ‘`-E`*`ignore-count`*’ and ‘`--ignore-entry` *`ignore-count`*’ tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of a call event that matches the breakpoint. The options ‘`-I`*`ignore-count`*’ and ‘`--ignore-interface` *`ignore-count`*’ tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of interface events that match the breakpoint.

Each occurrence of the options ‘`-p`*`printspec`*’ and ‘`--print-list` *`printspec`*’ tells the debugger to include the specified entity in the breakpoint’s print list.

Normally, if a variable with the given name or number doesn’t exist when execution reaches the breakpoint, mdb will issue a warning. The option ‘`-n`’ or ‘`--no-warn`’, if present, suppresses this warning. This can be useful if e.g. the name is the name of an output variable, which of course won’t be present at call events.

By default, the action of the break point is ‘`stop`’, the ignore count is zero, and the print list is empty.

`break [-PS] [-X`*`ignore-count`*`] [-n] [-p`*`print-spec`*`]* user_event [`*`user-event-set`*`] `*`user-event-name`*

Puts a break point on all user events named *user-event-name*, or, if *user-event-set* is specified as well, on the user event named *user-event-name* in that event set.

The options '`-P`' or '`--print`', and '`-S`' or '`--stop`' specify the action to be taken at the break point.

The options '`-X`*`ignore-count`*' and '`--ignore` *`ignore-count`*' tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of an event that matches the breakpoint.

Each occurrence of the options '`-p`*`printspec`*' and '`--print-list` *`printspec`*' tells the debugger to include the specified entity in the breakpoint's print list.

Normally, if a variable with the given name or number doesn't exist when execution reaches the breakpoint, mdb will issue a warning. The option '`-n`' or '`--no-warn`', if present, suppresses this warning. This can be useful if e.g. the name is the name of an output variable, which of course won't be present at call events.

By default, the action of the break point is '`stop`', the ignore count is zero, and the print list is empty.

`break [-PS] [-X`*`ignore-count`*`] [-n] [-p`*`print-spec`*`]* user_event_set [`*`user-event-set`*`]`

Puts a break point either on all user events in all event sets, or, if *user-event-set* is specified, on all user events in the event set of the given name.

The options '`-P`' or '`--print`', and '`-S`' or '`--stop`' specify the action to be taken at the break point.

The options '`-X`*`ignore-count`*' and '`--ignore` *`ignore-count`*' tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of an event that matches the breakpoint.

Each occurrence of the options '`-p`*`printspec`*' and '`--print-list` *`printspec`*' tells the debugger to include the specified entity in the breakpoint's print list.

Normally, if a variable with the given name or number doesn't exist when execution reaches the breakpoint, mdb will issue a warning. The option '`-n`' or '`--no-warn`', if present, suppresses this warning. This can be useful if e.g. the name is the name of an output variable, which of course won't be present at call events.

By default, the action of the break point is '`stop`', the ignore count is zero, and the print list is empty.

`break info`
Lists the details, status and print lists of all break points.

`condition [-b`*`break-num`*`] [-p] [-v]` *`varname`*`[`*`pathspec`*`]` *`op term`*
Attaches a condition to the most recent breakpoint, or, if the '`-b`' or '`--break-num`' is given, to the breakpoint whose number is given as the argument. Execution won't stop at the breakpoint if the condition is false.

The condition is a match between a variable live at the breakpoint, or a part thereof, and *term*. It is ok for *term* to contain spaces. The term from the program to be matched is specified by *varname*; if it is followed by *pathspec* (without a space), it specifies that the match is to be against the specified part of *varname*.

There are two kinds of values allowed for *op*. If *op* is '`=`' or '`==`', the condition is true if the term specified by *varname* (and *pathspec*, if present) matches *term*. If *op* is '`!=`' or '`\=`', the condition is true if the term specified by *varname* (and *pathspec*, if present) doesn't match *term*. *term* may contain integers and strings (as long as the strings don't contain double quotes), but floats and characters are not supported (yet), and neither is any special syntax for operators. Operators can be specified in prefix form by quoting them with escaped single quotes, as in '`\'+\'(1, 2)`'. Lists can be specified using the usual syntax. *term* also may not contain variables, with one exception: any occurrence of '`_`' in *term* matches any term.

If execution reaches a breakpoint and the condition cannot be evaluated, execution will normally stop at that breakpoint with a message to that effect. If the '`-p`' or '`--dont-require-path`' option is given, execution won't stop at breakpoints at which the specified part of the specified variable doesn't exist. If the '`-v`' or '`--dont-require-var`' option is given, execution won't stop at breakpoints at which the specified variable itself doesn't exist. The '`-v`' or '`--dont-require-var`' option is implicitly assumed if the specified breakpoint is on all user events.

`ignore [-E`*`ignore-count`*`] [-I`*`ignore-count`*`]` *`num`*
The options '`-E`*`ignore-count`*' and '`--ignore-entry` *`ignore-count`*' tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of a call event that matches the breakpoint with the specified number. The options '`-I`*`ignore-count`*' and '`--ignore-interface` *`ignore-count`*' tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of interface events that match the breakpoint with the specified number. If neither option is given, the default is to ignore one call event that matches the breakpoint with the

specified number. Reports an error if there is no break point with the specified number.

`ignore [-E`*ignore-count*`] [-I`*ignore-count*`]`
: The options '`-E`*ignore-count*' and '`--ignore-entry` *ignore-count*' tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of a call event that matches the most recently added breakpoint. The options '`-I`*ignore-count*' and '`--ignore-interface` *ignore-count*' tell the debugger to ignore the breakpoint until after *ignore-count* occurrences of interface events that match the most recently added breakpoint. If neither option is given, the default is to ignore one call event that matches the most recently added breakpoint. Reports an error if the most recently added breakpoint has since been deleted.

`break_print [-fpv] [-e] [-n] [-b` *num*`]` *print-spec*`*`
: Adds the specified print list elements (there may be more than one) to the print list of the breakpoint numbered *num* (if the '`-b`' or '`--break-num`' option is given), or to the print list of the most recent breakpoint (if it is not given).

  Normally, if a variable with the given name or number doesn't exist when execution reaches the breakpoint, mdb will issue a warning. The option '`-n`' or '`--no-warn`', if present, suppresses this warning. This can be useful if e.g. the name is the name of an output variable, which of course won't be present at call events.

  Normally, the specified elements will be added at the start of the breakpoint's print list. The option '`-e`' or '`--end`', if present, causes them to be added at the end.

  By default, the specified elements will be printed with format `"flat"`. The options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`', if given, explicitly specify the format to use.

`break_print [-b` *num*`] none`
: Clears the print list of the breakpoint numbered *num* (if the '`-b`' or '`--break-num`' option is given), or the print list of the most recent breakpoint (if it is not given).

`disable` *num*
: Disables the break point with the given number. Reports an error if there is no break point with that number.

`disable *` Disables all break points.

`disable` Disables the most recently added breakpoint. Reports an error if the most recently added breakpoint has since been deleted.

`enable` *num*
Enables the break point with the given number. Reports an error if there is no break point with that number.

`enable *` Enables all break points.

`enable` Enables the most recently added breakpoint. Reports an error if the most recently added breakpoint has since been deleted.

`delete` *num*
Deletes the break point with the given number. Reports an error if there is no break point with that number.

`delete *` Deletes all break points.

`delete` Deletes the most recently added breakpoint. Reports an error if the most recently added breakpoint has already been deleted.

`modules` Lists all the debuggable modules (i.e. modules that have debugging information).

`procedures` *module*
Lists all the procedures in the debuggable module *module*.

`register [-q]`
Registers all debuggable modules with the debugger. Has no effect if this registration has already been done. The debugger will perform this registration when creating breakpoints and when listing debuggable modules and/or procedures. The command will print a message to this effect unless the '`-q`' or '`--quiet`' option is given.

7.10.6 I/O tabling commands

`table_io` Reports which phase of I/O tabling we are in at the moment.

`table_io start`
Tells the debugger to start tabling I/O actions.

`table_io stop`
Tells the debugger to stop tabling I/O actions.

`table_io stats`
: Reports statistics about I/O tabling.

7.10.7 Parameter commands

`mmc_options` *option1 option2 ...*
: This command sets the options that will be passed to '`mmc`' to compile your query when you use one of the query commands: '`query`', '`cc_query`', or '`io_query`'. For example, if a query results in a compile error, it may sometimes be helpful to use '`mmc_options --verbose-error-messages`'.

`printlevel none`
: Sets the default print level to '`none`'.

`printlevel some`
: Sets the default print level to '`some`'.

`printlevel all`
: Sets the default print level to '`all`'.

`printlevel`
: Reports the current default print level.

`scroll on` Turns on user control over the scrolling of sequences of event reports. This means that every screenful of event reports will be followed by a '`--more--`' prompt. You may type an empty line, which allows the debugger to continue to print the next screenful of event reports. By typing a line that starts with '`a`', '`s`' or '`n`', you can override the print level of the current command, setting it to '`all`', '`some`' or '`none`' respectively. By typing a line that starts with '`q`', you can abort the current debugger command and get back control at the next event.

`scroll off`
: Turns off user control over the scrolling of sequences of event reports.

`scroll` *size*
: Sets the scroll window size to *size*, which tells scroll control to stop and print a '`--more--`' prompt after every *size* − 1 events. The default value of *size* is the value of the `LINES` environment variable, which should correspond to the number of lines available on the terminal.

`scroll` Reports whether user scroll control is enabled and what the window size is.

`stack_default_limit` *size*
: Set the default number of lines printed by the '`stack`' and '`nondet_stack`' commands to *size*. If *size* is zero, the limit is disabled.

`goal_paths on`
: Turns on printing of goal paths at events.

`goal_paths off`
: Turns off printing of goal paths at events.

`goal_paths`
: Reports whether goal paths are printed at events.

`scope all` Sets the default scope of new breakpoints to "all", i.e. by default, new breakpoints on procedures will stop at all events in the procedure.

`scope interface`
: Sets the default scope of new breakpoints to "interface", i.e. by default, new breakpoints on procedures will stop at all interface events in the procedure.

`scope entry`
: Sets the default scope of new breakpoints to "entry", i.e. by default, new breakpoints on procedures will stop only at events representing calls to the procedure.

`scope` Reports the current default scope of new breakpoints.

`echo on` Turns on the echoing of commands.

`echo off` Turns off the echoing of commands.

`echo` Reports whether commands are being echoed or not.

`context none`
: When reporting events or ancestor levels, does not print contexts (filename/line number pairs).

`context before`
: When reporting events or ancestor levels, prints contexts (filename/line number pairs) before the identification of the event or call to which they refer, on the same line. With long fully qualified predicate and function names, this may make the line wrap around.

`context after`
: When reporting events or ancestor levels, prints contexts (filename/line number pairs) after the identification of the event or call to which they refer, on the

same line. With long fully qualified predicate and function names, this may make the line wrap around.

`context prevline`
: When reporting events or ancestor levels, prints contexts (filename/line number pairs) on a separate line before the identification of the event or call to which they refer.

`context nextline`
: When reporting events or ancestor levels, prints contexts (filename/line number pairs) on a separate line after the identification of the event or call to which they refer.

`context` Reports where contexts are being printed.

`user_event_context none`
: When reporting user-defined events, does not print either filename/line number pairs or procedure ids.

`user_event_context file`
: When reporting user-defined events, prints only filename/line number pairs, not procedure ids.

`user_event_context proc`
: When reporting user-defined events, prints only procedure ids, not filename/line number pairs.

`user_event_context full`
: When reporting user-defined events, prints both filename/line number pairs and procedure ids.

`user_event_context`
: Reports what parts of the context are being printed at user events.

`list_context_lines` *num*
: Sets the number of lines to be printed by the '`list`' command printed before and after the target context.

`list_context_lines`
: Prints the number of lines to be printed by the '`list`' command printed before and after the target context.

`list_path` *`dir1 dir2 ...`*
: The '`list`' command searches a list of directories when looking for a source code file. The '`list_path`' command sets the search path to the given list of directories.

`list_path`
: When invoked without arguments, the '`list_path`' command prints the search path consulted by the '`list`' command.

`push_list_dir` *`dir1 dir2 ...`*
: Pushes the given directories on to the search path consulted by the '`list`' command.

`pop_list_dir`
: Pops the leftmost (most recently pushed) directory from the search path consulted by the '`list`' command.

`list_cmd` *`ExternalCommand`*
: Tells mdb that all future '`list`' commands should be handled by *ExternalCommand*. The command will be called with four arguments: the source file name, the first line number (counting from 1), the last line number, the current line number. The command should print all the lines from the first to the last, both inclusive, with the current line marked (or highlighted) in some fashion to standard output, and report any errors to standard error.

  If *ExternalCommand* is '`none`' then the '`list`' command will revert to printing source listings internally.

`list_cmd`
: When invoked without arguments, the '`list_cmd`' command prints the last value set by the '`list_cmd`' command.

`fail_trace_counts` *`filename`*
: The declarative debugger can exploit information about the failing and passing test cases to ask better questions. This command tells the '`dice`' command that *filename* contains execution trace counts from failing test cases. The '`dice`' command will use this file unless this is overridden with its '`--fail-trace-counts`' option.

`fail_trace_counts`
: Prints the name of the file containing execution trace counts from failing test cases, if this has already been set.

`pass_trace_counts` *`filename`*
: The declarative debugger can exploit information about the failing and passing test cases to ask better questions. This command tells the '`dice`'

command that *filename* contains execution trace counts from passing test cases. The '`dice`' command will use this file unless this is overridden with its '`--pass-trace-counts`' option.

`pass_trace_counts`
Prints the name of the file containing execution trace counts from passing test cases, if this has already been set.

`max_io_actions` *num*
Set the maximum number of I/O actions to print in questions from the declarative debugger to *num*.

`max_io_actions`
Prints the maximum number of I/O actions to print in questions from the declarative debugger.

`web_browser_cmd` *command*
Set the shell command used to launch a web browser to *command*.

`web_browser_cmd`
Prints the shell command used to launch a web browser, if this has been set.

`format [-APB]` *format*
Sets the default format of the browser to *format*, which should be one of '`flat`', '`pretty`' or '`verbose`'.

The browser maintains separate configuration parameters for the three commands '`print *`', '`print` *var*', and '`browse` *var*'. A '`format`' command applies to all three, unless it specifies one or more of the options '`-A`' or '`--print-all`', '`-P`' or '`--print`', and '`-B`' or '`--browse`', in which case it will set only the selected command's default format.

`format_param [-APBfpv]` *param value*
Sets one of the parameters of the browser to the given value. The parameter *param* must be one of '`depth`', '`size`', '`width`' and '`lines`'.

- '`depth`' is the maximum depth to which subterms will be displayed. Subterms at the depth limit may be abbreviated as functor/arity, or (in lists) may be replaced by an ellipsis ('`...`'). The principal functor of any term has depth zero. For subterms which are not lists, the depth of any argument of the functor is one greater than the depth of the functor. For subterms which are lists, the depth of each element of the list is one greater than the depth of the list.

- '`size`' is the suggested maximum number of functors to display. Beyond

this limit, subterms may be abbreviated as functor/arity, or (in lists) may be replaced by an ellipsis ('`...`'). For the purposes of this parameter, the size of a list is one greater than the sum of the sizes of the elements in the list.

- '`width`' is the width of the screen in characters.
- '`lines`' is the preferred maximum number of lines of one term to display.

The browser maintains separate configuration parameters for the three commands '`print *`', '`print` *`var`*', and '`browse` *`var`*'. A '`format_param`' command applies to all three, unless it specifies one or more of the options '`-A`' or '`--print-all`', '`-P`' or '`--print`', and '`-B`' or '`--browse`', in which case it will set only the selected command's parameters.

The browser also maintains separate configuration parameters for the different output formats: flat, pretty and verbose. A '`format_param`' command applies to all of these, unless it specifies one or more of the options '`-f`' or '`--flat`', '`-p`' or '`--pretty`', and '`-v`' or '`--verbose`', in which case it will set only the selected format's parameter.

`alias` *`name command`* `[`*`command-parameter ...`*`]`
Introduces *name* as an alias for the given command with the given parameters. Whenever a command line has *name* as its first word, the debugger will substitute the given command and parameters for this word before executing the command line.

If *name* is the upper-case word '`EMPTY`', the debugger will substitute the given command and parameters whenever the user types in an empty command line.

If *name* is the upper-case word '`NUMBER`', the debugger will insert the given command and parameters before the command line whenever the user types in a command line that consists of a single number.

`unalias` *`name`*
Removes any existing alias for *name*.

7.10.8 Help commands

`document_category` *`slot category`*
Create a new category of help items, named *category*. The summary text for the category is given by the lines following this command, up to but not including a line containing only the lower-case word '`end`'. The list of category

summaries printed in response to the command '`help`' is ordered on the integer *slot* numbers of the categories involved.

`document` *`category slot item`*
: Create a new help item named *item* in the help category *category*. The text for the help item is given by the lines following this command, up to but not including a line containing only the lower-case word '`end`'. The list of items printed in response to the command '`help` *`category`*' is ordered on the integer *slot* numbers of the items involved.

`help` *`category item`*
: Prints help text about the item *item* in category *category*.

`help` *`word`*
: Prints help text about *word*, which may be the name of a help category or a help item.

`help`
: Prints summary information about all the available help categories.

7.10.9 Declarative debugging mdb commands

The following commands relate to the declarative debugger. See Section 7.11 [Declarative debugging], page 66 for details.

`dd [-r] [-R] [-n`*`nodes`*`] [-s`*`search-mode`*`] [-p`*`passfile`*`] [-f`*`failfile`*`]`
: Starts declarative debugging using the current event as the initial symptom.

  When searching for bugs the declarative debugger needs to keep portions of the execution trace in memory. If it requires a new portion of the trace then it needs to rerun the program. The '`-n`*`nodes`*' or '`--nodes` *`nodes`*' option tells the declarative debugger how much of the execution trace to gather when it reruns the program. A higher value for *nodes* requires more memory, but improves the performance of the declarative debugger for long running programs since it will not have to rerun the program as often.

  The '`-s`*`search-mode`*' or '`--search-mode` *`search-mode`*' option tells the declarative debugger which search mode to use. Valid search modes are '`top_down`' (or '`td`'), '`divide_and_query`' (or '`dq`') and '`suspicion_divide_and_query`' (or '`sdq`'). '`top_down`' is the default when this option is not given.

  Use the '`-r`' or '`--resume`' option to continue your previous declarative debugging session. If the '`--resume`' option is given and there were no previous declarative debugging sessions then the option will be ignored. A '`dd --resume`' command can be issued at any event. The '`--search-mode`' option may be used

with the '`--resume`' option to change the search mode of a previously started declarative debugging session.

Use the '`-R`' or '`--reset-knowledge-base`' option to reset the declarative debugger's knowledge base. The declarative debugger will forget any previous answers that have been supplied. It will ask previous questions again if it needs to. This option does not affect what predicates or modules are trusted.

The arguments supplied to the '`--pass-trace-counts`' (or '`-p`') and '`--fail-trace-counts`' (or '`-f`') options are either trace count files or files containing a list of trace count files. The supplied trace counts are used to assign a suspicion to each event based on which parts of program were executed in the failing test case(s), but not the passing test case(s). This is used to guide the declarative debugger when the suspicion-divide-and-query search mode is used. If the suspicion-divide-and-query search mode is specified then either both the '`-p`' and '`-f`' options must be given, or the '`fail_trace_counts`' and '`pass_trace_counts`' configuration parameters must be set (using the '`set`' command).

`trust` *`module-name`*`|`*`proc-spec`*
Tells the declarative debugger to trust the given module, predicate or function.

Individual predicates or functions can be trusted by just giving the predicate or function name. If there is more than one predicate or function with the given name then a list of alternatives will be shown.

The entire Mercury standard library is trusted by default and can be untrusted in the usual manner using the 'untrust' command. To restore trusted status to the Mercury standard library issue the command 'trust standard library' or just 'trust std lib'.

See also 'trusted' and 'untrust'.

`trusted` Lists all the trusted modules, predicates and functions. See also 'trust' and 'untrust'.

`untrust` *`num`*
Removes the object from the list of trusted objects. *num* should correspond with the number shown in the list produced by issuing a 'trusted' command. See also 'trust' and 'trusted'.

7.10.10 Miscellaneous commands

`source [-i]` *`filename`* `[`*`args`*`]`
Executes the commands in the file named *filename*. Optionally a list of at most nine arguments can be given. Occurrences of the strings `"$1"` to `"$9"` in the

sourced file will be replaced by the corresponding arguments given in the source command before the commands in the sourced file are executed.

Lines that start with a hash (#) character are ignored. Hash characters can be used to place comments in your mdb scripts.

The option '`-i`' or '`--ignore-errors`' tells '`mdb`' not to complain if the named file does not exist or is not readable.

`save` *`filename`*
Saves the persistent state of the debugger (aliases, print level, scroll controls, set of breakpoints, browser parameters, set of objects trusted by the declarative debugger, etc) to the specified file. The state is saved in the form of mdb commands, so that sourcing the file will recreate the saved state. Note that this command does not save transient state, such as the current event. There is also a small part of the persistent state (breakpoints established with a '`break here`' command) that cannot be saved.

`quit [-y]` Quits the debugger and aborts the execution of the program. If the option '`-y`' is not present, asks for confirmation first. Any answer starting with '`y`', or end-of-file, is considered confirmation.

End-of-file on the debugger's input is considered a quit command.

7.10.11 Experimental commands

`histogram_all` *`filename`*
Prints (to file *filename*) a histogram that counts all events at various depths since the start of the program. This histogram is available only in some experimental versions of the Mercury runtime system.

`histogram_exp` *`filename`*
Prints (to file *filename*) a histogram that counts all events at various depths since the start of the program or since the histogram was last cleared. This histogram is available only in some experimental versions of the Mercury runtime system.

`clear_histogram`
Clears the histogram printed by '`histogram_exp`', i.e. sets the counts for all depths to zero.

`dice [-p`*`filename`*`] [-f`*`ffilename`*`] [-n`*`num`*`] [-s[pPfFsS]+] [-o` *`filename`*`] [-m` *`module`*`]`
Display a program dice on the screen.

A dice is a comparison between some successful test runs of the program and a failing test run. Before using the '`dice`' command one or more passing execution summaries and one failing execution summary need to be generated. This can be done by compiling the program with deep tracing enabled (either by compiling in a .debug or .decldebug grade or with the '`--trace deep`' or '`--trace rep`' compiler options) and then running the program under '`mtc`'. This will generate a file with the prefix '`.mercury_trace_counts`' and a unique suffix, that contains a summary of the program's execution. This summary is called a slice. Copy the generated slice to a new file for each test case, to end up with a failing slice, say '`fail`', and some passing slices, say '`pass1`', '`pass2`', '`pass3`', etc. Union the passing slices with a command such as '`mtc_union -p passes pass1 pass2 pass3`'.

The '`dice`' command can use these files to display a table of statistics comparing the passing test runs to the failing run. Here is an example of a dice displayed in an mdb session:

```
mdb> dice -f fail -p passes -s S -n 4
Procedure        Path/Port  File:Line Pass (3) Fail Suspicion
pred s.mrg/3-0   <s2;c2;e;> s.m:74       0 (0)    1      1.00
pred s.mrg/3-0   <s2;c2;t;> s.m:67      10 (3)    4      0.29
pred s.mrg/3-0   CALL       s.m:64      18 (3)    7      0.28
pred s.mrg/3-0   EXIT       s.m:64      18 (3)    7      0.28
```

This example tells us that the '`else`' in '`s.m`' on line 74 was executed once in the failing test run, but never in the passing test runs, so this would be a good place to start looking for a bug.

Each row in the table contains statistics about the execution of a separate goal in the program. Six columns are displayed:

- '`Procedure`': The procedure in which the goal appears.
- '`Path/Port`': The goal path and/or port of the goal. For atomic goals, statistics about the CALL event and the corresponding EXIT, FAIL or EXCP event are displayed on separate rows. For other types of goals the goal path is displayed, except for NEGE, NEGS and NEGF events where the goal path and port are displayed.
- '`File:Line`': The file name and line number of the goal. This can be used to set a breakpoint on the goal.
- '`Pass (total passing test runs)`': The total number of times the goal was executed in all the passing test runs. This is followed by a number in parentheses which indicates the number of test runs the goal was executed in. The heading of this column also has a number in parentheses which is the total number of passing test cases. In the example above we can see that 3 passing tests were run.

- '`Fail`': The number of times the goal was executed in the failing test run.
- '`Suspicion`': A number between 0 and 1 which gives an indication of how likely a particular goal is to be buggy. This is calculated as Suspicion = F / (P + F) where F is the number of times the goal was executed in the failing test run and P is the number of times the goal was executed in passing test runs.

The name of the file containing the failing slice can be specified with the '`-f`' or '`--fail-trace-counts`' option or with a separate '`set fail_trace_counts` *`filename`*' command.

The name of the file containing the union of the passing slices can be given with the '`-p`' or '`--pass-trace-counts`' option. Alternatively a separate '`set pass_trace_counts` *`filename`*' command can be given. See Section 7.12 [Trace counts], page 75 for more information about trace counts.

The table can be sorted on the Pass, Fail or Suspicion columns, or a combination of these. This can be done with the '`-s`' or '`--sort`' option. The argument of this option is a string made up of any combination of the letters '`pPfFsS`'. The letters in the string indicate how the table should be sorted:

- '`p`': Pass ascending
- '`P`': Pass descending
- '`f`': Fail ascending
- '`F`': Fail descending
- '`s`': Suspicion ascending
- '`S`': Suspicion descending

For example, the string `"SF"` means sort the table by suspicion, descending, and if any two suspicions are the same, then by number of executions in the failing test case, descending.

The option '`-n`' or '`--top`' can be used to limit the number lines displayed. Only the top *num* lines, with respect to the ordering specified by the '`-s`' option, will be displayed. By default the table is limited to 50 lines.

If the '`-o`' or '`--output-to-file`' option is given then the output will be written to the specified file instead of being displayed on the screen. Note that the file will be overwritten without warning if it already exists.

The '`-m`' or '`--module`' option limits the output to the given module and its submodules, if any.

7.10.12 Developer commands

The following commands are intended for use by the developers of the Mercury implementation.

`var_details`
: Prints all the information the debugger has about all the variables at the current program point.

`flag`
: Prints the values of all the runtime low-level debugging flags.

`flag` *`flagname`*
: Prints the value of the specified runtime low-level debugging flag.

`flag` *`flagname`* `on`
: Sets the specified runtime low-level debugging flag to true.

`flag` *`flagname`* `off`
: Sets the specified runtime low-level debugging flag to false.

`subgoal` *`n`*
: In minimal model grades, prints the details of the specified subgoal. In other grades, it reports an error.

`consumer` *`n`*
: In minimal model grades, prints the details of the specified consumer. In other grades, it reports an error.

`gen_stack`
: In minimal model grades, prints the contents of the frames on the generator stack. In other grades, it reports an error.

`cut_stack`
: In minimal model grades, prints the contents of the frames on the cut stack. In other grades, it reports an error.

`pneg_stack`
: In minimal model grades, prints the contents of the frames on the possible negated context stack. In other grades, it reports an error.

`mm_stacks`
: In minimal model grades, prints the contents of the frames on the generator stack, the cut stack and the possible negated context stack. In other grades, it reports an error.

`nondet_stack [-d] [-f`*numframes*`] [`*numlines*`]`
: Prints the contents of the frames on the nondet stack. By default, it prints only the fixed slots in each nondet stack frame, but if the '`-d`' or '`--detailed`' option is given, it will also print the names and values of the live variables in them.

  The '`-f`' option, if present, specifies that only the topmost *numframes* stack frames should be printed.

  The optional number *numlines*, if present, specifies that only the topmost *numlines* lines should be printed.

`stack_regs`
: Prints the contents of the virtual machine registers that point to the det and nondet stacks.

`all_regs` Prints the contents of all the virtual machine registers.

`debug_vars`
: Prints the values of the variables used by the debugger to record event numbers, call sequence numbers and call depths.

`stats [-f `*filename*`] `*subject*
: Prints statistics about the given subject to standard output, unless the '`-f`' or '`--filename`' option is given, in which case it prints the statistic to *filename*.

  subject can be '`procs`', which asks for statistics about proc layout structures in the program.

  subject can be '`labels`', which asks for statistics about label layout structures in the program.

  subject can be '`var_names`', which asks for statistics about the space occupied by variable names in the layout structures in the program.

  subject can be '`io_tabling`', which asks for statistics about the number of times each predicate appears in the I/O action table.

`print_optionals`
: Reports whether optionally-printed values such as typeinfos that are usually of interest only to implementors are being printed or not.

`print_optionals on`
: Tells the debugger to print optionally-printed values.

`print_optionals off`
: Tells the debugger not to print optionally-printed values.

`unhide_events`
: Reports whether events that are normally hidden (that are usually of interest only to implementors) are being exposed or not.

`unhide_events on`
: Tells the debugger to expose events that are normally hidden.

`unhide_events off`
: Tells the debugger to hide events that are normally hidden.

`table` *`proc`* `[`*`num1 ...`*`]`
: Tells the debugger to print the call table of the named procedure, together with the saved answer (if any) for each call. Reports an error if the named procedure isn't tabled.

  For now, this command is supported only for procedures whose arguments are all either integers, floats or strings.

  If the user specifies one or more integers on the command line, the output is restricted to the entries in the call table in which the *n*th argument is equal to the *n*th number on the command line.

`type_ctor [-fr]` *`modulename typectorname arity`*
: Tests whether there is a type constructor defined in the given module, with the given name, and with the given arity. If there isn't, it prints a message to that effect. If there is, it echoes the identity of the type constructor.

  If the option '`-r`' or '`--print-rep`' option is given, it also prints the name of the type representation scheme used by the type constructor (known as its 'type_ctor_rep' in the implementation).

  If the option '`-f`' or '`--print-functors`' option is given, it also prints the names and arities of function symbols defined by the type constructor.

`all_type_ctors [-fr] [`*`modulename`*`]`
: If the user specifies a module name, lists all the type constructors defined in the given module. If the user doesn't specify a module name, lists all the type constructors defined in the whole program.

  If the option '`-r`' or '`--print-rep`' option is given, it also prints the name of the type representation scheme of each type constructor (known as its 'type_ctor_rep' in the implementation).

If the option '`-f`' or '`--print-functors`' option is given, it also prints the names and arities of function symbols defined by each type constructor.

`class_decl [-im]` *`modulename typeclassname arity`*
Tests whether there is a type class defined in the given module, with the given name, and with the given arity. If there isn't, it prints a message to that effect. If there is, it echoes the identity of the type class.

If the option '`-m`' or '`--print-methods`' option is given, it also lists all the methods of the type class.

If the option '`-i`' or '`--print-instance`' option is given, it also lists all the instances of the type class.

`all_class_decls [-im] [`*`modulename`*`]`
If the user specifies a module name, lists all the type classes defined in the given module. If the user doesn't specify a module name, lists all the type classes defined in the whole program.

If the option '`-m`' or '`--print-methods`' option is given, it also lists all the methods of each type class.

If the option '`-i`' or '`--print-instance`' option is given, it also lists all the instances of each type class.

`all_procedures [-su] [-m` *`modulename`*`]` *`filename`*
In the absence of the '`-m`' or '`--module`' option, puts a list of all the debuggable procedures in the program into the named file. In the presence of the '`-m`' or '`--module`' option, puts a list of all the debuggable procedures in the named module into the named file.

If the '`-s`' or '`--separate`' option is given, the various components of procedure names are separated by spaces.

If the '`-u`' or '`--uci`' option is given, the list will include the procedures of compiler generated unify, compare, index and initialization predicates. Normally, the list includes the procedures of only user defined predicates.

`ambiguity [-o` *`filename`*`] [-ptf] [`*`modulename ...`*`]`
Print ambiguous procedure, type constructor and/or function symbol names. A procedure name is ambiguous if a predicate or function is defined with that name in more than one module or with more than one arity. A type constructor name is ambiguous if a type constructor is defined with that name in more than one module or with more than one arity. A function symbol name is ambiguous

if a function symbol is defined with that name in more than one module or with more than one arity.

If any module names are given, then only those modules are consulted, (any ambiguities involving predicates, functions and type constructors in non-listed modules are ignored). The module names have to be fully qualified, if a module *child* is a submodule of module *parent*, the module name list must include *parent.child*; listing just *child* won't work, since that is not a fully qualified module name.

If the '`-o`' or '`--outputfile`' option is given, the output goes to the file named as the argument of the option; otherwise, it goes to standard output.

If one or more of the '`-p`', '`-t`', '`-f`' options or their long equivalents, '`--types`', or '`--functors`', this command prints ambiguities only for the indicated kinds of constructs. The default is to print all ambiguities.

`trail_details`
Prints out low-level details of the state of the trail. In other grades, it reports an error.

7.11 Declarative debugging

The debugger incorporates a declarative debugger which can be accessed from its command line. Starting from an event that exhibits a bug, e.g. an event giving a wrong answer, the declarative debugger can find a bug which explains that behaviour using knowledge of the intended interpretation of the program only.

Note that this is a work in progress, so there are some limitations in the implementation.

7.11.1 Overview

The declarative debugger tries to find a bug in your program by asking questions about the correctness of calls executed in your program.

Because pure Mercury code does not have any side effects, the declarative debugger can make inferences such as "if a call produces incorrect output from correct input, then there must be a bug in the code executed by one of the descendents of the call".

The declarative debugger is therefore able to automate much of the 'detective work' that must be done manually when using the procedural debugger.

7.11.2 Concepts

Every CALL event corresponds to an atomic goal, the one printed by the "print" command at that event. This atom has the actual arguments in the input argument positions and distinct free variables in the output argument positions (including the return value for functions). We refer to this as the *call atom* of the event.

The same view can be taken of EXIT events, although in this case the outputs as well as the inputs will be bound. We refer to this as the *exit atom* of the event. The exit atom is always an instance of the call atom for the corresponding CALL event.

Using these concepts, it is possible to interpret the events at which control leaves a procedure as assertions about the semantics of the program. These assertions may be true or false, depending on whether or not the program's actual semantics are consistent with its intended semantics.

EXIT The assertion corresponding to an EXIT event is that the exit atom is valid in the intended interpretation. In other words, the procedure generates correct outputs for the given inputs.

FAIL Every FAIL event has a matching CALL event, and a (possibly empty) set of matching EXIT events between the call and fail. The assertion corresponding to a FAIL event is that every instance of the call atom which is true in the intended interpretation is an instance of one of the exit atoms. In other words, the procedure generates the complete set of answers for the given inputs. (Note that this does not imply that all exit atoms represent correct answers; some exit atoms may in fact be wrong, but the truth of the assertion is not affected by this.)

EXCP Every EXCP event is associated with an exception term, and has a matching CALL event. The assertion corresponding to an EXCP event is that the call atom can abnormally terminate with the given exception. In other words, the thrown exception was expected for that call.

If one of these assertions is wrong, then we consider the event to represent incorrect behaviour of the program. If the user encounters an event for which the assertion is wrong, then they can request the declarative debugger to diagnose the incorrect behaviour by giving the '`dd`' command to the procedural debugger at that event.

7.11.3 Oracle questions

Once the '`dd`' command has been given, the declarative debugger asks the user a series of questions about the truth of various assertions in the intended interpretation. The first question in this series will be about the validity of the event for which the '`dd`' command was given. The answer to this question will nearly always be "no", since the user has just implied the assertion is false by giving the '`dd`' command. Later questions will be about other events in the execution of the program, not all of them necessarily of the same kind as the first.

The user is expected to act as an "oracle" and provide answers to these questions based on their knowledge of the intended interpretation. The debugger provides some help here: previous answers are remembered and used where possible, so questions are not repeated unnecessarily. Commands are available to provide answers, as well as to browse the arguments more closely or to change the order in which the questions are asked. See the next section for details of the commands that are available.

When seeking to determine the validity of the assertion corresponding to an EXIT event, the declarative debugger prints the exit atom followed by the question '`Valid?`' for the user to answer. The atom is printed using the same mechanism that the debugger uses to print values, which means some arguments may be abbreviated if they are too large.

When seeking to determine the validity of the assertion corresponding to a FAIL event, the declarative debugger prints the call atom, prefixed by '`Call`', followed by each of the exit atoms (indented, and on multiple lines if need be), and prints the question '`Complete?`' (or '`Unsatisfiable?`' if there are no solutions) for the user to answer. Note that the user is not required to provide any missing instance in the case that the answer is no. (A limitation of the current implementation is that it is difficult to browse a specific exit atom. This will hopefully be addressed in the near future.)

When seeking to determine the validity of the assertion corresponding to an EXCP event, the declarative debugger prints the call atom followed by the exception that was thrown, and prints the question '`Expected?`' for the user to answer.

In addition to asserting whether a call behaved correctly or not the user may also assert that a call should never have occurred in the first place, because its inputs violated some precondition of the call. For example if an unsorted list is passed to a predicate that is only designed to work with sorted lists. Such calls should be deemed *inadmissible* by the user. This tells the declarative debugger that either the call was given the wrong input by its caller or whatever generated the input is incorrect.

In some circumstances the declarative debugger provides a default answer to the question. If this is the case, the default answer will be shown in square brackets immediately after the question, and simply pressing RET is equivalent to giving that answer.

7.11.4 Commands

At the above mentioned prompts, the following commands may be given. Most commands can be abbreviated by their first letter.

It is also legal to press RET without specifying a command. If there is a default answer (see Section 7.11.3 [Oracle questions], page 67), pressing RET is equivalent to giving that answer. If there is no default answer, pressing RET is equivalent to the skip command.

`yes` Answer 'yes' to the current question.

`no` Answer 'no' to the current question.

`inadmissible`
Answer that the call is inadmissible.

`trust` Answer that the predicate or function the question is about does not contain any bugs. However predicates or functions called by this predicate/function may contain bugs. The debugger will not ask you further questions about the predicate or function in the current question.

`trust module`
Answer that the module the current question relates to does not contain any bugs. No more questions about any predicates or functions from this module will be asked.

`skip` Skip this question and ask a different one if possible.

`undo` Undo the most recent answer or mode change.

`mode [ top-down | divide-and-query | binary ]`
Change the current search mode. The search modes may be abbreviated to '`td`', '`dq`' and '`b`' respectively.

`browse [--web] [n]`
Start the interactive term browser and browse the *n*th argument before answering. If the argument number is omitted then browse the whole call as if it were a data term. While browsing a '`track`' command may be issued to find the point at which the current subterm was bound (see Section 7.11.7 [Improving the search], page 73). To return to the declarative debugger question issue a '`quit`' command from within the interactive term browser. For more information on the use of the interactive term browser see the '`browse`' command in Section 7.10.4 [Browsing commands], page 37 or type '`help`' from within the interactive query browser.

Giving the '`--web`' or '`-w`' option causes the term to be displayed in a web browser.

`browse io [--web] n`
Browse the *n*'th I/O action.

`print [n]`
Print the *n*'th argument of the current question. If no argument is given, then display the current question.

`print io n`
Print the *n*'th I/O action.

`print io n-m`
Print the *n*'th to *m*'th I/O actions (inclusive).

`print io limits`
Print the values for which '`print n`' makes sense.

`print io` Print some I/O actions, starting just after the last action printed (if there was one) or at the first available action (if there was not).

`format format`
Set the default format to *format*, which should be one of '`flat`', '`verbose`' or '`pretty`'.

`depth num`
Set the maximum depth to which terms are printed to *num*.

`depth io` *num*
: Set the maximum depth to which I/O actions are printed to *num*. I/O actions are printed using the browser's '`print *`' command so the '`depth io`' command updates the configuration parameters for the browser's '`print *`' command.

`size` *num*
: Set the maximum number of function symbols to be printed in terms to *num*.

`size io` *num*
: Set the maximum number of function symbols to be printed in I/O actions to *num*. I/O actions are printed using the browser's '`print *`' command so the '`size io`' command updates the configuration parameters for the browser's '`print *`' command.

`width` *num*
: Set the number of columns in which terms are to be printed to *num*.

`width io` *num*
: Set the number of columns in which I/O actions are to be printed to *num*. I/O actions are printed using the browser's '`print *`' command so the '`width io`' command updates the configuration parameters for the browser's '`print *`' command.

`lines` *num*
: Set the maximum number of lines in terms to be printed to *num*.

`lines io` *num*
: Set the maximum number of lines in I/O actions to be printed to *num*. I/O actions are printed using the browser's '`print *`' command so the '`lines io`' command updates the configuration parameters for the browser's '`print *`' command.

`actions` *num*
: Set the maximum number of I/O actions to be printed in questions to *num*.

`params`
: Print the current values of browser parameters.

`track [-a] [`*term-path*`]`
: The '`track`' command can only be given from within the interactive term browser and tells the declarative debugger to find the point at which the current subterm was bound. If no argument is given the current subterm is taken to be incorrect. If a *term-path* is given then the subterm at *term-path* relative to the current subterm will be considered incorrect. The declarative debugger will ask about the call that bound the given subterm next. To find out the location of the unification that bound the subterm, issue an '`info`' command when asked about the call that bound the subterm. The declarative debugger can

use one of two algorithms to find the point at which the subterm was bound. The first algorithm uses some heuristics to find the subterm more quickly than the second algorithm. It is possible, though unlikely, for the first algorithm to find the wrong call. The first algorithm is the default. To tell the declarative debugger to use the second, more accurate but slower algorithm, give the '`-a`' or '`--accurate`' option to the '`track`' command.

`mark [-a] [`*`term-path`*`]`
: The '`mark`' command has the same effect as the '`track`' command except that it also asserts that the atom is inadmissible or erroneous, depending on whether the subterm is input or output respectively.

`pd`
: Commence procedural debugging from the current point. This command is notionally the inverse of the '`dd`' command in the procedural debugger. The session can be resumed with a '`dd --resume`' command.

`quit`
: End the declarative debugging session and return to the event at which the '`dd`' command was given. The session can be resumed with a '`dd --resume`' command.

`info`
: List the filename and line number of the predicate the current question is about as well as the filename and line number where the predicate was called (if this information is available). Also print some information about the state of the bug search, such as the current search mode, how many events are yet to be eliminated and the reason for asking the current question.

`help [`*`command`*`]`
: Summarize the list of available commands or give help on a specific command.

7.11.5 Diagnoses

If the oracle keeps providing answers to the asked questions, then the declarative debugger will eventually locate a bug. A "bug", for our purposes, is an assertion about some call which is false, but for which the assertions about every child of that call are not false (i.e. they are either correct or inadmissible). There are four different classes of bugs that this debugger can diagnose, one associated with each kind of assertion.

Assertions about EXIT events lead to a kind of bug we call an "incorrect contour". This is a contour (an execution path through the body of a clause) which results in a wrong answer for that clause. When the debugger diagnoses a bug of this kind, it displays the exit atoms in the contour. The resulting incorrect exit atom is displayed last. The program event associated with this bug, which we call the "bug event", is the exit event at the end of the contour.

Assertions about FAIL events lead to a kind of bug we call a "partially uncovered atom". This is a call atom which has some instance which is valid, but which is not covered by any of the applicable clauses. When the debugger diagnoses a bug of this kind, it displays the call atom; it does not, however, provide an actual instance that satisfies the above condition. The bug event in this case is the fail event reached after all the solutions were exhausted.

Assertions about EXCP events lead to a kind of bug we call an "unhandled exception". This is a contour which throws an exception that needs to be handled but which is not handled. When the debugger diagnoses a bug of this kind, it displays the call atom followed by the exception which was not handled. The bug event in this case is the exception event for the call in question.

If the assertion made by an EXIT, FAIL or EXCP event is false and one or more of the children of the call that resulted in the incorrect EXIT, FAIL or EXCP event is inadmissible, while all the other calls are correct, then an "inadmissible call" bug has been found. This is a call that behaved incorrectly (by producing the incorrect output, failing or throwing an exception) because it passed unexpected input to one of its children. The guilty call is displayed as well as the inadmissible child.

After the diagnosis is displayed, the user is asked to confirm that the event located by the declarative debugger does in fact represent a bug. The user can answer '`yes`' or '`y`' to confirm the bug, '`no`' or '`n`' to reject the bug, or '`abort`' or '`a`' to abort the diagnosis.

If the user confirms the diagnosis, they are returned to the procedural debugger at the event which was found to be the bug event. This gives the user an opportunity, if they need it, to investigate (procedurally) the events in the neighbourhood of the bug.

If the user rejects the diagnosis, which implies that some of their earlier answers may have been mistakes, diagnosis is resumed from some earlier point determined by the debugger. The user may now be asked questions they have already answered, with the previous answer they gave being the default, or they may be asked entirely new questions.

If the user aborts the diagnosis, they are returned to the event at which the '`dd`' command was given.

7.11.6 Search modes

The declarative debugger can operate in one of several modes when searching for a bug. Different search modes will result in different sequences of questions being asked by the declarative debugger. The user can specify which mode to use by giving the '`--search-mode`' option to the '`dd`' command (see Section 7.10.9 [Declarative debugging mdb commands], page 57) or with the '`mode`' declarative debugger command (see Section 7.11.4 [Declarative debugging commands], page 68).

7.11.6.1 Top-down mode

Using this mode the declarative debugger will ask about the children of the last question the user answered '`no`' to. The child calls will be asked about in the order they were executed. This makes the search more predictable from the user's point of view as the questions will more or less follow the program execution. The drawback of top-down search is that it may require a lot of questions to be answered before a bug is found, especially with deeply recursive programs.

This search mode is used by default when no other mode is specified.

7.11.6.2 Divide and query mode

With this search mode the declarative debugger attempts to halve the size of the search space with each question. In many cases this will result in the bug being found after O(log(N)) questions where N is the number of events between the event where the '`dd`'

command was given and the corresponding '`CALL`' event. This makes the search feasible for long running programs where top-down search would require an unreasonably large number of questions to be answered. However, the questions may appear to come from unrelated parts of the program which can make them harder to answer.

7.11.6.3 Suspicion divide and query mode

In this search mode the declarative debugger assigns a suspicion level to each event based on which parts of the program were executed in failing test cases, but not in passing test cases. It then attempts to divide the search space into two areas of equal suspicion with each question. This tends to result in questions about parts of the program executed in a failing test case, but not in passing test cases.

7.11.6.4 Binary search mode

The user may ask the declarative debugger to do a binary search along the path in the call tree between the current question and the question that the user last answered '`no`' to. This is useful, for example, when a recursive predicate is producing incorrect output, but the base case is correct.

7.11.7 Improving the search

The number of questions asked by the declarative debugger before it pinpoints the location of a bug can be reduced by giving it extra information. The kind of extra information that can be given and how to convey this information are explained in this section.

7.11.7.1 Tracking suspicious subterms

An incorrect subterm can be tracked to the call that bound the subterm from within the interactive term browser (see Section 7.11.4 [Declarative debugging commands], page 68).

After issuing a '`track`' command, the next question asked by the declarative debugger will be about the call that bound the incorrect subterm, unless that call was eliminated as a possible bug because of an answer to a previous question or the call that bound the subterm was not traced.

For example consider the following fragment of a program that calculates payments for a loan:

```
:- type payment
        --->   payment(
                        date    :: date,
                        amount  :: float
                ).

:- type date ---> date(int, int, int).  % date(day, month, year).

:- pred get_payment(loan::in, int::in, payment::out) is det.

get_payment(Loan, PaymentNo, Payment) :-
    get_payment_amount(Loan, PaymentNo, Amount),
    get_payment_date(Loan, PaymentNo, Date),
    Payment = payment(Date, Amount).
```

Suppose that `get_payment` produces an incorrect result and the declarative debugger asks:

```
get_payment(loan(...), 10, payment(date(9, 10, 1977), 10.000000000000)).
Valid?
```

Then if we know that this is the right payment amount for the given loan, but the date is incorrect, we can track the `date(...)` subterm and the debugger will then ask us about `get_payment_date`:

```
get_payment(loan(...), 10, payment(date(9, 10, 1977), 10.000000000000)).
Valid? browse
browser> cd 3/1
browser> ls
date(9, 10, 1977)
browser> track
get_payment_date(loan(...), 10, date(9, 10, 1977)).
Valid?
```

Thus irrelevant questions about `get_payment_amount` are avoided.

If, say, the date was only wrong in the year part, then we could also have tracked the year subterm in which case the next question would have been about the call that constructed the year part of the date.

This feature is also useful when using the procedural debugger. For example, suppose that you come across a '`CALL`' event and you would like to know the source of a particular input to the call. To find out you could first go to the final event by issuing a '`finish`' command. Invoke the declarative debugger with a '`dd`' command and then track the input term you are interested in. The next question will be about the call that bound the term. Issue a '`pd`' command at this point to return to the procedural debugger. It will now show the final event of the call that bound the term.

Note that this feature is only available if the executable is compiled in a .decldebug grade or with the '`--trace rep`' option. If a module is compiled with the '`--trace rep`' option but other modules in the program are not then you will not be able to track subterms through those other modules.

7.11.7.2 Trusting predicates, functions and modules

The declarative debugger can also be told to assume that certain predicates, functions or entire modules do not contain any bugs. The declarative debugger will never ask questions about trusted predicates or functions. It is a good idea to trust standard library modules imported by a program being debugged.

The declarative debugger can be told which predicates/functions it can trust before the '`dd`' command is given. This is done using the '`trust`', '`trusted`' and '`untrust`' commands at the mdb prompt (see Section 7.10.9 [Declarative debugging mdb commands], page 57 for details on how to use these commands).

Trust commands may be placed in the '`.mdbrc`' file which contains default settings for mdb (see Section 7.6 [Mercury debugger invocation], page 24). Trusted predicates will also be exported with a '`save`' command (see Section 7.10.10 [Miscellaneous commands], page 58).

During the declarative debugging session the user may tell the declarative debugger to trust the predicate or function in the current question. Alternatively the user may tell the declarative debugger to trust all the predicates and functions in the same module as the predicate or function in the current question. See the '`trust`' command in Section 7.11.4 [Declarative debugging commands], page 68.

7.11.7.3 When different search modes are used

If a search mode is given when invoking the declarative debugger then that search mode will be used, unless (a) a subterm is tracked during the session, or (b) the user has not answered '`no`' to any questions yet, in which case top-down search is used until '`no`' is answered to at least one question.

If no search mode is specified with the '`dd`' command, then the search mode depends on whether the '`--resume`' option is given. If it is, then the previous search mode will be used, otherwise top-down search will be used.

You can check the search mode used to find a particular question by issuing an '`info`' command at the question prompt in the declarative debugger. You can also change the search mode from within the declarative debugger with the '`mode`' command.

7.12 Trace counts

A program with debugging enabled may be run in a special mode that causes it to write out to a *trace count file* a record of how many times each *debugger event* in the program was executed during that run.

Trace counts are useful for determining what parts of a failing program are being run and possibly causing the failure; this is called *slicing*. Slices from failing and passing runs can be compared to see which parts of the program are being executed during failing runs, but not during passing runs; this is called *dicing*.

7.12.1 Generating trace counts

To generate a slice for a program run, first compile the program with deep tracing enabled (either by using the '`--trace deep`' option or by compiling the program in a debugging grade). Then invoke the program with the '`mtc`' script, passing any required arguments after the program name.

For example:

```
mtc ./myprog arg1 arg2
```

The program will run as usual, except that when it terminates it will write the number of times each debugger event was executed to a trace count file.

'`mtc`' accepts an '`-o`' or '`--output-file`' option. The argument to this option is the filename to use for the generated trace count file. If this option is not given, then the trace count will be written to a file with the prefix '`.mercury_trace_counts`' and a unique suffix.

Ordinarily, the generated trace count file will list only the debugger events that were actually executed during this run. However, it will list all debugger events, even unexecuted ones, if '`mtc`' is given the '`-c`' or '`--coverage-test`' option.

'`mtc`' also supports two more options intended for coverage testing: '`-s`' or '`--summary-file`', and '`--summary-count`'. These each set an option in the `MERCURY_OPTIONS` environment variable, '`--trace-count-summary-file`' and '`--trace-count-summary-max`' respectively. For the documentation of these '`mtc`' options, see the documentation of the `MERCURY_OPTIONS` environment variable.

Trace count files can be manipulated with the '`mtc_union`' and '`mtc_diff`' tools, and they can be analysed by the '`mslice`' and '`mdice`' tools. They can also be used to help direct a declarative debugging search (see Section 7.11.6 [Search modes], page 72).

7.12.2 Combining trace counts

The '`mtc_union`' tool can be used to combine several trace count files into one trace count file. You need to use this when you have many trace count files you wish to analyse with '`mslice`' or '`mdice`'.

'`mtc_union`' is invoked by issuing a command of the form:

```
mtc_union [-v] -o output_file file1 file2 ...
```

'`file1`', '`file2`', etc. are the trace count files that should be combined. The new trace count file will be written to '`output_file`'. This file will preserve the count of the test cases that contributed to its contents, even if some of '`file1`', '`file2`', etc themselves were created by '`mtc_union`'. If the '`-v`' or '`--verbose`' option is specified then a progress message will be displayed as each file is read and its contents merged into the union.

The '`mtc_diff`' tool can be used to subtract one trace count file from another.

'`mtc_diff`' is invoked by issuing a command of the form:

```
mtc_diff -o output_file file1 file2
```

'`file1`' and '`file2`' must both be trace counts files. The output, written to '`output_file`', will contain the difference between the trace counts in '`file1`' and '`file2`' for every event that occurs in '`file1`'. Unlike '`mtc_union`', '`mtc_diff`' does not preserve the count of the test cases that contributed to its contents in any useful way.

7.12.3 Slicing

Once a slice has been generated it can be viewed in various ways using the mslice tool. The output of the mslice tool will look something like the following:

```
Procedure            Path/Port File:Line Count (1)
pred mrg.merge/3-0   CALL      mrg.m:60     14 (1)
pred mrg.merge/3-0   EXIT      mrg.m:60     14 (1)
```

```
pred mrg.msort_n/4-0 CALL      mrg.m:33     12 (1)
pred mrg.msort_n/4-0 EXIT      mrg.m:33     12 (1)
pred mrg.msort_n/4-0 <?;>      mrg.m:35     12 (1)
```

Each row corresponds to a label in the program. The meanings of the columns are as follows:

- '`Procedure`': This column displays the procedure that the label relates to.
- '`Path/Port`': For interface events this column displays the event port, while for internal events it displays the goal path. (See Section 7.3 [Tracing of Mercury programs], page 19 for an explanation of interface and internal events.)
- '`File:Line`': This column displays the context of the event.
- '`Count`': This column displays how many times the event was executed. The number in parentheses for each event row says in how many runs the event was executed. The number in parentheses in the heading row (after the word "Count") indicates how many runs were represented in the trace counts file analysed by the mslice tool.

The mslice tool is invoked using a command of the form:

```
mslice [-s sortspec] [-l N] [-m module] [-n N] [-p N] [-f N] file
```

where '`file`' is a trace count file, generated either directly by a program run or indirectly by the '`mtc_union`' or '`mtc_diff`' tools.

The '`-s`' or '`--sort`' option specifies how the output should be sorted. '`sortspec`' should be a string made up of any combination of the letters '`cCtT`'. Each letter specifies a column and direction to sort on:

- '`c`': Count ascending
- '`C`': Count descending
- '`t`': Number of runs ascending
- '`T`': Number of runs descending

For example, the option '`-s cT`' will sort the output table by the Count column in ascending order. If the counts for two or more events are the same, then those events will be sorted by number of runs in descending order.

The default is to sort descending on the Count column.

The '`-l`' or '`--limit`' option limits the output to '`N`' lines.

The '`-m`' or '`--module`' option limits the output to events only from the given module.

The '`-n`' or '`--max-name-column`' option's argument gives the maximum width of the column containing predicate names. If the argument is zero, there is no maximum width.

The '`-p`' or '`--max-path-column`' option's argument gives the maximum width of the column containing ports and goal paths. If the argument is zero, there is no maximum width.

The '`-f`' or '`--max-file-column`' option's argument gives the maximum width of the column containing file names and line numbers. If the argument is zero, there is no maximum width.

7.12.4 Dicing

A dice is a comparison between passing and failing runs of a program.

Dice are created using the '`mdice`' tool. To use the '`mdice`' tool, one must first generate a set of trace count files for passing runs and a set of trace count files for failing runs using the '`mtc`' tool (Section 7.12.1 [Generating trace counts], page 75). Once this has been done, and the union of each set computed using '`mtc_union`', '`mdice`' can be used to display a table of statistics that compares the passing runs to the failing runs.

Here is an example of the output of the '`mdice`' tool:

```
Procedure        Path/Port  File:Line Pass (3) Fail Suspicion
pred s.mrg/3-0   <s2;c2;e;> s.m:74       0 (0)    1      1.00
pred s.mrg/3-0   <s2;c2;t;> s.m:67      10 (3)    4      0.29
pred s.mrg/3-0   CALL       s.m:64      18 (3)    7      0.28
pred s.mrg/3-0   EXIT       s.m:64      18 (3)    7      0.28
```

This example tells us that the '`else`' in '`s.m`' on line 74 was executed once in the failing test run, but never during the passing test runs, so this would be a good place to start looking for a bug.

Each row corresponds to an event in the program. The meanings of the columns are as follows:

- '`Procedure`': This column displays the procedure the event relates to.
- '`Path/Port`': For interface events this column displays the event port, while for internal events it displays the goal path. (See Section 7.3 [Tracing of Mercury programs], page 19 for an explanation of interface and internal events.)
- '`File:Line`': This column displays the context of the event.
- '`Pass (total passing test runs)`': This column displays the total number of times the event was executed in all the passing test runs. This is followed by a number in parentheses which indicates the number of test runs the event was executed in. The heading of this column also has a number in parentheses which is the total number of passing test cases.
- '`Fail`': This column displays the number of times the goal was executed in the failing test run(s).

- '`Suspicion`': This column displays a number between 0 and 1 which gives an indication of how likely a particular goal is to contain a bug. The suspicion is calculated as Suspicion = F / (P + F) where F is the number of times the goal was executed in failing runs and P is the number of times the goal was executed in passing runs.

The '`mdice`' tool is invoked with a command of the form:

```
mdice [-s sortspec] [-l N] [-m module] [-n N] [-p N] [-f N] passfile failfile
```

'`passfile`' is a trace count file, generated either directly by a passing program run or as the union of the trace count files of passing program runs. '`failfile`' is a trace count file, generated either directly by a failing program run or as the union of the trace count files of failing program runs.

The table can be sorted on the Pass, Fail or Suspicion columns, or a combination of these. This can be done with the '`-s`' or '`--sort`' option. The argument of this option is a string made up of any combination of the letters '`pPfFsS`'. The letters in the string indicate how the table should be sorted:

- '`p`': Pass ascending
- '`P`': Pass descending
- '`f`': Fail ascending
- '`F`': Fail descending
- '`s`': Suspicion ascending
- '`S`': Suspicion descending

For example the string `"SF"` means sort the table by suspicion in descending order, and if any two suspicions are the same, then by number of executions in the failing run(s), also in descending order.

The default is to sort descending on the Suspicion column.

The option '`-l`' or '`--limit`' can be used to limit the number of lines displayed.

The '`-m`' or '`--module`' option limits the output to the given module and any submodules.

The '`-n`' or '`--max-name-column`' option's argument gives the maximum width of the column containing predicate names. If the argument is zero, there is no maximum width.

The '`-p`' or '`--max-path-column`' option's argument gives the maximum width of the column containing ports and goal paths. If the argument is zero, there is no maximum width.

The '`-f`' or '`--max-file-column`' option's argument gives the maximum width of the column containing file names and line numbers. If the argument is zero, there is no maximum width.

7.12.5 Coverage testing

Coverage testing is the process of finding out which parts of the code of a program are not executed during any test case, so that new test cases can be designed specifically to exercise those parts.

The first step in coverage testing a Mercury program is compiling that program with execution tracing enabled, either by using the '`--trace deep`' option or by compiling the program in a debugging grade. The second step is to execute that program on all its test cases with coverage testing enabled. This can be done either by running the program with '`mtc --coverage-test`', or by including one of the corresponding options ('`--coverage-test`' or '`--coverage-test-if-exec=`*`programname`*') in the value of the `MERCURY_OPTIONS` environment variable. These runs generate a set of trace counts files that can be given to the Mercury test coverage tool, the '`mcov`' program. As usual, trace count files are named with the prefix '`.mercury_trace_counts`' if the '`mtc --output-file`' option is not given.

The '`mcov`' tool is invoked with a command of the form:

```
mcov [-d] [-v] [-o output_file] tracecountfile1 ...
```

The arguments consist of one or more trace count files. The output will normally be a list of all the procedures in the program that were not executed in any of the runs that generated these trace count files. The output will go to standard output unless this is overridden by the '`-o`' or '`--output-file`' option.

If the '`-d`' or '`--detailed`' option is specified, then the output will list all the *events* in the program that were not executed in any of these runs. This option can thus show the unexecuted parts of the executed procedures.

If the '`-v`' or '`--verbose`' option is specified, then a progress message will be displayed as each file is read.

8 Profiling

8.1 Profiling introduction

To obtain the best trade-off between productivity and efficiency, programmers should not spend too much time optimizing their code until they know which parts of the code are really taking up most of the time. Only once the code has been profiled should the programmer consider making optimizations that would improve efficiency at the expense of readability or ease of maintenance. A good profiler is therefore a tool that should be part of every software engineer's toolkit.

Mercury programs can be analyzed using two distinct profilers. The Mercury profiler '`mprof`' is a conventional call-graph profiler (or graph profiler for short) in the style of '`gprof`'. The Mercury deep profiler '`mdprof`' is a new kind of profiler that associates a lot more context with each measurement. '`mprof`' can be used to profile either time or space, but not both at the same time; '`mdprof`' can profile both time and space at the same time.

The parallel execution of Mercury programs can be analyzed with a third profiler called '`threadscope`'. '`threadscope`' allows programmers to visualise CPU utilisation for work, garbage collection, and idle time. This enables programmers to see the effect of parallelization decisions such as task granularity. The '`threadscope`' tool is not included with the Melbourne Mercury Compiler, See Threadscope: Performance Tuning Parallel Haskell Programs.

8.2 Building profiled applications

To enable profiling, your program must be built with profiling enabled. The three different profilers require different support, and thus you must choose which one to enable when you build your program.

- To build your program with time profiling enabled for '`mprof`', pass the '`-p`' ('`--profiling`') option to '`mmc`' (and also to '`mgnuc`' and '`ml`', if you invoke them separately).
- To build your program with memory profiling enabled for '`mprof`', pass the '`--memory-profiling`' option to '`mmc`', '`mgnuc`' and '`ml`'.
- To build your program with deep profiling enabled (for '`mdprof`'), pass the '`--deep-profiling`' option to '`mmc`', '`mgnuc`' and '`ml`'.
- To build your program with threadscope profiling enabled (for '`threadscope`'). pass the '`--parallel`' and '`--threadscope`' options to '`mmc`', '`mgnuc`' and '`ml`'.

If you are using Mmake, then you pass these options to all the relevant programs by setting the '`GRADEFLAGS`' variable in your Mmakefile, e.g. by adding the line '`GRADEFLAGS=--profiling`'. (For more information about the different grades, see Section 9.8 [Compilation model options], page 108.)

Enabling '`mprof`' or '`mdprof`' profiling has several effects. First, it causes the compiler to generate slightly modified code, which counts the number of times each predicate or function is called, and for every call, records the caller and callee. With deep profiling, there are other modifications as well, the most important impact of which is the loss of tail-recursion. (By default, the deep profiling versions of the library and runtime are built with '`--stack-segments`' in order to minimize this impact.) Second, your program will be linked with versions of the library and runtime that were compiled with the same kind of profiling enabled. Third, if you enable graph profiling, the compiler will generate for each source file the static call graph for that file in '*`module`*`.prof`'.

Enabling '`threadscope`' profiling causes the compiler to build the program against a different runtime system. This runtime system logs events relevant to parallel execution. '`threadscope`' support is not compatible with all processors, see '`README.ThreadScope`' for more information.

8.3 Creating profiles

Once you have created a profiled executable, you can gather profiling information by running the profiled executable on some test data that is representative of the intended uses of the program. The profiling version of your program will collect profiling information during execution, and save this information at the end of execution, provided execution terminates normally and not via an abort.

Executables compiled with '`--profiling`' save profiling data in the files '`Prof.Counts`', '`Prof.Decl`', and '`Prof.CallPair`'. ('`Prof.Decl`' contains the names of the procedures and their associated addresses, '`Prof.CallPair`' records the number of times each procedure was called by each different caller, and '`Prof.Counts`' records the number of times that execution was in each procedure when a profiling interrupt occurred.) Executables compiled with '`--memory-profiling`' will use two of those files ('`Prof.Decl`' and '`Prof.CallPair`') and two others: '`Prof.MemoryWords`' and '`Prof.MemoryCells`'. Executables compiled with '`--deep-profiling`' save profiling data in a single file, '`Deep.data`'. Executables compiled with the '`--threadscope`' option write profiling data to a file whose name is that of the program being profiled with the extension '`.eventlog`'. For example, the profile for the program '`my_program`' would be written to the file '`my_program.eventlog`'.

It is also possible to combine '`mprof`' profiling results from multiple runs of your program. You can do this by running your program several times, and typing '`mprof_merge_counts`' after each run. It is not (yet) possible to combine '`mdprof`' profiling results from multiple runs of your program.

Due to a known timing-related bug in our code, you may occasionally get segmentation violations when running your program with '`mprof`' profiling enabled. If this happens, just run it again — the problem occurs only very rarely. The same vulnerability does not occur with '`mdprof`' profiling.

With the '`mprof`' and '`mdprof`' profilers, you can control whether time profiling measures real (elapsed) time, user time plus system time, or user time only, by including the options '`-Tr`', '`-Tp`', or '`-Tv`' respectively in the environment variable `MERCURY_OPTIONS` when you run the program to be profiled. Currently, only the '`-Tr`' option works on Cygwin; on that platform it is the default.

The default is user time plus system time, which counts all time spent executing the process, including time spent by the operating system working on behalf of the process, but not including time that the process was suspended (e.g. due to time slicing, or while waiting for input). When measuring real time, profiling counts even periods during which the process was suspended. When measuring user time only, profiling does not count time inside the operating system at all.

8.4 Using mprof for time profiling

To display the graph profile information gathered from one or more profiling runs, just type '`mprof`' or '`mprof -c`'. (For programs built with '`--high-level-code`', you need to also pass the '`--no-demangle`' option to '`mprof`' as well.) Note that '`mprof`' can take quite a while to execute (especially with '`-c`'), and will usually produce quite a lot of output, so you will usually want to redirect the output into a file with a command such as '`mprof > mprof.out`'.

The output of '`mprof -c`' consists of three major sections. These are named the call graph profile, the flat profile and the alphabetic listing. The output of '`mprof`' contains the flat profile and the alphabetic listing only.

The call graph profile presents the local call graph of each procedure. For each procedure it shows the parents (callers) and children (callees) of that procedure, and shows the execution time and call counts for each parent and child. It is sorted on the total amount of time spent in the procedure and all of its descendents (i.e. all of the procedures that it calls, directly or indirectly.)

The flat profile presents the just execution time spent in each procedure. It does not count the time spent in descendents of a procedure.

The alphabetic listing just lists the procedures in alphabetical order, along with their index number in the call graph profile, so that you can quickly find the entry for a particular procedure in the call graph profile.

The profiler works by interrupting the program at frequent intervals, and each time recording the currently active procedure and its caller. It uses these counts to determine the proportion of the total time spent in each procedure. This means that the figures calculated for these times are only a statistical approximation to the real values, and so they should be treated with some caution. In particular, if the profiler's assumption that calls to a procedure from different callers have roughly similar costs is not true, the graph profile can be quite misleading.

The time spent in a procedure and its descendents is calculated by propagating the times up the call graph, assuming that each call to a procedure from a particular caller takes the same amount of time. This assumption is usually reasonable, but again the results should be treated with caution. (The deep profiler does not make such an assumption, and hence its output is significantly more reliable.)

Note that any time spent in a C function (e.g. time spent in '`GC_malloc()`', which does memory allocation and garbage collection) is credited to the Mercury procedure that called that C function.

Here is a small portion of the call graph profile from an example program.

```
                                  called/total       parents
index  %time    self descendents  called+self    name           index
                                  called/total       children

                                                     <spontaneous>
[1]    100.0    0.00        0.75       0         call_engine_label [1]
                0.00        0.75       1/1           do_interpreter [3]

-----------------------------------------------

                0.00        0.75       1/1           do_interpreter [3]
[2]    100.0    0.00        0.75       1         io.run/0(0) [2]
                0.00        0.00       1/1           io.init_state/2(0) [11]
                0.00        0.74       1/1           main/2(0) [4]

-----------------------------------------------
```

```
                0.00        0.75       1/1           call_engine_label [1]
[3]    100.0    0.00        0.75       1          do_interpreter [3]
                0.00        0.75       1/1           io.run/0(0) [2]

-----------------------------------------------

                0.00        0.74       1/1           io.run/0(0) [2]
[4]     99.9    0.00        0.74       1          main/2(0) [4]
                0.00        0.74       1/1           sort/2(0) [5]
                0.00        0.00       1/1           print_list/3(0) [16]
                0.00        0.00       1/10          io.write_string/3(0) [18]

-----------------------------------------------

                0.00        0.74       1/1           main/2(0) [4]
[5]     99.9    0.00        0.74       1          sort/2(0) [5]
                0.05        0.65       1/1           list.perm/2(0) [6]
                0.00        0.09   40320/40320       sorted/1(0) [10]

-----------------------------------------------

                                          8           list.perm/2(0) [6]
                0.05        0.65       1/1           sort/2(0) [5]
[6]     86.6    0.05        0.65       1+8        list.perm/2(0) [6]
                0.00        0.60    5914/5914        list.insert/3(2) [7]
                                          8           list.perm/2(0) [6]

-----------------------------------------------

                0.00        0.60    5914/5914        list.perm/2(0) [6]
[7]     80.0    0.00        0.60    5914          list.insert/3(2) [7]
                0.60        0.60    5914/5914        list.delete/3(3) [8]

-----------------------------------------------

                                       40319           list.delete/3(3) [8]
                0.60        0.60    5914/5914        list.insert/3(2) [7]
[8]     80.0    0.60        0.60    5914+40319    list.delete/3(3) [8]
                                       40319           list.delete/3(3) [8]

-----------------------------------------------

                0.00        0.00       3/69283       tree234.set/4(0) [15]
                0.09        0.09   69280/69283       sorted/1(0) [10]
[9]     13.3    0.10        0.10   69283          compare/3(0) [9]
                0.00        0.00       3/3           __Compare___io__stream/0(0) [20]
```

```
                0.00        0.00   69280/69280       builtin_compare_int/3(0) [27]

-----------------------------------------------

                0.00        0.09   40320/40320       sort/2(0) [5]
[10]    13.3    0.00        0.09   40320          sorted/1(0) [10]
                0.09        0.09   69280/69283       compare/3(0) [9]

-----------------------------------------------
```

The first entry is '`call_engine_label`' and its parent is '`<spontaneous>`', meaning that it is the root of the call graph. (The first three entries, '`call_engine_label`', '`do_interpreter`', and '`io.run/0`' are all part of the Mercury runtime; '`main/2`' is the entry point to the user's program.)

Each entry of the call graph profile consists of three sections, the parent procedures, the current procedure and the children procedures.

Reading across from the left, for the current procedure the fields are:

- The unique index number for the current procedure. (The index numbers are used only to make it easier to find a particular entry in the call graph.)
- The percentage of total execution time spent in the current procedure and all its descendents. As noted above, this is only a statistical approximation.
- The "self" time: the time spent executing code that is part of current procedure. As noted above, this is only a statistical approximation.
- The descendent time: the time spent in the current procedure and all its descendents. As noted above, this is only a statistical approximation.
- The number of times a procedure is called. If a procedure is (directly) recursive, this column will contain the number of calls from other procedures, a plus sign, and then the number of recursive calls. These numbers are exact, not approximate.
- The name of the procedure followed by its index number.

The predicate or function names are not just followed by their arity but also by their mode in brackets. A mode of zero corresponds to the first mode declaration of that predicate in the source code. For example, '`list.delete/3(3)`' corresponds to the '`(out, out, in)`' mode of '`list.delete/3`'.

Now for the parent and child procedures the self and descendent time have slightly different meanings. For the parent procedures the self and descendent time represent the proportion of the current procedure's self and descendent time due to that parent. These times are obtained using the assumption that each call contributes equally to the total time of the current procedure.

8.5 Using mprof for profiling memory allocation

To create a profile of memory allocations, you can invoke '`mprof`' with the '`-m`' ('`--profile memory-words`') option. This will profile the amount of memory allocated, measured in units of words. (A word is 4 bytes on a 32-bit architecture, and 8 bytes on a 64-bit architecture.)

Alternatively, you can use '`mprof`''s '`-M`' ('`--profile memory-cells`') option. This will profile memory in units of "cells". A cell is a group of words allocated together in a single

allocation, to hold a single object. Selecting this option this will therefore profile the number of memory allocations, while ignoring the size of each memory allocation.

With memory profiling, just as with time profiling, you can use the '`-c`' ('`--call-graph`') option to display call graph profiles in addition to flat profiles.

When invoked with the '`-m`' option, '`mprof`' only reports allocations, not deallocations (garbage collection). It can tell you how much memory was allocated by each procedure, but it won't tell you how long the memory was live for, or how much of that memory was garbage-collected. This is also true for '`mdprof`'.

The memory retention profiling tool described in the next section can tell you which memory cells remain on the heap.

8.6 Using mprof for profiling memory retention

When a program is built with memory profiling enabled and uses the Boehm garbage collector, i.e. a grade with '`.memprof.gc`' modifiers, each memory cell is "attributed" with information about its origin and type. This information can be collated to tell you what kinds of objects are being retained when the program executes.

To do this, you must instrument the program by adding calls to `benchmarking.report_memory_attribution/1` or `benchmarking.report_memory_attribution/3` at points of interest. The first argument of the `report_memory_attribution` predicates is a string that is used to label the memory retention data corresponding to that call in the profiling output. You may want to call them from within '`trace`' goals:

```
trace [run_time(env("SNAPSHOTS")), io(!IO)] (
    benchmarking.report_memory_attribution("Phase 2", !IO)
)
```

If a program operates in distinct phases you may want to add a call in between the phases. The '`report_memory_attribution`' predicates do nothing in other grades, so are safe to leave in the program.

Next, build the program in a '`.memprof.gc`' grade. After the program has finished executing, it will generate a file called '`Prof.Snapshots`' in the current directory. Run '`mprof -s`' to view the profile. You will see the memory cells which were on the heap at each time that '`report_memory_attribution`' was called: the origin of the cells, and their type constructors.

Passing the option '`-T`' will group the profile first by type constructors, then by procedure. The '`-b`' option produces a brief profile by hiding the secondary level of information. Memory cells allocated by the Mercury runtime system itself are normally excluded from the profile; they can be viewed by passing the '`-r`' option.

Note that Mercury values which are dead may in fact be still reachable from the various execution stacks. This is particularly noticeable on the high-level C back-end, as the C compiler does not take conservative garbage collection into account and Mercury values may linger on the C stack for longer than necessary. The low-level C grades should suffer to a lesser extent.

The attribution requires an extra word of memory per cell, which is then rounded up by the memory allocator. This is accounted for in '`mprof`' output, but the memory usage of the program may be significantly higher than in non-memory profiling grades.

8.7 Using mdprof

The user interface of the deep profiler is a browser. To display the information contained in a deep profiling data file (which will be called '`Deep.data`' unless you renamed it), start up your browser and give it a URL of the form '`http://server.domain.name/cgi-bin/mdprof_cgi?/full/path/name/Deep.data`'.
The '`server.domain.name`' part should be the name of a machine with the following qualifications: it should have a web server running on it, and it should have the '`mdprof_cgi`' program installed in the web server's CGI program directory. (On many Linux systems, this directory is '`/usr/lib/cgi-bin`'.) The '`/full/path/name/Deep.data`' part should be the full path name of the deep profiling data file whose data you wish to explore. The name of this file must not have percent signs in it, and it must end in the suffix '`.data`'.

When you start up '`mdprof`' using the command above, you will see a list of the usual places where you may want to start looking at the profile. Each place is represented by a link. Clicking on and following that link will give you a web page that contains both the profile information you asked for and other links, some of which present the same information in a different form and some of which lead to further information. You explore the profile by clicking on links and looking at the resulting pages.

The deep profiler can generate several kinds of pages.

The menu page
: The menu page gives summary information about the profile, and the usual starting points for exploration.

Clique pages
: Clique pages are the most fundamental pages of the deep profiler. Each clique page presents performance information about a clique, which is either a single procedure or a group of mutually recursive procedures, in a given ancestor context, which in turn is a list of other cliques starting with the caller of the entry point of the clique and ending with the clique of the '`main`' predicate.

  Each clique page lists the closest ancestor cliques, and then the procedures of the clique. It gives the cost of each call site in each procedure, as well as the cost of each procedure in total. These costs will be just those incurred in the given ancestor context; the costs incurred by these call sites and procedures in other ancestor contexts will be shown on other clique pages.

Procedure pages
: Procedure pages give the total cost of a procedure and its call sites in all ancestor contexts.

Module pages
: Module pages give the total cost of all the procedures of a module.

Module getters and setters pages
: These pages identifies the getter and setter procedures in a module. Getters and setters are simply predicates and functions that contain '`_get_`' and '`_set_`' respectively in their names; they are usually used to access fields of data structures.

Program modules page
: The program modules page gives the list of the program's modules.

Top procedure pages
: Top procedure pages identify the procedures that are most expensive as measured by various criteria.

Procedure caller pages
: A procedure caller page lists the call sites, procedures, modules or cliques that call the given procedure.

When exploring a procedure's callers, you often want only the ancestors that are at or above a certain level of abstraction. Effectively you want to draw a line through the procedures of the program, such that you are interested in the procedures on or above the line but those below the line. Since we want to exclude procedures below the line from procedure caller pages, we call this line an *exclusion contour*.

You can tell the deep profiler where you want to draw this line by giving it a '`exclusion contour file`'. The name of this file should be the same as the name of the deep profiling data file, but with the suffix '`.data`' replaced with '`.contour`'. This file should consist of a sequence of lines, and each line should contain two words. The first word should be either '`all`' or '`internal`'; the second should the name of a module. If the first word is '`all`', then all procedures in the named module are below the exclusion contour; if the first word is '`internal`', then all internal (non-exported) procedures in the named module are below the exclusion contour. Here is an example of an exclusion contour file.

```
all bag
all list
all map
internal set
```

8.8 Using threadscope

The ThreadScope tools are not distributed with Mercury. For information about how to install them please see the '`README.ThreadScope`' file included in the Mercury distribution.

ThreadScope provides two programs that can be used to view profiles in '`.eventlog`' files. The first, '`show-ghc-events`', lists the ThreadScope events sorted from the earliest to the latest, while the second, '`threadscope`' provides a graphical display for browsing the profile.

Both programs accept the name of a '`.eventlog`' file on the command line. The '`threadscope`' program also provides a menu from which users can choose a file to open.

8.9 Profiling and shared libraries

On some operating systems, Mercury's profiling doesn't work properly with shared libraries. The symptom is errors ('`map.lookup failed`') or warnings from '`mprof`'. On some systems, the problem occurs because the C implementation fails to conform to the semantics specified by the ISO C standard for programs that use shared libraries. For other systems, we have not been able to analyze the cause of the failure (but we suspect that the cause may be the same as on those systems where we have been able to analyze it).

If you get errors or warnings from '`mprof`', and your program is dynamically linked, try rebuilding your application statically linked, e.g. by using '`MLFLAGS=--static`' in your Mmakefile. Another work-around that sometimes works is to set the environment variable `LD_BIND_NOW` to a non-null value before running the program.

9 Invocation

This section contains a brief description of all the options available for '`mmc`', the Mercury compiler. Sometimes this list is a little out-of-date; use '`mmc --help`' to get the most up-to-date list.

9.1 Invocation overview

`mmc` is invoked as

```
mmc [options] arguments
```

Arguments can be either module names or file names. Arguments ending in '`.m`' are assumed to be file names, while other arguments are assumed to be module names, with '`.`' (rather than '`__`') as module qualifier. If you specify a module name such as '`foo.bar.baz`', the compiler will look for the source in files '`foo.bar.baz.m`', '`bar.baz.m`', and '`baz.m`', in that order.

Options are either short (single-letter) options preceded by a single '`-`', or long options preceded by '`--`'. Options are case-sensitive. We call options that do not take arguments *flags*. Single-letter flags may be grouped with a single '`-`', e.g. '`-vVc`'. Single-letter flags may be negated by appending another trailing '`-`', e.g. '`-v-`'. Long flags may be negated by preceding them with '`no-`', e.g. '`--no-verbose`'.

9.2 Warning options

`-w`
`--inhibit-warnings`
: Disable all warning messages.

`--inhibit-style-warnings`
: Disable all warning messages about programming style.

`--halt-at-warn`
: This option causes the compiler to treat all warnings as if they were errors when generating target code. This means that if the compiler issues any warning, it will not generate target code — instead, it will return a non-zero exit status.

`--halt-at-warn-make-int`
`--halt-at-warn-make-interface`
: This option causes the compiler to treat all warnings as if they were errors when generating an interface file (a '`.int`', '`.int0`', '`.int2`' or '`.int3`' file). This means that if the compiler issues any warnings at that time, it will not generate the interface file — instead, it will return a non-zero exit status.

`--halt-at-warn-make-opt`
: This option causes the compiler to treat all warnings as if they were errors when generating an optimization file (a '`.opt`' or '`.trans_opt`' file). This means that if the compiler issues any warnings at that time, it will not generate the optimization file — instead, it will return a non-zero exit status.

`--halt-at-syntax-error`
: This option causes the compiler to halt immediately after syntax checking and not do any semantic checking if it finds any syntax errors in the program.

`--no-halt-at-invalid-interface`
: This option operates when the compiler is invoked with the –make–interface option to generate .int and .int2 files for one or more modules. In its default setting, '`--halt-at-invalid-interface`', it causes the compiler to check the consistency of those parts of each of those modules that are intended to end up in the .int and .int2 files. If these checks find any problems, the compiler will stop *without* generating those files after printing an error message for each problem.

  Users can prevent this behavior, and thus allow the generation of invalid interface files, by specifying '`--no-halt-at-invalid-interface`'.

`--no-warn-accumulator-swaps`
: Do not warn about argument order rearrangement caused by '`--introduce-accumulators`'.

`--no-warn-singleton-vars`
`--no-warn-singleton-variables`
: Do not warn about variables which only occur once.

`--no-warn-missing-det-decls`
: For predicates that are local to a module (those that are not exported), don't issue a warning if the '`pred`' or '`mode`' declaration does not have a determinism annotation. Use this option if you want the compiler to perform automatic determinism inference for non-exported predicates.

`--no-warn-det-decls-too-lax`
: Do not warn about determinism declarations which could have been stricter.

`--no-warn-inferred-erroneous`
: Do not warn about procedures whose determinism is inferred erroneous but whose determinism declarations are laxer.

`--no-warn-insts-without-matching-type`
: Do not warn about insts that are not consistent with any types in scope.

`--warn-insts-with-functors-without-type`
: Warn about insts that do specify functors but do not specify what type they are for.

`--warn-unused-imports`
: Warn about modules that are imported but not used.

`--no-warn-nothing-exported`
: Do not warn about modules whose interface sections have no exported predicates, functions, insts, modes or types.

`--warn-unused-args`
: Warn about predicate or function arguments which are not used.

`--no-warn-interface-imports`
: Do not warn about modules imported in the interface which are not used in the interface.

`--warn-interface-imports-in-parents`
: Warn about modules imported in the interface a parent module which are not used in the interface of that module.

`--warn-missing-opt-files`
: Warn about '`.opt`' files that cannot be opened.

`--warn-missing-trans-opt-files`
: Warn about '`.trans_opt`' files that cannot be opened.

`--warn-inconsistent-pred-order`
`--warn-inconsistent-pred-order-clauses`
: Generate a warning if the order of the definitions does not match the order of the declarations for either the exported predicates and functions of the module, or for the nonexported predicates and functions of the module. Applies for definitions by Mercury clauses.

`--warn-inconsistent-pred-order-foreign-procs`
: Generate a warning if the order of the definitions does not match the order of the declarations for either the exported predicates and functions of the module, or for the nonexported predicates and functions of the module. Applies for definitions by either Mercury clauses or foreign_proc pragmas.

`--no-warn-non-contiguous-decls`
: Do not generate a warning if the mode declarations of a predicate or function don't all immediately follow its predicate or function declaration.

`--no-warn-non-contiguous-clauses`
: Do not generate a warning if the clauses of a predicate or function are not contiguous.

`--warn-non-contiguous-foreign-procs`
: Generate a warning if the clauses and foreign_procs of a predicate or function are not contiguous.

`--warn-non-stratification`
: Warn about possible non-stratification of the predicates and/or functions in the module. Non-stratification occurs when a predicate or function can call itself negatively through some path along its call graph.

`--no-warn-simple-code`
: Disable warnings about constructs which are so simple that they are likely to be programming errors.

`--warn-duplicate-calls`
: Warn about multiple calls to a predicate with the same input arguments.

`--warn-implicit-stream-calls`
: Warn about calls to I/O predicates that could take explicit stream arguments, but do not do so.

`--no-warn-missing-module-name`
: Disable warnings for modules that do not start with a '`:- module`' declaration.

`--no-warn-wrong-module-name`
: Disable warnings for modules whose '`:- module`' declaration does not match the module's file name.

`--no-warn-smart-recompilation`
: Disable warnings from the smart recompilation system.

`--no-warn-undefined-options-vars`
`--no-warn-undefined-options-variables`
: Do not warn about references to undefined variables in options files with '`--make`'.

`--warn-suspicious-recursion`
: Warn about recursive calls which are likely to have problems, such as leading to infinite recursion.

`--warn-non-tail-recursion` *type*
: Warn about any recursive calls that are not tail recursive. Implies the next two options. Warn about any recursive calls that are not tail recursive. *type* may be '`self`', '`self-and-mutual`' or '`none`'.

`--warn-obvious-non-tail-recursion`
: Warn about recursive calls that are not tail calls even if they obviously cannot be tail calls, because they are followed by other recursive calls.

`--no-warn-target-code`
: Disable warnings from the compiler used to process the target code (e.g. gcc).

`--no-warn-up-to-date`
: Do not warn if targets specified on the command line with '`--make`' are already up-to-date.

`--no-warn-stubs`
: Disable warnings about procedures for which there are no clauses. Note that this option only has any effect if the '`--allow-stubs`' option (see Section 9.6 [Language semantics options], page 106) is enabled.

`--warn-dead-procs`
`--warn-dead-procedures`
: Warn about procedures which are never called.

`--warn-dead-preds`
`--warn-dead-predicates`
: Warn about predicates that have no procedures which are ever called.

`--no-warn-table-with-inline`
: Disable warnings about tabled procedures that also have a '`pragma inline`' declaration.

`--no-warn-non-term-special-preds`
: Do not warn about types that have user-defined equality or comparison predicates that cannot be proved to terminate. This option is only enabled when termination analysis is enabled. (See Section 9.7 [Termination analysis options], page 107 for further details).

`--no-warn-known-bad-format-calls`
: Do not warn about calls to string.format or io.format that the compiler knows for sure contain mismatches between the format string and the supplied values.

`--no-warn-only-one-format-string-error`
: If a format string has more than one mismatch with the supplied values, generate a warning for all mismatches, not just the first. The later mismatches may be avalanche errors caused by earlier mismatches.

`--warn-unknown-format-calls`
: Warn about calls to string.format or io.format for which the compiler cannot tell whether there are any mismatches between the format string and the supplied values.

`--no-warn-obsolete`
: Do not warn about calls to predicates or functions that have been marked as obsolete.

`--inform-ite-instead-of-switch`
: Generate informational messages for if-then-elses that could be replaced by switches.

`--inform-incomplete-switch`
: Generate informational messages for switches that do not cover all the function symbols that the switched-on variable could be bound to.

`--inform-incomplete-switch-threshold` *n*
: –inform-incomplete-switch-threshold <N> Have the `--inform-incomplete-switch` option generate its messages only for switches that *do* cover at least *n* percent of the function symbols that the switched-on variable could be bound to.

`--no-warn-unresolved-polymorphism`
: Do not warn about unresolved polymorphism.

`--warn-suspicious-foreign-procs`
: Warn about possible errors in the bodies of foreign procedures. When enabled, the compiler attempts to determine whether the success indicator for a foreign procedure is correctly set, and whether the foreign procedure body contains operations that are not allowed (for example, `return` statements in a C foreign procedure). Note that since the compiler's ability to parse foreign language code is limited, some warnings reported by this option may be spurious, and some actual errors may not be detected at all.

`--warn-suspicious-foreign-code`
: Warn about possible errors in the bodies of foreign code pragmas. Note that since the compiler's ability to parse foreign language code is limited, some warnings reported by this option may be spurious, and some actual errors may not be detected at all.

`--no-warn-state-var-shadowing`
: Do not warn about one state variable shadowing another.

`--no-warn-suspected-occurs-check-failure`
: Do not warn about code that looks like it unifies a variable with a term that contains that same variable. Such code cannot succeed because it fails what is called the '`occurs check`'.

`--warn-potentially-ambiguous-pragma`
: Generate warnings for pragmas that do not specify whether they are for a predicate or a function.

`--no-inform-inferred`
: Do not generate messages about inferred types or modes.

`--no-inform-inferred-types`
: Do not generate messages about inferred types.

`--no-inform-inferred-modes`
: Do not generate messages about inferred modes.

`--inform-suboptimal-packing`
: Generate messages if the arguments of a data constructor could be packed more tightly if they were reordered.

9.3 Verbosity options

`-v`
`--verbose`
: Output progress messages at each stage in the compilation.

`-V`
`--very-verbose`
: Output very verbose progress messages.

`-E`
`--verbose-error-messages`
: Explain error messages. Asks the compiler to give you a more detailed explanation of any errors it finds in your program.

`--no-verbose-make`
: Disable messages about the progress of builds using the '`--make`' option.

`--verbose-commands`
: Output each external command before it is run. Note that some commands will only be printed with '`--verbose`'.

`--verbose-recompilation`
: When using '`--smart-recompilation`', output messages explaining why a module needs to be recompiled.

`--find-all-recompilation-reasons`
: Find all the reasons why a module needs to be recompiled, not just the first. Implies '`--verbose-recompilation`'.

`--output-compile-error-lines` *n*
: With '`--make`', output the first *n* lines of the '`.err`' file after compiling a module (default: 15).

`--report-cmd-line-args`
: Report the command line arguments.

`--report-cmd-line-args-in-doterr`
: Report the command line arguments for compilations whose output mmake normally redirects to a '`.err`' file.

`-S`
`--statistics`
: Output messages about the compiler's time/space usage. At the moment this option implies '`--no-trad-passes`', so you get information at the boundaries between phases of the compiler.

`--proc-size-statistics` *filename*
: Append information about the size of each procedure in the module in terms of goals and variables to the end of the named file.

`--limit-error-contexts` *filename:minline1-maxline1,minline2-maxline2*
: Print errors and warnings for the named file only when their line number is in one of the specified ranges. The minimum or maximum line number in each range may be omitted, in which case the range has no lower or upper bound respectively. Multiple '`--limit-error-context`' options accumulate. If more than one '`--limit-error-context`' option is given for the same file, only the last one will have an effect. If the file name and colon are missing, the limit will apply to all files.

`-N`
`--debug-modes`
: Output debugging traces of the mode checking.

`--debug-modes-verbose`
: Output detailed debugging traces of the mode checking.

`--debug-modes-pred-id` *predid*
: With '`--debug-modes`', restrict the debugging traces to the mode checking of the predicate or function with the specified pred id.

`--debug-det`
`--debug-determinism`
: Output detailed debugging traces of determinism analysis.

`--debug-opt`
: Output detailed debugging traces of the optimization process.

`--debug-opt-pred-id` *predid*
: Output detailed debugging traces of the optimization process only for the predicate/function with the specified pred id. May be given more than once.

`--debug-opt-pred-name` *name*
: Output detailed debugging traces of the optimization process only for the predicate/function with the specified name. May be given more than once.

`--debug-pd`
: Output detailed debugging traces of the partial deduction and deforestation process.

`--debug-liveness <n>`
: Output detailed debugging traces of the liveness analysis of the predicate with the given predicate id.

`--debug-make`
: Output detailed debugging traces of the '`--make`' option.

`--debug-intermodule-analysis`
: Output detailed debugging traces of the '`--intermodule-analysis`' option.

`--debug-indirect-reuse`
: Output detailed debugging traces of the indirect reuse pass of '`--structure-reuse`' option.

`--debug-type-rep`
: Output debugging traces of type representation choices.

9.4 Output options

These options are mutually exclusive. If more than one of these options is specified, only the first in this list will apply. If none of these options are specified, the default action is to compile and link the modules named on the command line to produce an executable.

`-f`
`--generate-source-file-mapping`
: Output the module name to file name mapping for the list of source files given as non-option arguments to '`mmc`' to '`Mercury.modules`'. This must be done before '`mmc --generate-dependencies`' if there are any modules for which the file name does not match the module name. If there are no such modules the mapping need not be generated.

`-M`
`--generate-dependencies`
: Output "Make"-style dependencies for the module and all of its dependencies to '*`module`*`.dep`', '*`module`*`.dv`' and the relevant '`.d`' files.

`--generate-dependency-file`
: Output 'Make'-style dependencies for the module to '*`module`*`.d`''.

`--generate-module-order`
: Output the strongly connected components of the module dependency graph in top-down order to '*`module`*`.order`'. Effective only if '`--generate-dependencies`' is also specified.

`--generate-standalone-interface` *basename*
: Output a stand-alone interface. *basename* is used as the basename of any files generated for the stand-alone interface. (See see Chapter 13 [Stand-alone interfaces], page 143 for further details.)

`--generate-mmc-deps`
`--generate-mmc-make-module-dependencies`
: Generate dependencies for use by '`mmc --make`' even when using Mmake. This is recommended when building a library for installation.

`-i`
`--make-int`
`--make-interface`
: Write the module interface to '*`module`*`.int`'. Also write the short interface to '*`module`*`.int2`'.

`--make-short-int`
`--make-short-interface`
: Write the unqualified version of the short interface to '*`module`*`.int3`'.

`--make-priv-int`
`--make-private-interface`
: Write the module's private interface (used for compiling nested submodules) to '*module*`.int0`'.

`--make-opt-int`
`--make-optimization-interface`
: Write information used for inter-module optimization to '*module*`.opt`'.

`--make-trans-opt`
`--make-transitive-optimization-interface`
: Write the '*module*`.trans_opt`' file. This file is used to store information used for inter-module optimization. The information is read in when the compiler is invoked with the '`--transitive-intermodule-optimization`' option. The file is called the "transitive" optimization interface file because a '`.trans_opt`' file may depend on other '`.trans_opt`' and '`.opt`' files. In contrast, a '`.opt`' file can only hold information derived directly from the corresponding '`.m`' file.

`--make-xml-documentation`
: Output an XML representation of all the declarations in the module into the file '*module*`.xml`'. This XML file can then be transformed via a XSL transform into another documentation format.

`-P`
`--pretty-print`
`--convert-to-mercury`
: Convert to Mercury. Output to file '*module*`.ugly`'. This option acts as a Mercury ugly-printer. (It would be a pretty-printer, except that comments are stripped and nested if-then-elses are indented too much — so the result is rather ugly.)

`--typecheck-only`
: Just check the syntax and type-correctness of the code. Don't invoke the mode analysis and later passes of the compiler. When converting Prolog code to Mercury, it can sometimes be useful to get the types right first and worry about modes second; this option supports that approach.

`-e`
`--errorcheck-only`
: Check the module for errors, but do not generate any code.

`-C`
`--target-code-only`
: Generate target code (i.e. C in '*module*`.c`', C# in '*module*`.cs`', or Java in '*module*`.java`', but not object code.

`-c`
`--compile-only`
: Generate C code in '*`module`*`.c`' and object code in '*`module`*`.o`' but do not attempt to link the named modules.

`--output-grade-string`
: Compute the canonical string representing the currently selected grade, and print it on the standard output.

`--output-link-command`
: Print to standard output the command used to link executables.

`--output-shared-lib-link-command`
: Print to standard output the command used to link shared libraries.

`--output-stdlib-grades`
: Print to standard output the list of compilation grades in which the Mercury standard library is available with this compiler.

`--output-libgrades`
: Print to standard output the list of compilation grades in which a library to be installed should be built.

`--output-cc`
: Print to standard output the command used to invoke the C compiler.

`--output-cc-type`
`--output-c-compiler-type`
: Print the C compiler type to the standard output.

`--output-cflags`
: Print to standard output the flags with which the C compiler will be invoked.

`--output-csharp-compiler`
: Print to standard output the command used to invoke the C# compiler.

`--output-csharp-compiler-type`
: Print the C# compiler type to the standard output.

`--output-library-link-flags`
: Print to standard output the flags that are passed to the linker in order to link against the current set of libraries. This includes the standard library, as well as any other libraries specified via the '`--ml`' option.

`--output-grade-defines`
: Print to standard output the flags that are passed to the C compiler to define the macros whose values specify the compilation grade.

`--output-c-include-dir-flags`
`--output-c-include-directory-flags`
: Print to standard output the flags that are passed to the C compiler to specify which directories to search for C header files. This includes the C header files from the standard library.

`--output-target-arch`
: Print the target architecture to the standard output.

`--output-class-dir`
`--output-class-directory`
`--output-java-class-dir`
`--output-java-class-directory`
: Print to standard output the name of the directory in which generated Java class files will be placed.

`--output-stdlib-modules`
: Print to standard output the names of the modules in the Mercury standard library.

9.5 Auxiliary output options

`--smart-recompilation`
: When compiling, write program dependency information to be used to avoid unnecessary recompilations if an imported module's interface changes in a way which does not invalidate the compiled code. '`--smart-recompilation`' does not yet work with '`--intermodule-optimization`'.

`--trace-level` *level*
: Generate code that includes the specified level of execution tracing. The *level* should be one of '`none`', '`shallow`', '`deep`', '`rep`' and '`default`'. See Chapter 7 [Debugging], page 16.

`--trace-optimized`
: Do not disable optimizations that can change the trace.

`--profile-optimized`
: Do not disable optimizations that can distort deep profiles.

`--no-delay-death`
: When the trace level is '`deep`', the compiler normally preserves the values of variables as long as possible, even beyond the point of their last use, in order to make them accessible from as many debugger events as possible. However, it will not do this if this option is given.

`--delay-death-max-vars` *N*
: Delay the deaths of variables only when the number of variables in the procedure is no more than *N*. The default value is 1000.

`--stack-trace-higher-order`
: Enable stack traces through predicates and functions with higher-order arguments, even if stack tracing is not supported in general.

`--generate-bytecode`
: Output a bytecode form of the module for use by an experimental debugger.

`--auto-comments`
: Output comments in the generated target language file. This is primarily useful for trying to understand how the generated target language code relates to the source code, e.g. in order to debug the compiler. The code may be easier to understand if you also use the '`--no-llds-optimize`' or '`--no-mlds-optimize`' options.

`-n-`
`--no-line-numbers`
: Put source line numbers in the generated code. The generated code may be in C (the usual case) or in Mercury (with '`--convert-to-mercury`').

`--no-line-numbers-around-foreign_code`
: Do not put source line numbers into the generated code around inclusions of foreign language code.

`--line-numbers-for-c-headers`
: Put source line numbers in the generated C header files. This can make it easier to track down any problems with C code in foreign_decl pragmas, but may cause unnecessary recompilations of other modules if any of these line numbers changes (e.g. because the location of a predicate declaration changes in the Mercury source file).

`--max-error-line-width` *N*
: Set the maximum width of an error message line to *N* characters (unless a long single word forces the line over this limit). Specifying '`--no-max-error-line-width`' removes the limit.

`--show-definitions`
: Write out a list of the types, insts, modes, predicates, functions, typeclasses and instances defined in the module to '*module*`.defns`'.

`--show-definition-line-counts`
: Write out a list of the predicates and functions defined in the module, together with the names of the files containing them and their approximate line counts, to '*module*`.defn_line_counts`'. The list will be ordered on the names and arities of the predicates and functions.

`--show-definition-extents`
: Write out a list of the predicates and functions defined in the module, together with the approximate line numbers of their first and last lines, to '*module*`.defn_extents`'. The list will be ordered on the starting line numbers of the predicates and functions.

`--show-local-type-representations`
: Write out information about the representations of all types defined in the module being compiled to '*module*`.type_repns`'.

`--show-all-type-representations`
: Write out information about the representations of all types visible in the module being compiled to '*module*`.type_repns`'.

`--show-dependency-graph`
: Write out the dependency graph to '*module*`.dependency_graph`'.

`--imports-graph`
: Write out the imports graph to '*module*`.imports_graph`'. The imports graph contains the directed graph module A imports module B. The resulting file can be processed by the graphviz tools. Effective only if '`--generate-dependencies`' is also specified.

`-d` *stage*
`--dump-hlds` *stage*
: Dump the HLDS (a high-level intermediate representation) after the specified stage number or stage name to '*module*`.hlds_dump.`*num-name*'. Stage numbers range from 1 to 599; not all stage numbers are valid. If a stage number is followed by a plus sign, all stages after the given stage will be dumped as well. The special stage name '`all`' causes the dumping of all stages. Multiple dump options accumulate.

`--dump-hlds-options` *options*
: With '`--dump-hlds`', include extra detail in the dump. Each type of detail is included in the dump if its corresponding letter occurs in the *options* argument. These details are:

  '`a`' argument modes in unifications

  '`b`' builtin flags on calls

  '`c`' contexts of goals and types

  '`d`' determinism of goals

  '`e`' created, removed, carried, allocated into, and used regions

  '`f`' follow_vars sets of goals

  '`g`' goal feature lists

‘i’	variables whose instantiation changes
‘l’	pred/mode ids and unify contexts of called predicates
‘m’	mode information about clauses
‘n’	nonlocal variables of goals
‘p’	pre-birth, post-birth, pre-death and post-death sets of goals
‘r’	resume points of goals
‘s’	store maps of goals
‘t’	results of termination analysis
‘u’	unification categories and other implementation details of unifications
‘v’	variable numbers in variable names
‘x’	predicate type information
‘y’	structured insts in the arg-modes of unification
‘z’	purity annotations on impure and semipure goals
‘A’	argument passing information
‘B’	mode constraint information
‘C’	clause information
‘D’	instmap deltas of goals (meaningful only with ‘i’)
‘E’	deep profiling information
‘G’	compile-time garbage collection information
‘I’	imported predicates
‘L’	restrict the output of type table entries to local types only
‘M’	mode and inst information
‘O’	dependency and dependency ordering information
‘P’	goal id and path information
‘R’	live forward use, live backward use and reuse possibilities
‘S’	information about structure sharing
‘T’	type and typeclass information
‘U’	unify and compare predicates
‘X’	constant structures
‘Z’	information about globals structs representing call and answer tables

`--dump-hlds-pred-id` *predid*
: With '`--dump-hlds`', restrict the output to the HLDS of the predicate or function with the specified pred id. May be given more than once.

`--dump-hlds-pred-name` *name*
: With '`--dump-hlds`', restrict the output to the HLDS of the predicate or function with the specified name. May be given more than once.

`--dump-hlds-inst-limit` *N*
: Dump at most *N* insts in each inst table.

`--dump-hlds-file-suffix`
: Append the given suffix to the names of the files created by the '`--dump-hlds`' option.

`--dump-same-hlds`
: Create a file for a HLDS stage even if the file notes only that this stage is identical to the previously dumped HLDS stage.

`--dump-mlds` *stage*
: Dump the MLDS (a C-like intermediate representation) after the specified stage number or stage name. The MLDS is converted to a C source file/header file pair, which is dumped to '*module*`.c_dump.`*num-name*' and '*module*`.h_dump.`*num-name*'. Stage numbers range from 1 to 99; not all stage numbers are valid. The special stage name '`all`' causes the dumping of all stages. Multiple dump options accumulate.

`--dump-mlds-pred-name` *name*
: Dump the MLDS (medium level intermediate representation) of the predicate or function with the specified name at the stages specified by the '`--dump-mlds`' option. The dump file will consist of the predicates and functions named by all the occurrences of this option (there may be more than one), and nothing else.

`--verbose-dump-mlds` *stage*
: Dump the internal compiler representation of the MLDS, after the specified stage number or stage name, to '*module*`.mlds_dump.`*num-name*'.

`--mode-constraints`
: Perform constraint based mode analysis on the given modules. At the moment, the only effect of this is to include more information in HLDS dumps, to allow the constraint based mode analysis algorithm to be debugged.

`--simple-mode-constraints`
: Ask for the simplified variant of constraint based mode analysis, in which there is only one constraint variable per program variable, rather than one constraint

variable per node in the inst graph of a program variable. This option is ignored unless '`--mode-constraints`' is also given.

`--benchmark-modes`
: Output information about the performance of the constraint based mode analysis algorithm.

`--benchmark-modes-repeat` *num*
: Specifies the number of times the mode analysis algorithm should run. More repetitions may smooth out fluctuations due to background load or clock granularity. This option is ignored unless '`--benchmark-modes`' is also given.

9.6 Language semantics options

See the Mercury language reference manual for detailed explanations of these options.

`--no-reorder-conj`
: Execute conjunctions left-to-right except where the modes imply that reordering is unavoidable.

`--no-reorder-disj`
: Execute disjunctions strictly left-to-right.

`--no-fully-strict`
: Allow infinite loops or goals with determinism erroneous to be optimised away.

`--allow-stubs`
: Allow procedures to have no clauses. Any calls to such procedures will raise an exception at run-time. This option is sometimes useful during program development. (See also the documentation for the '`--warn-stubs`' option in Section 9.2 [Warning options], page 89.)

`--infer-all`
: An abbreviation for '`--infer-types --infer-modes --infer-det`'.

`--infer-types`
: If there is no type declaration for a predicate or function, try to infer the type, rather than just reporting an error.

`--infer-modes`
: If there is no mode declaration for a predicate, try to infer the modes, rather than just reporting an error.

`--no-infer-det`
`--no-infer-determinism`
: If there is no determinism declaration for a procedure, don't try to infer the determinism, just report an error.

`--type-inference-iteration-limit` *n*
: Perform at most *n* passes of type inference (default: 60).

`--mode-inference-iteration-limit` *n*
: Perform at most *n* passes of mode inference (default: 30).

9.7 Termination analysis options

For detailed explanations, see the "Termination analysis" section of the "Implementation-dependent extensions" chapter in the Mercury Language Reference Manual.

`--enable-term`
`--enable-termination`
: Enable termination analysis. Termination analysis analyses each mode of each predicate to see whether it terminates. The '`terminates`', '`does_not_terminate`' and '`check_termination`' pragmas have no effect unless termination analysis is enabled. When using termination, '`--intermodule-optimization`' should be enabled, as it greatly improves the accuracy of the analysis.

`--chk-term`
`--check-term`
`--check-termination`
: Enable termination analysis, and emit warnings for some predicates or functions that cannot be proved to terminate. In many cases in which the compiler is unable to prove termination, the problem is either a lack of information about the termination properties of other predicates, or the fact that the program used language constructs (such as higher-order calls) which cannot be analysed. In these cases the compiler does not emit a warning of non-termination, as it is likely to be spurious.

`--verb-chk-term`
`--verb-check-term`
`--verbose-check-termination`
: Enable termination analysis, and emit warnings for all predicates or functions that cannot be proved to terminate.

`--term-single-arg` *limit*
`--termination-single-argument-analysis` *limit*
: When performing termination analysis, try analyzing recursion on single arguments in strongly connected components of the call graph that have up to *limit* procedures. Setting this limit to zero disables single argument analysis.

`--termination-norm` *norm*
: The norm defines how termination analysis measures the size of a memory cell. The '`simple`' norm says that size is always one. The '`total`' norm says that it is the number of words in the cell. The '`num-data-elems`' norm says that it is the number of words in the cell that contain something other than pointers to cells of the same type.

`--term-err-limit` *limit*
`--termination-error-limit` *limit*
: Print at most *n* reasons for any single termination error.

`--term-path-limit` *limit*
`--termination-path-limit` *limit*
: Perform termination analysis only on predicates with at most *n* paths.

9.8 Compilation model options

The following compilation options affect the generated code in such a way that the entire program must be compiled with the same setting of these options, and it must be linked to a version of the Mercury standard library which has been compiled with the same setting. (Attempting to link object files compiled with different settings of these options will generally result in an error at link time, typically of the form '`undefined symbol MR_grade_...`' or '`symbol MR_runtime_grade multiply defined`'.)

The options below must be passed to '`mgnuc`', '`c2init`' and '`ml`' as well as to '`mmc`'. If you are using Mmake, then you should specify these options in the '`GRADEFLAGS`' variable rather than specifying them in '`MCFLAGS`', '`MGNUCFLAGS`' and '`MLFLAGS`'.

9.8.1 Grades and grade components

`-s` *grade*
`--grade` *grade*
: Select the compilation model. The *grade* should be a '`.`' separated list in which the first element is a '`base grade`', and each following element (if any) is a '`grade modifier`'.

  The base grade specifies what target language to compile the Mercury program to, and if the compiler can do this in several different ways, selects one of those ways. The available base grades are the following.

  '`hlc`'
  : This base grade calls for generating idiomatic C, which we call high-level C.

  '`none`', '`reg`', and '`asm_fast`'
  : These base grades call for generating assembly-like C code, which we call low-level C. '`none`' calls for generating standard C without using any GNU C extensions. '`reg`' and '`asm_fast`' both call for the use of GNU C's extension for storing global variables in registers, while '`asm_fast`' also calls for the use of two GNU C extensions that together allow direct gotos to labels in other functions, even

in other modules. In general, the more GNU C extensions are used, the faster the program will be, but some platforms, compilers or compiler versions do not support specific extensions.

‘`csharp`’ This base grade calls for generating C#.

‘`java`’ This base grade calls for generating Java.

The default base grade is system dependent, but will be either ‘`hlc`’ or ‘`asm_fast`’, as these are the two fastest.

The grade modifiers may be given in any order. Each grade modifier sets one or more compilation model options. The available options each belong to a set of mutually exclusive alternatives governing a single aspect of the compilation model. The set of aspects and their alternatives are as follows. Note that not all grade modifiers are applicable to all grades (most apply only when targeting C), and not all grade modifiers are compatible with each other.

garbage collection

The default is no garbage collection beyond what the target language provides. C# and Java have their own builtin garbage collectors, but C does not. Since garbage collection is essential for all programs other than those with *very* short runtimes, base grades that target C are usually followed by the ‘`.gc`’ grade modifier, which calls for the use of the Boehm-Demers-Weiser conservative collector for C.

profiling The default is no profiling. The grade modifier ‘`.prof`’ calls for gprof-style profiling; the grade modifier ‘`.memprof`’ calls for profiling of memory usage; while the grade modifier ‘`.profdeep`’ calls for deep profiling. See Chapter 8 [Profiling], page 80 for an explanation of the differences between these forms of profiling, all of which are available only when targeting C.

trailing The default is no trailing. Applications that need trailing should specify the grade modifier ‘`.tr`’. ‘`.trseg`’ is currently accepted as synonym for ‘`.tr`’. Trailing is available only when targeting C.

(A trail is a data structure that records changes to a mutable data store during normal execution so that each of those changes can be unwound (undone) when execution backtracks to the point in time before it was made. Mercury normally needs a trail only when executing programs that use constraint solvers. While the act of posting a constraint in such a program is a declarative action, its implementation requires updating a constraint store.)

floating point precision

By default when targeting C, the Mercury ‘`float`’ type is implemented as an IEEE 754 double-precision floating point number, which occupies 64 bits of space. On platforms whose word size is 32 bits, users whose programs do not require double precision may improve memory consumption and speed by specifying the grade

modifier '`.spf`' (short for single-precision float), which calls for implementing all Mercury floats as IEEE 754 single-precision floating point numbers. Note that on 64 bit platforms, using single-precision floats definitely will *not* improve memory consumption, and will probably not improve speed. On the other hand, it may simplify the use of C APIs that exclusively use single precision floats.

For target languages other than C, the Mercury '`float`' type is always implemented as a double-precision floating point number, and the '`.spf`' modifier is not supported.

stack size

When generating low-level C, Mercury implements its own stacks (two of them). The default is to make each stack a fixed size (usually a relatively large size, though the size of each stack can be controlled using runtime options). We make the page(s) at the tops of the stacks inaccessible, so that a stack overflow, instead of accessing and overwriting memory effectively randomly, causes the operating system to send a signal to the program. If not caught and handled, this signal will abort the program, minimizing the damage.

However, for programs for which this is not acceptable, users can specify the '`.stseg`' grade component. This calls for each stack to be composed of small memory segments chained together in a list. When there is no room in the current segment for a new stack frame, we simply allocate a new segment and add it to the list. This approach has higher overhead, since calls to, and returns from, procedures must execute more code, but it avoids imposing any limit on stack size other than the size of available memory.

When targeting anything other than low-level C, the stack is always managed by the implementation of the target language, so for them, the '`.stseg`' modifier is neither relevant nor supported.

debugging

The default is to generate executables that do not support debugging. However, when generating low-level C code, specifying one of the grade modifiers '`.debug`' and '`.decldebug`' will cause the compiler to generate executables that can be debugged using the Mercury debugger '`mdb`'. The difference between them is that the declarative debugging aspects of '`mdb`' will work only with '`.decldebug`'. The price of this is that '`.decldebug`' results in significantly larger executable files.

threads

The default is whatever thread support is provided by the target language. When targeting C, thread support can be enabled by specifying the grade modifier '`.par`'. When targeting low-level C, this also enables the use of the parallel conjunction operator '`&`'. Since Mercury implements parallel conjunctions only in low-level C grades with the '`.par`' grade modifier, in every other situation, the compiler silently converts every occurrence of '`&`' to a comma, the usual *sequential* conjunctions operator.

thread profiling
: In low-level C grades with the grade modifier '`.par`', users can enable support for ThreadScope-style thread profiling by also specifying the grade module '`.threadscope`'. The default is no support for thread profiling. Note that this form of profiling is experimental, and it is not currently supported.

minimal model tabling
: The default is no support for minimal model evaluation. When targeting low-level C, users can specify the grade modifier '`.mm`', which enables support for minimal model tabled evaluation of procedures. The grade component '`.mmsc`' is a synonym for '`.mm`', standing for minimal model via stack copying, since the standard implementation works by copying stack segments. (The synonym exists because Mercury also has another implementation of minimal model tabling. This other implementation, which is incomplete and was only ever useful for experiments, is based on a completely different implementation technique.)

The default grade is system-dependent; it is chosen at installation time by '`configure`', the auto-configuration script, but can be overridden if desired with the environment variable `MERCURY_DEFAULT_GRADE`. On any given particular installation, the Mercury runtime system and Mercury standard library will be installed in only a subset of the possible grades; you can find out which grades these are by invoking the Mercury compiler with the '`--output-stdlib-grades`' option. Attempting to use a grade which has not been installed will result in an error at link time. (The error message will typically be something like '`ld: can't find library for -lmercury`'.)

The tables below show the options that are selected by each base grade and grade modifier; they are followed by descriptions of those options.

Grade	*Options implied.*
'`hlc`'	`--target c --high-level-code`.
'`none`'	`--target c --no-gcc-global-registers --no-gcc-nonlocal-gotos --no-asm-labels`.
'`reg`'	`--target c --gcc-global-registers --no-gcc-nonlocal-gotos --no-asm-labels`.
'`asm_fast`'	`--target c --gcc-global-registers --gcc-nonlocal-gotos --asm-labels`.
'`csharp`'	`--target csharp --high-level-code`.
'`java`'	`--target java --high-level-code`.
'`.gc`'	`--gc boehm`.
'`.prof`'	`--profiling`.

‘`.memprof`’
`--memory-profiling`.

‘`.profdeep`’
`--deep-profiling`.

‘`.tr`’ `--use-trail --no-trail-segments`.

‘`.trseg`’ `--use-trail --trail-segments`.

‘`.spf`’ `--single-prec-float`

‘`.stseg`’ `--stack-segments`

‘`.debug`’ `--debug`.

‘`.decldebug`’
`--decl-debug`.

‘`.par`’ `--parallel`.

‘`.threadscope`’
`--threadscope`.

9.8.2 Target options

`--target c` (grades: none, reg, asm_fast, hlc)
`--csharp`, `--target csharp` (grades: csharp)
`--java`, `--target java` (grades: java)
Specify the target language used for compilation: C, C# or Java. C means ANSI/ISO C, optionally with GNU C extensions (see below). Targets other than C imply ‘`--high-level-code`’.

`--compile-to-c`
`--compile-to-C`
An abbreviation for ‘`--target c --target-code-only`’. Generate C code in ‘*`module`*`.c`’, but do not invoke the C compiler to generate object code.

`--csharp-only`
An abbreviation for ‘`--target csharp --target-code-only`’. Generate C# code in ‘*`module`*`.cs`’, but do not invoke the C# compiler to produce CIL bytecode.

`--java-only`
An abbreviation for ‘`--target java --target-code-only`’. Generate Java code in ‘*`module`*`.java`’, but do not invoke the Java compiler to produce Java bytecode.

9.8.3 LLDS back-end compilation model options

`--gcc-global-registers` (grades: reg, asm_fast)
`--no-gcc-global-registers` (grades: none)
: Specify whether or not to use GNU C's global register variables extension. This option is ignored if the '`--high-level-code`' option is enabled.

`--gcc-non-local-gotos` (grades: asm_fast)
`--no-gcc-non-local-gotos` (grades: none, reg)
: Specify whether or not to use GNU C's "labels as values" extension. This option is ignored if the '`--high-level-code`' option is enabled.

`--asm-labels` (grades: asm_fast)
`--no-asm-labels` (grades: none, reg)
: Specify whether or not to use GNU C's asm extensions for inline assembler labels. This option is ignored if the '`--high-level-code`' option is enabled.

`--stack-segments` (grades: any grade containing '.stseg')
: Specify whether use dynamically sized stacks that are composed of small segments. This can help to avoid stack exhaustion at the cost of increased execution time. This option is not supported by the '`--high-level-code`' back-ends.

9.8.4 MLDS back-end compilation model option

`-H`, `--high-level-code` (grades: hlc, csharp, java)
: Use an alternative back-end that generates high-level code rather than the very low-level code that is generated by our original back-end.

9.8.5 Optional features compilation model options

`--debug` (grades: any grade containing '`.debug`')
: Enables the inclusion in the executable of code and data structures that allow the program to be debugged with '`mdb`' (see Chapter 7 [Debugging], page 16). This option is only supported by the low-level C back-end.

`--decl-debug` (grades: any grade containing '`.decldebug`')
: Enables the inclusion in the executable of code and data structures that allow subterm dependency tracking in the declarative debugger. This option is only supported by the low-level C back-end.

`--profiling`, `--time-profiling` (grades: any grade containing '`.prof`')
: Enable time profiling. Insert profiling hooks in the generated code, and also output some profiling information (the static call graph) to the file '*`module`*`.prof`'. See Chapter 8 [Profiling], page 80. This option is only supported by the C back-ends.

`--memory-profiling` (grades: any grade containing '`.memprof`')
: Enable memory profiling. Insert memory profiling hooks in the generated code, and also output some profiling information (the static call graph) to the file '*`module`*`.prof`'. See Section 8.5 [Using mprof for profiling memory allocation], page 85. See Section 8.6 [Using mprof for profiling memory retention], page 86. This option is only supported by the C back-ends.

`--deep-profiling` (grades: any grade containing '`.profdeep`')
: Enable deep profiling by inserting the appropriate hooks in the generated code. This option is only supported by the low-level C back-end.

`--no-coverage-profiling`
: Disable coverage profiling. Coverage profiling is part of the deep profiler and only used in deep profiling grades, it inserts coverage points that are used to measure how execution moves through a procedure.

`--profile-for-feedback`
: Enable options in the deep profiler that generate suitable profiles for use with profile directed feedback analysis. Currently the only feedback analysis is used for automatic parallelism. This option only effects programs built in deep profiling grades.

`--experimental-complexity` *`filename`*
: Enable experimental complexity analysis for the predicates listed in the given file. This option is supported for the C back-end, with '`--no-high-level-code`'. For now, this option itself is for developers only.

`--gc {none, boehm, automatic}`
`--garbage-collection {none, boehm, automatic}`
: Specify which method of garbage collection to use. Grades containing '`csharp`' or '`java`' use '`--gc automatic`', grades containing '`.gc`' use '`--gc boehm`', other grades use '`--gc none`'. '`conservative`' or '`boehm`' is Hans Boehm et al's conservative garbage collector. '`automatic`' means the target language provides it. This is the case for the C# and Java back-ends, which always use the underlying implementation's garbage collector.

`--use-trail` (grades: any grade containing '`.tr`' or '`.trseg`')
: Enable use of a trail. This is necessary for interfacing with constraint solvers, or for backtrackable destructive update. This option is only supported by the C back-ends.

`--trail-segments` (grades: any grade containing '`.trseg`')
: This option is deprecated as trailing now always uses trail segments. The '`.trseg`' grade modifier is a synonym for '`.tr`'.

`--parallel`
: In low-level C grades this enables support for parallel execution. Parallel execution can be achieved by using either the parallel conjunction operator or the concurrency support in the '`thread`' module of the standard library. See Section "Goals" in *The Mercury Language Reference Manual*, and See Section "thread" in *The Mercury Library Reference Manual*.

  In high-level C grades this enables support for concurrency, which is accessible via the '`thread`' module in the standard library. The runtime uses POSIX threads to achieve this, therefore it may also support parallel execution of threads.

  The Java and C# grades support concurrency without this option. Parallel execution may also be available depending on the target's runtime.

`--threadscope`
: Enable support for threadscope profiling. This enables runtime support for profiling the parallel execution of programs, See Section 8.8 [Using threadscope], page 88.

`--maybe-thread-safe {yes, no}`
: Specify how to treat the '`maybe_thread_safe`' foreign code attribute. '`yes`' means that a foreign procedure with the '`maybe_thread_safe`' option is treated as thought is has a '`thread_safe`' attribute. '`no`' means that the foreign procedure is treated as though it has a '`not_thread_safe`' attribute. The default is '`no`'.

`--single-prec-float` (grades: any grade containing '`.spf`')
: Use single precision floats so that, on 32-bit machines floating point values don't need to be boxed. Double precision floats are used by default. This option is not supported for the C# or Java back-ends.

9.8.6 Developer compilation model options

Of the options listed below, the '`--num-tag-bits`' option may be useful for cross-compilation, but apart from that these options are all experimental and are intended for use by developers of the Mercury implementation rather than by ordinary Mercury programmers.

`--num-ptag-bits` *n*
`--num-tag-bits` *n*
: (This option is not intended for general use.)
  Use *n* primary tag bits. The default number of primary tag bits to use is determined by the auto-configuration script; the only valid reason for overriding this default is cross-compilation to an architecture where auto-configuration would yield a different value.

`--no-type-layout`
: (This option is not intended for general use.)
Don't output base_type_layout structures or references to them. This option will generate smaller executables, but will not allow the use of code that uses the layout information (e.g. '`functor`', '`arg`'). Using such code will result in undefined behaviour at runtime. The C code also needs to be compiled with '`-DNO_TYPE_LAYOUT`'.

9.9 Code generation options

`--low-level-debug`
: Enables various low-level debugging stuff that was in the distant past used to debug the Mercury compiler's low-level code generation. This option is not likely to be useful to anyone except the Mercury implementors. It causes the generated code to become very big and very inefficient, and slows down compilation a lot.

`--no-trad-passes`
: The default '`--trad-passes`' completely processes each predicate before going on to the next predicate. This option tells the compiler to complete each phase of code generation on all predicates before going on the next phase on all predicates.

`--no-reclaim-heap-on-nondet-failure`
: Don't reclaim heap on backtracking in nondet code.

`--no-reclaim-heap-on-semidet-failure`
: Don't reclaim heap on backtracking in semidet code.

`--no-reclaim-heap-on-failure`
: Combines the effect of the two options above.

`--max-jump-table-size` *n*
: The maximum number of entries a jump table can have. The special value 0 indicates the table size is unlimited. This option can be useful to avoid exceeding fixed limits imposed by some C compilers.

`--fact-table-max-array-size` *size*
: Specify the maximum number of elements in a single '`pragma fact_table`' data array (default: 1024). The data for fact tables is placed into multiple C arrays, each with a maximum size given by this option. The reason for doing this is that most C compilers have trouble compiling very large arrays.

`--fact-table-hash-percent-full` *percentage*
: Specify how full the '`pragma fact_table`' hash tables should be allowed to get. Given as an integer percentage (valid range: 1 to 100, default: 90). A lower

value means that the compiler will use larger tables, but there will generally be less hash collisions, so it may result in faster lookups.

9.9.1 Code generation target options

The following options allow the Mercury compiler to optimize the generated C code based on the characteristics of the expected target architecture. The default values of these options will be whatever is appropriate for the host architecture that the Mercury compiler was installed on, so normally there is no need to set these options manually. They might come in handy if you are cross-compiling. But even when cross-compiling, it is probably not worth bothering to set these unless efficiency is absolutely paramount.

`--have-delay-slot`
: (This option is not intended for general use.)
Assume that branch instructions have a delay slot.

`--num-real-r-regs` *n*
: (This option is not intended for general use.)
Assume r1 up to r*n* are real general purpose registers.

`--num-real-f-regs` *n*
: (This option is not intended for general use.)
Assume f1 up to f*n* are real floating point registers.

`--num-real-r-temps` *n*
: (This option is not intended for general use.)
Assume that *n* non-float temporaries will fit into real machine registers.

`--num-real-f-temps` *n*
: (This option is not intended for general use.)
Assume that *n* float temporaries will fit into real machine registers.

9.10 Optimization options

9.10.1 Overall optimization options

`-O` *n*
`--opt-level` *n*
`--optimization-level` *n*
: Set optimization level to *n*. Optimization levels range from -1 to 6. Optimization level -1 disables all optimizations, while optimization level 6 enables all optimizations except for the cross-module optimizations listed below. Some experimental optimizations (for example constraint propagation) are not be enabled at any optimization level.

In general, there is a trade-off between compilation speed and the speed of the generated code. When developing, you should normally use optimization level 0, which aims to minimize compilation time. It enables only those optimizations that in fact usually *reduce* compilation time. The default optimization

level is level 2, which delivers reasonably good optimization in reasonable time. Optimization levels higher than that give better optimization, but take longer, and are subject to the law of diminishing returns. The difference in the quality of the generated code between optimization level 5 and optimization level 6 is very small, but using level 6 may increase compilation time and memory requirements dramatically.

Note that if you want the compiler to perform cross-module optimizations, then you must enable them separately; the cross-module optimizations are not enabled by any '`-O`' level, because they affect the compilation process in ways that require special treatment by '`mmake`'.

`--opt-space`
`--optimize-space`
Turn on optimizations that reduce code size and turn off optimizations that significantly increase code size.

`--intermod-opt`
`--intermodule-optimization`
Perform inlining and higher-order specialization of the code for predicates or functions imported from other modules.

`--trans-intermod-opt`
`--transitive-intermodule-optimization`
Use the information stored in '*`module`*`.trans_opt`' files to make intermodule optimizations. The '*`module`*`.trans_opt`' files are different to the '*`module`*`.opt`' files as '`.trans_opt`' files may depend on other '`.trans_opt`' files, whereas each '`.opt`' file may only depend on the corresponding '`.m`' file. Note that '`--transitive-intermodule-optimization`' does not work with '`mmc --make`'.

`--no-read-opt-files-transitively`
Only read the inter-module optimization information for directly imported modules, not the transitive closure of the imports.

`--use-opt-files`
Perform inter-module optimization using any '`.opt`' files which are already built, e.g. those for the standard library, but do not build any others.

`--use-trans-opt-files`
Perform inter-module optimization using any '`.trans_opt`' files which are already built, e.g. those for the standard library, but do not build any others.

`--intermodule-analysis`
Perform analyses such as termination analysis and unused argument elimination across module boundaries. This option is not yet fully implemented.

`--analysis-repeat` *n*
: The maximum number of times to repeat analyses of suboptimal modules with '`--intermodule-analyses`' (default: 0). This option only works with '`mmc --make`'.

9.10.2 High-level (HLDS -> HLDS) optimization options

These optimizations are high-level transformations on our HLDS (high-level data structure).

`--no-inlining`
: Disable all forms of inlining.

`--no-inline-simple`
: Disable the inlining of simple procedures.

`--no-inline-builtins`
: Generate builtins (e.g. arithmetic operators) as calls to out-of-line procedures. This is done by default when debugging, as without this option the execution of builtins is not traced.

`--no-inline-single-use`
: Disable the inlining of procedures called only once.

`--inline-compound-threshold` *threshold*
: Inline a procedure if its size (measured roughly in terms of the number of connectives in its internal form), multiplied by the number of times it is called, is below the given threshold.

`--inline-simple-threshold` *threshold*
: Inline a procedure if its size is less than the given threshold.

`--intermod-inline-simple-threshold` *threshold*
: Similar to –inline-simple-threshold, except used to determine which predicates should be included in '`.opt`' files. Note that changing this between writing the '`.opt`' file and compiling to C may cause link errors, and too high a value may result in reduced performance.

`--inline-vars-threshold` *threshold*
: Don't inline a call if it would result in a procedure containing more than *threshold* variables. Procedures containing large numbers of variables can cause slow compilation.

`--inline-linear-tail-rec-sccs`
: Given a set of mutually recursive procedures (an SCC, or strongly connected component, of the call graph) in which each procedure contains exactly tail call to a procedure in the SCC, so that the tail recursive calls form a linear chain through the SCC, inline the callee at every one of those mutually tail recursive call sites. This converts mutual tail recursion into self tail recursion, which the MLDS backend can turn into code that runs in constant stack space.

`--loop-invariants`
: Optimize loop invariants by moving computations within a loop that are the same on every iteration to the outside so they are only calculated once.

`--no-common-struct`
: Disable optimization of common term structures.

`--constraint-propagation`
: Enable the constraint propagation transformation, which attempts to transform the code so that goals which can fail are executed as early as possible.

`--local-constraint-propagation`
: Enable the constraint propagation transformation, but only rearrange goals within each procedure. Specialized versions of procedures will not be created.

`--no-follow-code`
: Don't migrate builtin goals into branched goals.

`--optimize-unused-args`
: Remove unused predicate arguments. The compiler will generate more efficient code for polymorphic predicates.

`--intermod-unused-args`
: Perform unused argument removal across module boundaries. This option implies '`--optimize-unused-args`' and '`--intermodule-optimization`'.

`--unneeded-code`
: Remove goals from computation paths where their outputs are not needed, provided the language semantics options allow the deletion or movement of the goal.

`--unneeded-code-copy-limit` *limit*
: Gives the maximum number of places to which a goal may be copied when removing it from computation paths on which its outputs are not needed. A value of zero forbids goal movement and allows only goal deletion; a value of one prevents any increase in the size of the code.

`--optimize-higher-order`
: Specialize calls to higher-order predicates where the higher-order arguments are known.

`--type-specialization`
: Specialize calls to polymorphic predicates where the polymorphic types are known.

`--user-guided-type-specialization`
: Enable specialization of polymorphic predicates for which there are '`pragma type_spec`' declarations. See the "Type specialization" section in the "Pragmas" chapter of the Mercury Language Reference Manual for more details.

`--higher-order-size-limit` *limit*
: Set the maximum goal size of specialized versions created by '`--optimize-higher-order`' and '`--type-specialization`'. Goal size is measured as the number of calls, unifications and branched goals.

`--higher-order-arg-limit` *limit*
: Set the maximum size of higher-order arguments to be specialized by '`--optimize-higher-order`' and '`--type-specialization`'.

`--optimize-constant-propagation`
: Evaluate constant expressions at compile time.

`--introduce-accumulators`
: Attempt to introduce accumulating variables into procedures, so as to make the procedure tail recursive.

`--optimize-constructor-last-call`
: Enable the optimization of "last" calls that are followed by constructor application.

`--optimize-dead-procs`
: Enable dead procedure elimination.

`--excess-assign`
: Remove excess assignment unifications.

`--no-optimize-format-calls`
: Do not attempt to interpret the format string in calls to string.format and related predicates at compile time; always leave this to be done at runtime.

`--optimize-duplicate-calls`
: Optimize away multiple calls to a predicate with the same input arguments.

`--delay-constructs`
: Reorder goals to move construction unifications after primitive goals that can fail.

`--optimize-saved-vars`
: Minimize the number of variables that have to be saved across calls.

`--deforestation`
: Enable deforestation. Deforestation is a program transformation whose aim is to avoid the construction of intermediate data structures and to avoid repeated traversals over data structures within a conjunction.

`--deforestation-depth-limit` *limit*
: Specify a depth limit to prevent infinite loops in the deforestation algorithm. A value of -1 specifies no depth limit. The default is 4.

`--deforestation-vars-threshold` *threshold*
: Specify a rough limit on the number of variables in a procedure created by deforestation. A value of -1 specifies no limit. The default is 200.

`--deforestation-size-threshold` *threshold*
: Specify a rough limit on the size of a goal to be optimized by deforestation. A value of -1 specifies no limit. The default is 15.

`--analyse-exceptions`
: Try to identify those procedures that cannot throw an exception. This information can be used by some optimization passes.

`--analyse-trail-usage`
: Enable trail usage analysis. Identify those procedures that will not modify the trail. This information is used to reduce the overhead of trailing.

`--analyse-mm-tabling`
: Identify those goals that do not calls procedures that are evaluated using minimal model tabling. This information is used to reduce the overhead of minimal model tabling.

9.10.3 MLDS back-end (MLDS -> MLDS) optimization options

These optimizations are applied to the medium level intermediate code.

`--no-mlds-optimize`
: Disable the MLDS -> MLDS optimization passes.

`--no-mlds-peephole`
: Do not perform peephole optimization of the MLDS.

`--no-optimize-tailcalls`
: Treat tailcalls as ordinary calls rather than optimizing by turning self-tailcalls into loops.

`--no-optimize-initializations`
: Leave initializations of local variables as assignment statements, rather than converting such assignments statements into initializers.

`--eliminate-local-variables`
: Eliminate local variables with known values, where possible, by replacing occurrences of such variables with their values.

`--no-generate-trail-ops-inline`
: Do not generate trailing operations inline, but instead insert calls to the versions of these operations in the standard library.

9.10.4 Medium-level (HLDS -> LLDS) optimization options

These optimizations are applied during the process of generating low-level intermediate code from our high-level data structure.

`--no-static-ground-terms`
: Disable the optimization of constructing constant ground terms at compile time and storing them as static constants. Note that auxiliary data structures created by the compiler for purposes such as debugging will still be created as static constants.

`--no-smart-indexing`
: Generate switches as simple if-then-else chains; disable string hashing and integer table-lookup indexing.

`--dense-switch-req-density` *`percentage`*
: The jump table generated for an atomic switch must have at least this percentage of full slots (default: 25).

`--dense-switch-size` *`size`*
: The jump table generated for an atomic switch must have at least this many entries (default: 4).

`--lookup-switch-req-density` *`percentage`*
: The lookup tables generated for an atomic switch in which all the outputs are constant terms must have at least this percentage of full slots (default: 25).

`--lookup-switch-size` *`size`*
: The lookup tables generated for an atomic switch in which all the outputs are constant terms must have at least this many entries (default: 4).

`--string-trie-switch-size` *`size`*
: The trie generated for a string switch must have at least this many entries (default: 16).

`--string-hash-switch-size` *`size`*
: The hash table generated for a string switch must have at least this many entries (default: 8).

`--string-binary-switch-size` *`size`*
: The binary search table generated for a string switch must have at least this many entries (default: 4).

`--tag-switch-size` *`size`*
: The number of alternatives in a tag switch must be at least this number (default: 3).

`--try-switch-size` *size*
: The number of alternatives in a try-chain switch must be at least this number (default: 3).

`--binary-switch-size` *size*
: The number of alternatives in a binary search switch must be at least this number (default: 4).

`--no-use-atomic-cells`
: Don't use the atomic variants of the Boehm gc allocator calls, even when this would otherwise be possible.

`--no-middle-rec`
: Disable the middle recursion optimization.

`--no-simple-neg`
: Don't generate simplified code for simple negations.

9.10.5 Low-level (LLDS -> LLDS) optimization options

These optimizations are transformations that are applied to our low-level intermediate code before emitting C code.

`--no-common-data`
: Disable optimization of common data structures.

`--no-layout-common-data`
: Disable optimization of common subsequences in layout structures.

`--no-llds-optimize`
: Disable the low-level optimization passes.

`--no-optimize-peep`
: Disable local peephole optimizations.

`--no-optimize-jumps`
: Disable elimination of jumps to jumps.

`--no-optimize-fulljumps`
: Disable elimination of jumps to ordinary code.

`--pessimize-tailcalls`
: Disable the optimization of tailcalls.

`--checked-nondet-tailcalls`
: Convert nondet calls into tail calls whenever possible, even when this requires a runtime check. This option tries to minimize stack consumption, possibly at the expense of speed.

`--no-use-local-vars`
: Disable the transformation to use local variables in C code blocks wherever possible.

`--no-optimize-labels`
: Disable elimination of dead labels and code.

`--optimize-dups`
: Enable elimination of duplicate code within procedures.

`--optimize-proc-dups`
: Enable elimination of duplicate procedures.

`--no-optimize-frames`
: Disable stack frame optimizations.

`--no-optimize-delay-slot`
: Disable branch delay slot optimizations.

`--optimize-reassign`
: Optimize away assignments to locations that already hold the assigned value.

`--optimize-repeat` *n*
: Iterate most optimizations at most *n* times (default: 3).

`--layout-compression-limit` *n*
: Attempt to compress the layout structures used by the debugger only as long as the arrays involved have at most *n* elements (default: 4000).

9.10.6 Output-level (LLDS -> C) optimization options

These optimizations are applied during the process of generating C intermediate code from our low-level data structure.

`--no-emit-c-loops`
: Use only gotos — don't emit C loop constructs.

`--use-macro-for-redo-fail`
: Emit the fail or redo macro instead of a branch to the fail or redo code in the runtime system.

`--procs-per-c-function` *n*
: Don't put the code for more than *n* Mercury procedures in a single C function. The default value of *n* is one. Increasing *n* can produce slightly more efficient code, but makes compilation slower. Setting *n* to the special value zero has the effect of putting all the procedures in a single function, which produces the most efficient code but tends to severely stress the C compiler.

`--no-local-thread-engine-base`
: Do not copy the thread-local Mercury engine base address into local variables. This option only affects low-level parallel grades not using the GNU C global register variables extension.

9.11 Target code compilation options

If you are using Mmake, you need to pass these options to the target code compiler (e.g. '`mgnuc`') rather than to '`mmc`'.

`--target-debug`
: Enable debugging of the generated target code. If the target language is C, this has the same effect as '`--c-debug`' (see below). If the target language is C#, this causes the compiler to pass '`/debug`' to the C# compiler.

`--cc` *compiler-name*
: Specify which C compiler to use.

`--c-include-directory` *dir*
`--c-include-dir` *dir*
: Append *dir* to the list of directories to be searched for C header files. Note that if you want to override this list, rather than append to it, then you can set the `MERCURY_MC_ALL_C_INCL_DIRS` environment variable to a sequence of '`--c-include-directory`' options.

`--c-debug`
: Pass the '`-g`' flag to the C compiler, to enable debugging of the generated C code, and also disable stripping of C debugging information from the executable. Since the generated C code is very low-level, this option is not likely to be useful to anyone except the Mercury implementors, except perhaps for debugging code that uses Mercury's C interface extensively.

`--no-c-optimize`
: Don't enable the C compiler's optimizations.

`--no-ansi-c`
: Don't specify to the C compiler that the ANSI dialect of C should be used. Use the full contents of system headers, rather than the ANSI subset.

`--inline-alloc`
: Inline calls to '`GC_malloc()`'. This can improve performance a fair bit, but may significantly increase code size. This option has no effect if '`--gc boehm`' is not set or if the C compiler is not GNU C.

`--cflags` *options*
`--cflag` *option*
: Specify options to be passed to the C compiler. '`--cflag`' should be used for single words which need to be quoted when passed to the shell.

`--javac` *compiler-name*
`--java-compiler` *compiler-name*
: Specify which Java compiler to use. The default is '`javac`'.

`--java-interpreter` *interpreter-name*
: Specify which Java interpreter to use. The default is '`java`'.

`--java-flags` *options*
`--java-flag` *option*
: Specify options to be passed to the Java compiler. '`--java-flag`' should be used for single words which need to be quoted when passed to the shell.

`--java-classpath` *dir*
: Set the classpath for the Java compiler.

`--java-object-file-extension` *extension*
: Specify an extension for Java object (bytecode) files. By default this is '`.class`'.

`--csharp-compiler` *compiler-name*
: Specify which C# compiler to use. The default is '`csc`'.

`--csharp-flags` *options*
`--csharp-flag` *option*
: Specify options to be passed to the C# compiler. '`--csharp-flag`' should be used for single words which need to be quoted when passed to the shell.

`--cil-interpreter` *interpreter-name*
: Specify the program that implements the Common Language Infrastructure (CLI) execution environment, e.g. '`mono`'.

9.12 Link options

`-o` *filename*
`--output-file` *filename*
: Specify the name of the final executable. (The default executable name is the same as the name of the first module on the command line, but without the '`.m`' extension.) This option is ignored by '`mmc --make`'.

`--ld-flags` *options*
`--ld-flags` *option*
: Specify options to be passed to the command invoked by '`mmc`' to link an executable. Use `mmc --output-link-command` to find out which command is used. '`--ld-flag`' should be used for single words which need to be quoted when passed to the shell.

`--ld-libflags` *options*
`--ld-libflag` *option*
: Specify options to be passed to the command invoked by '`mmc`' to link a shared library. Use `mmc --output-shared-lib-link-command` to find out which command is used. '`--ld-libflag`' should be used for single words which need to be quoted when passed to the shell.

`-L` *directory*
`--library-directory` *directory*
: Append *directory* to the list of directories in which to search for libraries.

`-R` *directory*
`--runtime-library-directory` *directory*
: Append *directory* to the list of directories in which to search for shared libraries at runtime.

`--no-default-runtime-library-directory`
: Do not add any directories to the runtime search path automatically.

`--shlib-linker-install-name-path` *directory*
: Specify the path where a shared library will be installed. This option is useful on systems where the runtime search path is obtained from the shared library and not via the -R option (such as Mac OS X).

`-l` *library*
`--library` *library*
: Link with the specified library.

`--link-object` *file*
: Link with the specified object file or archive of object files.

`--search-lib-files-dir` *directory*
`--search-library-files-directory` *directory*
: Search *directory* for Mercury library files have not yet been installed. Similar to adding *directory* using all of the '`--search-directory`', '`--intermod-directory`', '`--library-directory`', '`--init-file-directory`', and '`--c-include-directory`' options, but does the right thing when '`--use-subdirs`' or '`--use-grade-subdirs`' options are used.

`--mld` *directory*
`--mercury-library-directory` *directory*
: Append *directory* to the list of directories to be searched for Mercury libraries. This will add '`--search-directory`', '`--library-directory`', '`--init-file-directory`' and '`--c-include-directory`' options as needed. See Section 6.2.2 [Using installed libraries with mmc –make], page 12.

`--ml` *library*
`--mercury-library` *library*
: Link with the specified Mercury library. See Section 6.2.2 [Using installed libraries with mmc –make], page 12.

`--mercury-standard-library-directory` *directory*
`--mercury-stdlib-dir` *directory*
: Search *directory* for the Mercury standard library. Implies '`--mercury-library-directory` *directory*' and '`--mercury-configuration-directory` *directory*'.

`--no-mercury-standard-library-directory`
`--no-mercury-stdlib-dir`
: Don't use the Mercury standard library. Implies '`--no-mercury-configuration-directory`'.

`--init-file-directory` *directory*
: Append *directory* to the list of directories to be searched for '`.init`' files by '`c2init`'.

`--init-file` *file*
: Append *file* to the list of '`.init`' files to be passed to '`c2init`'.

`--trace-init-file` *file*
: Append *file* to the list of '`.init`' files to be passed to '`c2init`' when tracing is enabled.

`--linkage {shared,static}`
: Specify whether to use shared or static linking for executables. Shared libraries are always linked with '`--linkage shared`'.

`--mercury-linkage {shared,static}`
: Specify whether to use shared or static linking when linking an executable with Mercury libraries. Shared libraries are always linked with '`--mercury-linkage shared`'.

`--no-demangle`
: Don't pipe link errors through the Mercury demangler.

`--no-main`
: Don't generate a C main() function. The user's code must provide a main() function.

`--no-allow-undefined`
: Do not allow undefined symbols in shared libraries.

`--no-use-readline`
: Disable use of the readline library in the debugger.

`--runtime-flags` *`flags`*
: Specify flags to pass to the Mercury runtime.

`--extra-initialization-functions`
`--extra-inits`
: Search '`.c`' files for extra initialization functions. (This may be necessary if the C files contain hand-coded C code with '`INIT`' comments, rather than containing only C code that was automatically generated by the Mercury compiler.)

`--link-executable-command` *`command`*
: Specify the command used to invoke the linker when linking an executable.

`--link-shared-lib-command` *`command`*
: Specify the command used to invoke the linker when linking a shared library.

`--no-strip`
: Do not strip executables.

`--strip-executable-command` *`command`*
: Specify the command used to strip executables if no linker flag to do so is available. This option has no effect on '`ml`'.

`--strip-executable-shared-flags` *`options`*
: Specify options to pass to the strip executable command when linking against Mercury shared libraries.

`--stripe-executable-static-flags` *`options`*
: Specify options to pass to the strip executable command when linking against Mercury static libraries.

`--java-archive-command` *`command`*
: Specify the command used to produce Java archive (JAR) files.

`--framework` *`framework`*
: Build and link against the specified framework. (Mac OS X only.)

`-F` *directory*
`--framework-directory` *directory*
: Append the specified directory to the framework search path. (Mac OS X only.)

`--sign-assembly` *keyfile*
: Sign the current assembly with the strong name contained in the specified key file. (This option is only meaningful when generating library assemblies with the C# back-end.)

`--cstack-reserve-size` *size*
: Set the total size of the C stack in virtual memory for executables. The stack size is given in bytes. This option is only supported (and indeed only necessary) on systems running Microsoft Windows.

9.13 Build system options

`-m`
`--make`
: Treat the non-option arguments to '`mmc`' as files to make, rather than source files. Create the specified files, if they are not already up-to-date. (Note that this option also enables '`--use-subdirs`'.)

`-r`
`--rebuild`
: Same as '`--make`', but always rebuild the target files even if they are up to date.

`-k`
`--keep-going`
: With '`--make`' keep going as far as possible even if an error is detected.

`-j` *n*
`--jobs` *n*
: With '`--make`', attempt to perform up to *n* jobs concurrently for some tasks.

`--track-flags`
: With '`--make`', keep track of the options used when compiling each module. If an option for a module is added or removed, '`mmc --make`' will then know to recompile the module even if the timestamp on the file itself has not changed. Warning, verbosity and build system options are not tracked.

`--pre-link-command` *command*
: Specify a command to run before linking with '`mmc --make`'. This can be used to compile C source files which rely on header files generated by the Mercury compiler. The command will be passed the names of all of the source files in the program or library, with the source file containing the main module given first.

`--extra-init-command` *command*
: Specify a command to produce extra entries in the '`.init`' file for a library. The command will be passed the names of all of the source files in the program or library, with the source file containing the main module given first.

`--install-prefix` *dir*
: Specify the directory under which to install Mercury libraries.

`--install-command` *command*
: Specify the command to use to install the files in Mercury libraries. The given command will be invoked as *`command source target`* to install each file in a Mercury library. The default command is '`cp`'.

`--install-command-dir-option` *option*
: Specify the flag to pass to the install command to install a directory. The given command will be invoked as *`command option source target`* to install each directory. The default option is '`-R`'.

`--no-detect-libgrades`
: Do not scan the installation directory to determine which standard library grades are available.

`--libgrade` *grade*
: Add *grade* to the list of compilation grades in which a library to be installed should be built.

`--no-libgrade`
: Clear the list of compilation grades in which a library to be installed should be built. The main use of this is to avoid building and installing the default set of grades.

`--libgrades-include-component` *component*
`--libgrades-include` *component*
: Remove grades that do not contain the specified component from the set of library grades to be installed. (This option does not work with Mmake, only '`mmc --make`'.)

`--libgrades-exclude-component` *component*
`--libgrades-exclude` *component*
: Remove grades that contain the specified component from the set of library grades to be installed. (This option does not work with Mmake, only '`mmc --make`'.)

`--lib-linkage {shared,static}`
: Specify whether libraries should be installed for shared or static linking. This option can be specified multiple times. By default libraries will be installed for both shared and static linking.

`--flags` *file*
`--flags-file` *file*
: Take options from the specified file, and handle them as if they were specified on the command line.

`--options-file` *file*
: Add *file* to the list of options files to be processed. If *file* is '`-`', an options file will be read from the standard input. By default the file '`Mercury.options`' in the current directory will be read. See Chapter 5 [Using Mmake], page 5 for a description of the syntax of options files.

`--config-file` *file*
: Read the Mercury compiler's configuration information from *file*. If the '`--config-file`' option is not set, a default configuration will be used, unless '`--no-mercury-stdlib-dir`' is passed to '`mmc`'. The configuration file is just an options file (see Chapter 5 [Using Mmake], page 5).

`--options-search-directory` *dir*
: Add *dir* to the list of directories to be searched for options files.

`--mercury-configuration-directory` *dir*
`--mercury-config-dir` *dir*
: Search *dir* for Mercury system's configuration files.

`-I` *dir*
`--search-directory` *dir*
: Append *dir* to the list of directories to be searched for imported modules.

`--intermod-directory` *dir*
: Append *dir* to the list of directories to be searched for '`.opt`' files.

`--use-search-directories-for-intermod`
: Append the arguments of all -I options to the list of directories to be searched for '`.opt`' files.

`--no-libgrade-install-check`
: Do not check that libraries have been installed before attempting to use them. (This option is only meaningful with '`mmc --make`'.)

`--use-subdirs`
: Create intermediate files in a '`Mercury`' subdirectory, rather than in the current directory.

`--use-grade-subdirs`
: Generate intermediate files in a '`Mercury`' subdirectory, laid out so that multiple grades can be built simultaneously. Executables and libraries will be symlinked or copied into the current directory. '`--use-grade-subdirs`' does not work with Mmake (it does work with '`mmc --make`').

`--order-make-by-timestamp`
: Make '`mmc --make`' compile more recently modified source files first.

`--show-make-times`
: Report run times for commands executed by '`mmc --make`'.

`--extra-library-header` *file*
`--extra-lib-header` *file*
: Install the specified C header file along with a Mercury library. (This option is only supported by '`mmc --make`')

`--restricted-command-line`
: Enable this option if your shell doesn't support long command lines. This option uses temporary files to pass arguments to sub-commands. (This option is only supported by '`mmc --make`')

`--env-type` *type*
: Specify the environment type in which the compiler and generated programs will be invoked. The environment type controls how the compiler and generated programs interact with the shell and other system tools. The *type* should be one of '`posix`', '`cygwin`', '`msys`', or '`windows`'. This option is equivalent to setting all of '`--host-env-type`', '`--system-env-type`' and '`--target-env-type`' to *type*.

`--host-env-type` *type*
: Specify the environment type in which the compiler will be invoked. (See above for a list of supported environment types.)

`--system-env-type` *type*
: Specify the environment type in which external programs invoked by the compiler will run. If not specified, this defaults to the value given by '`--host-env-type`'.

`--target-env-type` *type*
: Specify the environment type in which generated programs will be invoked. (See above for a list of supported environment types.)

9.14 Miscellaneous options

`-?`
`-h`
`--help` Print a usage message.

`--filenames-from-stdin`
: Read then compile a newline terminated module name or file name from the standard input. Repeat this until EOF is reached. (This allows a program or user to interactively compile several modules without the overhead of process creation for each one.)

`--typecheck-ambiguity-warn-limit` *n*
: Set the number of type assignments required to generate a warning about highly ambiguous overloading to *n*.

`--typecheck-ambiguity-error-limit` *n*
: Set the number of type assignments required to generate an error excessively ambiguous overloading to *n*. If this limit is reached, the typechecker will not process the predicate or function any further.

`--control-granularity`
: Don't try to generate more parallelism than the machine can handle, which may be specified at runtime or detected automatically. (see the '`-P`' option in the `MERCURY_OPTIONS` environment variable.)

`--distance-granularity` *distance_value*
: Control the granularity of parallel execution using the specified distance value.

`--implicit-parallelism`
: Introduce parallel conjunctions where it could be worthwhile (implicit parallelism) using deep profiling feedback information generated by mdprof_create_feedback. The profiling feedback file can be specified using the '`--feedback-file`' option.

`--feedback-file` *file*
: Use the specified profiling feedback file which may currently only be processed for automatic parallelism.

10 Environment variables

The shell scripts in the Mercury compilation environment will use the following environment variables if they are set. There should be little need to use these, because the default values will generally work fine.

`MERCURY_DEFAULT_GRADE`
: The default grade to use if no '`--grade`' option is specified.

`MERCURY_STDLIB_DIR`
: The directory containing the installed Mercury standard library. '`--mercury-stdlib-dir`' options passed to the '`mmc`', '`ml`', '`mgnuc`' and '`c2init`' scripts override the setting of the `MERCURY_STDLIB_DIR` environment variable.

`MERCURY_OPTIONS`
: A list of options for the Mercury runtime system, which gets linked into every Mercury program. The options given in this environment variable apply to every program; the options given in an environment variable whose name is of the form `MERCURY_OPTIONS_`*`progname`* apply only to programs named *progname*. Options may also be set for a particular executable at compile time by passing '`--runtime-flags`' options to the invocations of '`ml`' and '`c2init`' which create that executable. These options are processed first, followed by those in `MERCURY_OPTIONS`, with the options in `MERCURY_OPTIONS_`*`progname`* being processed last.

  The Mercury runtime system accepts the following options.

  `-C` *size*
  : Tells the runtime system to optimize the locations of the starts of the various data areas for a primary data cache of *size* kilobytes. The optimization consists of arranging the starts of the areas to differ as much as possible modulo this size.

  `-D` *debugger*
  : Enables execution tracing of the program, via the internal debugger if *debugger* is '`i`' and via the external debugger if *debugger* is '`e`'. (The mdb script works by including '`-Di`' in MERCURY_OPTIONS.) The external debugger is not yet available.

  `-p`
  : Disables profiling. This only has an effect if the executable was built in a profiling grade.

  `-P` *num*
  : Tells the runtime system to create *num* threads for executing Mercury code if the program was built in a parallel low-level C grade. The Mercury runtime attempts to automatically determine this value if support is available from the operating system. If it cannot or support is unavailable it defaults to '`1`'.

`--max-engines` *num*
: Tells the runtime system to allow a maximum of *num* POSIX threads, each with its own Mercury engine. This only affects programs in low-level C parallel grades.

`--max-contexts-per-thread` *num*
: Tells the runtime system to create at most *num* contexts per POSIX thread for parallel execution. Each context created requires a set of stacks, setting this value too high can consume excess memory. This only has an effect if the executable was built in a low-level C parallel grade.

`--thread-pinning`
: Request that the runtime system attempts to pin Mercury engines (POSIX threads) to CPU cores/hardware threads. This only has an effect if the executable was built in a parallel low-level C grade. This is disabled by default but may be enabled by default in the future.

`-T` *time-method*
: If the executable was compiled in a grade that includes time profiling, this option specifies what time is counted in the profile. *time-method* must have one of the following values:

  `'r'` Profile real (elapsed) time (using ITIMER_REAL).

  `'p'` Profile user time plus system time (using ITIMER_PROF). This is the default.

  `'v'` Profile user time (using ITIMER_VIRTUAL).

  Currently, only the '`-Tr`' option works on Cygwin; on that platform it is the default.

`--heap-size` *size*
: Sets the size of the heap to *size* kilobytes.

`--heap-size-kwords` *size*
: Sets the size of the heap to *size* kilobytes multiplied by the word size in bytes.

`--detstack-size` *size*
: Sets the size of the det stack to *size* kilobytes.

`--detstack-size-kwords` *size*
: Sets the size of the det stack to *size* kilobytes multiplied by the word size in bytes.

`--nondetstack-size` *size*
: Sets the size of the nondet stack to *size* kilobytes.

`--nondetstack-size-kwords` *size*
: Sets the size of the nondet stack to *size* kilobytes multiplied by the word size in bytes.

`--small-detstack-size` *size*
: Sets the size of the det stack used for executing parallel conjunctions to *size* kilobytes. The regular det stack size must be equal or greater.

`--small-detstack-size-kwords` *size*
: Sets the size of the det stack used for executing parallel conjunctions to *size* kilobytes multiplied by the word size in bytes. The regular det stack size must be equal or greater.

`--small-nondetstack-size` *size*
: Sets the size of the nondet stack for executing parallel computations to *size* kilobytes. The regular nondet stack size must be equal or greater.

`--small-nondetstack-size-kwords` *size*
: Sets the size of the nondet stack for executing parallel computations to *size* kilobytes multiplied by the word size in bytes. The regular nondet stack size must be equal or greater.

`--solutions-heap-size` *size*
: Sets the size of the solutions heap to *size* kilobytes.

`--solutions-heap-size-kwords` *size*
: Sets the size of the solutions heap to *size* kilobytes multiplied by the word size in bytes.

`--trail-segment-size` *size*
: Sets the size of each trail segment to be *size* kilobytes. This option is ignored in grades that do not use a trail.

`--trail-segment-size-kwords` *size*
: Set the size of each trail segment to be *size* kilobytes multiplied by the words size in bytes. This option is ignored in grades that do not use trail.

`--genstack-size` *size*
: Sets the size of the generator stack to *size* kilobytes.

`--genstack-size-kwords` *size*
: Sets the size of the generator stack to *size* kilobytes multiplied by the word size in bytes.

`--cutstack-size` *size*
: Sets the size of the cut stack to *size* kilobytes.

`--cutstack-size-kwords` *size*
: Sets the size of the cut stack to *size* kilobytes multiplied by the word size in bytes.

`--pnegstack-size` *size*
: Sets the size of the pneg stack to *size* kilobytes.

`--pnegstack-size-kwords` *size*
: Sets the size of the pneg stack to *size* kilobytes multiplied by the word size in bytes.

`-i` *filename*
`--mdb-in` *filename*
: Read debugger input from the file or device specified by *filename*, rather than from standard input.

`-o` *filename*
`--mdb-out` *filename*
: Print debugger output to the file or device specified by *filename*, rather than to standard output.

`-e` *filename*
`--mdb-err` *filename*
: Print debugger error messages to the file or device specified by *filename*, rather than to standard error.

`-m` *filename*
`--mdb-tty` *filename*
: Redirect all three debugger I/O streams — input, output, and error messages — to the file or device specified by *filename*.

`--debug-threads`
: Output information to the standard error stream about the locking and unlocking occurring in each module which has been compiled with the C macro symbol '`MR_DEBUG_THREADS`' defined.

`--tabling-statistics`
: Prints statistics about tabling when the program terminates.

`--mem-usage-report` *prefix*
: Print a report about the memory usage of the program when the program terminates. The report is printed to a new file named '`.mem_usage_report`*N*' for the lowest value of *N* (up to 99) which doesn't overwrite an existing file. Note that this capability is not supported on all operating systems.

`--trace-count`
: When the program terminates, generate a trace counts file listing all the debugger events the program executed, if the program actually executed any debugger events. If `MERCURY_OPTIONS` includes the '`--trace-count-output-file` *filename*' option, then the trace counts are put into the file *filename*. If `MERCURY_OPTIONS` includes the '`--trace-count-summary-file` *basename*' option, then the trace counts are put either in the file *basename* (if it does not exist), or in *basename.N* for the lowest value of the integer *N* which doesn't overwrite an existing file. (There is a limit on the value of *N*; see the option '`--trace-count-summary-max`' below.) If neither option is specified, then the output will be written to a file with the prefix '`.mercury_trace_counts`' and a unique suffix. Specifying both options is an error.

`--coverage-test`
: Act as the '`--trace-count`' option, except include *all* debugger events in the output, even the ones that were not executed.

`--trace-count-if-exec` *prog*
: Act as the '`--trace-count`' option, but only if the executable is named *prog*. This is to allow the collection of trace count information from only one Mercury program even if several Mercury programs are executed with the same setting of `MERCURY_OPTIONS`.

`--coverage-test-if-exec` *prog*
: Act as the '`--coverage-test`' option, but only if the executable is named *prog*. This is to allow the collection of coverage test information from only one Mercury program even if several Mercury programs are executed with the same setting of `MERCURY_OPTIONS`.

`--trace-count-output-file` *filename*
: Documented alongside the '`--trace-count`' option.

`--trace-count-summary-file` *basename*
: Documented alongside the '`--trace-count`' option.

`--trace-count-summary-max` *N*
: If `MERCURY_OPTIONS` includes both the '`--trace-count` option' (or one of the other options that imply '`--trace-count`') and the '`--trace-count-summary-file` *basename*' option, then the generated program will put the generated trace counts either in *basename* (if it does not exist), or in *basename.N* for the lowest value of the integer *N* which doesn't overwrite an existing file. The '`--trace-count-summary-max`' option specifies the maximum value of this *N*. When this maximum is reached, the program will invoke the '`mtc_union`' program to summarize *basename*, *basename.1*, ... *basename.N* into a single file *basename*, and delete the rest. By imposing a limit on the total number (and hence indirectly on the total size) of these trace count files, this mechanism allows the gathering of trace count information from a large number of program runs. The default maximum value of *N* is 20.

`--boehm-gc-free-space-divisor` *N*
: This option sets the value of the free space divisor in the Boehm garbage collector to *N*. It is meaningful only when using the Boehm garbage collector. The default value is 3. Increasing its value will reduce heap space but increase collection time. See the Boehm GC documentation for details.

`--boehm-gc-calc-time`
: This option enables code in the Boehm garbage collector to calculate the time spent doing garbage collection so far. Its only effect is to enable the '`report_stats`' predicate in the '`benchmarking`' module of the standard library to report this information.

`--fp-rounding-mode` *mode*
: Set the rounding mode for floating point operations to *mode*. Recognised modes are '`downward`', '`upward`', '`to_nearest`' and '`toward_zero`'. Exactly what modes are available and even if it is possible to set the rounding mode is system dependent.

`MERCURY_COMPILER`
: Filename of the Mercury Compiler.

`MERCURY_MKINIT`
: Filename of the program to create the '`*_init.c`' file.

`MERCURY_DEBUGGER_INIT`
: Name of a file that contains startup commands for the Mercury debugger. This file should contain documentation for the debugger command set, and possibly a set of default aliases.

11 Using a different C compiler

The Mercury compiler takes special advantage of certain extensions provided by GNU C to generate much more efficient code. We therefore recommend that you use GNU C for compiling Mercury programs. However, if for some reason you wish to use another compiler, it is possible to do so. Here is what you need to do.

- Create a new configuration for the Mercury system using the '`mercury_config`' script, specifying the different C compiler, e.g. '`mercury_config --output-prefix=/usr/local/mercury-cc --with-cc=cc`'.
- Add the '`bin`' directory of the new configuration to the beginning of your PATH.
- You must use a grade beginning with '`none`' or '`hlc`' (e.g. '`hlc.gc`'). You can specify the grade in one of three ways: by setting the `MERCURY_DEFAULT_GRADE` environment variable, by adding a line '`GRADE=...`' to your '`Mmake`' file, or by using the '`--grade`' option to '`mmc`'. (You will also need to install those grades of the Mercury library, if you have not already done so.)
- If your compiler is particularly strict in enforcing ANSI compliance, you may also need to compile the Mercury code with '`--no-static-ground-terms`'.

12 Foreign language interface

The Mercury foreign language interfaces allows '`pragma foreign_proc`' to specify multiple implementations (in different foreign programming languages) for a procedure.

If the compiler generates code for a procedure using a back-end for which there are multiple applicable foreign languages, it will choose the foreign language to use for each procedure according to a builtin ordering.

If the language specified in a '`foreign_proc`' is not available for a particular back-end, it will be ignored.

If there are no suitable '`foreign_proc`' clauses for a particular procedure but there are Mercury clauses, they will be used instead.

'`C`'
: This is the default foreign language on all back-ends which compile to C. Only available on back-ends that compile to C.

'`C#`'
: This is the only foreign language for back-ends which compile to C#.

'`Java`'
: This is the only foreign language for back-ends which compile to Java.

13 Stand-alone interfaces

Programs written in a language other than Mercury should not make calls to foreign exported Mercury procedures unless the Mercury runtime has been initialised. (In the case where the Mercury runtime has not been initialised, the behaviour of these calls is undefined.) Such programs must also ensure that any module specific initialisation is performed before calling foreign exported procedures in Mercury modules. Likewise, module specific finalisation may need to be performed after all calls to Mercury procedures have been made.

A stand-alone interface provides a mechanism by which non-Mercury programs may initialise (and shut down) the Mercury runtime plus a specified set of Mercury libraries.

A stand-alone interface is created by invoking the compiler with the '`--generate-standalone-interface`' option. The set of Mercury libraries to be included in the stand-alone interface is given via one of the usual mechanisms for specifying what libraries to link against, e.g. the '`--ml`' and '`--mld`' options. (see Chapter 6 [Libraries], page 11). The Mercury standard library is always included in this set.

In C grades, the '`--generate-standalone-interface`' option causes the compiler to generate an object file that should be linked into the executable. This object file contains two functions: `mercury_init()` and `mercury_terminate()`. The compiler also generates a C header file that contains the prototypes of these functions. (This header file may be included in C++ programs.) The roles of the two functions are described below.

- `mercury_init()`

  Prototype:

  ```
  void mercury_init(int argc, char **argv, void *stackbottom);
  ```

  Initialise the Mercury runtime, standard library and any other Mercury libraries that were specified when the stand-alone interface was generated. *argc* and *argv* are the argument count and argument vector, as would be passed to the function `main()` in a C program. *stackbottom* is the address of the base of the stack. In grades that use conservative garbage collection this is used to tell the collector where to begin tracing. This function must be called before any Mercury procedures and must only be called once. It is recommended that the value of *stackbottom* be set by passing the address of a local variable in the `main()` function of a program, for example:

  ```
  int main(int argc, char **argv) {
      void *dummy;
      mercury_init(argc, argv, &dummy);
      ...
  }
  ```

  Note that the address of the stack base should be word aligned as some garbage collectors rely upon this. (This is why the type of the dummy variable in the above example is `void *`.) If the value of *stackbottom* is `NULL` then the collector will attempt to determine the address of the base of the stack itself. Note that modifying the argument vector, *argv*, after the Mercury runtime has been initialised will result in undefined behaviour since the runtime maintains a reference into *argv*.

- `mercury_terminate()`

  Prototype:

  ```
  int mercury_terminate(void);
  ```

  Shut down the Mercury runtime. The value returned by this function is Mercury's exit status (as set by the predicate '`io.set_exit_status/3`'). This function will also invoke any finalisers contained in the set of libraries for which the stand-alone interface was generated.

The basename of the object and header file are provided as the argument of '`--generate-standalone-interface`' option.

Stand-alone interfaces are not required if the target language is Java or C#. For those target languages the Mercury runtime will be automatically initialised when the classes or library assemblies containing code generated by the Mercury compiler are loaded.

For an example of using a stand-alone interface see the '`samples/c_interface/standalone_c`' directory in the Mercury distribution.

Index

-

--allow-stubs 106
--analyse-exceptions 122
--analyse-mm-tabling 122
--analyse-trail-usage 122
--analysis-repeat 119
--asm-labels 111, 113
--auto-comments 102
--benchmark-modes 106
--benchmark-modes-repeat *num* 106
--binary-switch-size 124
--boehm-gc-calc-time (runtime option) 141
--boehm-gc-free-space-divisor (runtime option) 141
--c-debug 126
--c-include-dir 126
--c-include-directory 126
--cc 126, 142
--cflag 127
--cflags 127
--check-term 107
--check-termination 107
--checked-nondet-tailcalls 124
--chk-term 107
--cil-interpreter 127
--common-data 124
--common-layout-data 124
--common-struct 120
--compile-only 100
--compile-to-c 112
--config-file 133
--constraint-propagation 120
--control-granularity 135
--convert-to-mercury 99
--coverage-test (runtime option) 140
--coverage-test-if-exec (runtime option) 140
--csharp 111
--csharp-compiler 127
--csharp-flag 127
--csharp-flags 127
--csharp-only 112
--cstack-reserve-size 131
--cutstack-size (runtime option) 139
--cutstack-size-kwords (runtime option) .. 139
--debug 16, 113
--debug-det 97
--debug-determinism 97
--debug-indirect-reuse 97
--debug-intermodule-analysis 97
--debug-liveness 97
--debug-make 97
--debug-modes 96
--debug-modes-pred-id 97
--debug-modes-verbose 97
--debug-opt 97
--debug-opt-pred-id 97
--debug-opt-pred-name 97
--debug-pd 97
--debug-threads (runtime option) 140
--debug-type-rep 97
--decl-debug 113
--deep-profiling 111, 114
--deforestation 121
--deforestation-depth-limit 122
--deforestation-size-threshold 122
--deforestation-vars-threshold 122
--delay-constructs 121
--delay-death 101
--delay-death-max-vars 102
--demangle 82
--dense-switch-req-density 123
--dense-switch-size 123
--detstack-size 5
--detstack-size (runtime option) 137
--detstack-size-kwords (runtime option) .. 138
--distance-granularity 135
--dump-hlds 103
--dump-hlds-file-suffix 105
--dump-hlds-inst-limit 105
--dump-hlds-options 103
--dump-hlds-pred-id 105
--dump-hlds-pred-name 105
--dump-mlds 105
--dump-mlds-pred-name 105
--dump-same-hlds 105
--eliminate-local-variables 122
--enable-term 107
--enable-termination 107
--env-type 134
--errorcheck-only 99
--excess-assign 121
--experimental-complexity 114
--extra-init-command 132
--extra-initialization-functions 130
--extra-inits 130
--extra-lib-header 134
--extra-library-header 134
--fact-table-hash-percent-full 116
--fact-table-max-array-size *size* 116
--feedback-file 135
--filenames-from-stdin 135
--find-all-recompilation-reasons 96
--flags 133
--flags-file 133
--follow-code 120
--fp-rounding-mode (runtime option) 141
--framework 130
--framework-directory 131

--garbage-collection 114
--gc 111, 114
--gcc-global-registers 111, 113
--gcc-non-local-gotos 113
--gcc-nonlocal-gotos 111
--generate-bytecode 102
--generate-dependencies 98
--generate-dependency-file 98
--generate-mmc-deps 98
--generate-mmc-make-module-dependencies .. 98
--generate-module-order 98
--generate-source-file-mapping 98
--generate-standalone-interface *basename* 98
--generate-trail-ops-inline 122
--genstack-size (runtime option) 139
--genstack-size-kwords (runtime option) .. 139
--grade 108
--halt-at-syntax-error 90
--halt-at-warn 89
--halt-at-warn-make-int 89
--halt-at-warn-make-interface 89
--halt-at-warn-make-opt 90
--have-delay-slot 117
--heap-size (runtime option) 137
--heap-size-kwords (runtime option) 137
--help 89, 135
--high-level-code 82, 111, 113
--higher-order-arg-limit 121
--higher-order-size-limit 121
--host-env-type 134
--implicit-parallelism 135
--imports-graph 103
--infer-all 106
--infer-det 107
--infer-determinism 107
--infer-modes 106
--infer-types 106
--inform-incomplete-switch 94
--inform-incomplete-switch-threshold 94
--inform-suboptimal-packing 95
--inhibit-style-warnings 89
--inhibit-warnings 89
--init-file 13, 129
--init-file-directory 129
--inline-alloc 126
--inline-compound-threshold 119
--inline-linear-tail-rec-sccs 119
--inline-simple 119
--inline-simple-threshold 119
--inline-single-use 119
--inline-vars-threshold 119
--inlining 119
--install-command 132
--install-command-dir-command 132
--install-prefix 132
--intermod-directory 133
--intermod-inline-simple-threshold 119
--intermod-opt 118
--intermod-unused-args 120
--intermodule-analysis 118
--intermodule-optimization 14, 15, 118
--introduce-accumulators 121
--ite-instead-of-switch 94
--java 111
--java-archive-command 130
--java-classpath 127
--java-compiler 127
--java-flag 127
--java-flags 127
--java-interpreter 127
--java-object-file-extension 127
--java-only 112
--javac 127
--jobs 131
--keep-going 131
--layout-compression-limit 125
--ld-flag 128
--ld-flags 128
--ld-libflag 128
--ld-libflags 128
--lib-linkage 133
--libgrade 132
--libgrades-exclude 132
--libgrades-exclude-component 132
--libgrades-include 132
--libgrades-include-component 132
--library 128
--library-directory 128
--limit-error-contexts 96
--line-numbers 102
--line-numbers-around-foreign-code 102
--line-numbers-for-c-headers 102
--link-executable-command 130
--link-object 13, 128
--link-shared-lib-command 130
--linkage 129
--llds-optimize 102, 124
--local-constraint-propagation 120
--lookup-switch-req-density 123
--lookup-switch-size 123
--loop-invariants 119
--low-level-debug 116
--make 10, 92, 95, 96, 97, 98, 131
--make-int 1, 3, 98
--make-interface 1, 98
--make-opt-int 4, 99
--make-optimization-interface 1, 99
--make-priv-int 3, 99
--make-priv-interface 1
--make-private-interface 1, 99
--make-short-int 1, 3, 98
--make-short-interface 1, 98
--make-trans-opt 1, 4, 99
--make-transitive-optimization-interface 1, 99

--make-xml-documentation 99
--max-contexts-per-thread (runtime option) 137
--max-engines (runtime option) 137
--max-error-line-width 102
--max-jump-table-size *n* 116
--maybe-thread-safe 115
--mdb-err (runtime option) 139
--mdb-in (runtime option) 139
--mdb-in-window (mdb option) 25
--mdb-out (runtime option) 139
--mdb-tty (runtime option) 139
--mem-usage-report (runtime option) 140
--memory-profiling 111, 114
--mercury-config-dir 133
--mercury-configuration-directory 133
--mercury-library 12, 129
--mercury-library-directory 12, 129
--mercury-linkage 129
--mercury-standard-library-directory 129
--mercury-stdlib-dir 129
--middle-rec 124
--ml 12, 129
--mld 12, 129
--mlds-optimize 102, 122
--mlds-peephole 122
--mode-constraints 105
--no- 89
--no-allow-undefined 130
--no-ansi-c 126
--no-asm-labels 111, 113
--no-c-optimize 126
--no-common-data 124
--no-common-struct 120
--no-coverage-profiling 114
--no-default-runtime-library-directory 128
--no-delay-death 101
--no-demangle 82, 129
--no-detect-libgrades 132
--no-eliminate-local-variables 122
--no-emit-c-loops 125
--no-follow-code 120
--no-fully-strict 106
--no-gcc-global-registers 111, 113
--no-gcc-non-local-gotos 113
--no-gcc-nonlocal-gotos 111
--no-generate-trail-ops-inline 122
--no-halt-at-invalid-interface 90
--no-high-level-code 111
--no-infer-det 107
--no-infer-determinism 107
--no-inform-inferred 95
--no-inform-inferred-modes 95
--no-inform-inferred-types 95
--no-inline-builtins 119
--no-inline-simple 119
--no-inline-single-use 119
--no-inlining 119
--no-layout-common-data 124
--no-libgrade 132
--no-libgrade-install-check 133
--no-line-numbers 102
--no-line-numbers-around-foreign-code 102
--no-line-numbers-for-c-headers 102
--no-llds-optimize 102, 124
--no-local-thread-engine-base 126
--no-main 130
--no-mercury-standard-library-directory 129
--no-mercury-stdlib-dir 129
--no-middle-rec 124
--no-mlds-optimize 102, 122
--no-mlds-peephole 122
--no-optimize-delay-slot 125
--no-optimize-format-calls 121
--no-optimize-frames 125
--no-optimize-fulljumps 124
--no-optimize-initializations 122
--no-optimize-jumps 124
--no-optimize-labels 125
--no-optimize-peep 124
--no-optimize-tailcalls 122
--no-read-opt-files-transitively 118
--no-reclaim-heap-on-failure 116
--no-reclaim-heap-on-nondet-failure 116
--no-reclaim-heap-on-semidet-failure 116
--no-reorder-conj 106
--no-reorder-disj 106
--no-simple-neg 124
--no-smart-indexing 123
--no-static-ground-terms 123
--no-strip 130
--no-trad-passes 96, 116
--no-trail-segments 111
--no-type-layout 116
--no-use-atomic-cells 124
--no-use-readline 130
--no-verbose-make 95
--no-warn-accumulator-swaps 90
--no-warn-det-decls-too-lax 90
--no-warn-implicit-stream-calls 92
--no-warn-inferred-erroneous 90
--no-warn-insts-with-functors-without-type 91
--no-warn-insts-without-matching-type 90
--no-warn-interface-imports 91
--no-warn-interface-imports-in-parents 91
--no-warn-known-bad-format-calls 93
--no-warn-missing-det-decls 90
--no-warn-missing-module-name 92
--no-warn-non-contiguous-clauses 92
--no-warn-non-contiguous-decls 91
--no-warn-non-term-special-preds 93
--no-warn-nothing-exported 91
--no-warn-obsolete 94
--no-warn-only-one-format-string-error 94

--no-warn-simple-code 92
--no-warn-singleton-variables 90
--no-warn-singleton-vars 90
--no-warn-smart-recompilation 92
--no-warn-state-var-shadowing 95
--no-warn-stubs 93
--no-warn-suspected-occurs-check-failure 95
--no-warn-table-with-inline 93
--no-warn-target-code 93
--no-warn-undefined-options-variables 92
--no-warn-undefined-options-vars 92
--no-warn-unresolved-polymorphism 94
--no-warn-unused-imports 91
--no-warn-up-to-date 93
--no-warn-wrong-module-name 92
--nondetstack-size 5
--nondetstack-size (runtime option) 138
--nondetstack-size-kwords (runtime option) 138
--num-ptag-bits 115
--num-real-f-regs 117
--num-real-f-temps 117
--num-real-r-regs 117
--num-real-r-temps 117
--num-tag-bits 115
--opt-level 117
--opt-space 118
--optimization-level 117
--optimize-constant-propagation 121
--optimize-constructor-last-call 121
--optimize-dead-procs 121
--optimize-delay-slot 125
--optimize-duplicate-calls 121
--optimize-dups 125
--optimize-frames 125
--optimize-fulljumps 124
--optimize-higher-order 120
--optimize-initializations 122
--optimize-jumps 124
--optimize-labels 125
--optimize-peep 124
--optimize-proc-dups 125
--optimize-reassign 125
--optimize-repeat 125
--optimize-saved-vars 121
--optimize-space 118
--optimize-tailcalls 122
--optimize-unused-args 120
--options-file 133
--options-search-directory 133
--order-make-by-timestamp 134
--output-c-compiler-type 100
--output-c-include-dir-flags 101
--output-c-include-directory-flags 101
--output-cc 100
--output-cc-type 100
--output-cflags 100
--output-class-dir 101
--output-class-directory 101
--output-compile-error-lines 96
--output-csharp-compiler 100
--output-csharp-compiler-type 100
--output-file 127
--output-grade-defines 101
--output-grade-string 100
--output-java-class-dir 101
--output-java-class-directory 101
--output-libgrades 100
--output-library-link-flags 100
--output-link-command 100
--output-shared-lib-link-command 100
--output-stdlib-grades 100
--output-stdlib-modules 101
--output-target-arch 101
--parallel 81, 115
--pessimize-tailcalls 124
--pnegstack-size (runtime option) 139
--pnegstack-size-kwords (runtime option) 139
--pre-link-command 131
--pretty-print 99
--proc-size-statistics 96
--procs-per-c-function 125
--profile-for-feedback 114
--profile-for-implicit-parallelism 114
--profile-optimized 101
--profiling 111, 113
--program-in-window (mdb option) 25
--rebuild 131
--reclaim-heap-on-failure 116
--reclaim-heap-on-nondet-failure 116
--reclaim-heap-on-semidet-failure 116
--record-term-sizes-as-cells 111
--record-term-sizes-as-words 111
--reorder-conj 106
--reorder-disj 106
--report-cmd-line-args 96
--report-cmd-line-args-in-doterr 96
--restricted-command-line 134
--runtime-flags 130
--runtime-library-directory 128
--search-directory 133
--search-lib-files-dir 13, 128
--search-library-files-directory 128
--shlib-linker-install-name-path 128
--show-all-type-representations 103
--show-definition-extents 103
--show-definition-line-counts 102
--show-definitions 102
--show-dependency-graph 103
--show-local-type-representations 103
--show-make-times 134
--sign-assembly 131
--simple-mode-constraints 105
--simple-neg 124

--single-prec-float 111, 115
--small-detstack-size (runtime option) ... 138
--small-detstack-size-kwords (runtime option) 138
--small-nondetstack-size (runtime option) 138
--small-nondetstack-size-kwords (runtime option) 138
--smart-indexing 123
--smart-recompilation 2, 101
--solutions-heap-size (runtime option) ... 138
--solutions-heap-size-kwords (runtime option) 138
--stack-segments 111, 113
--stack-trace-higher-order 102
--static-ground-terms 123
--statistics 96
--string-switch-size 123
--strip-executable-command 130
--strip-executable-shared-flags 130
--strip-executable-static-flags 130
--system-env-type 134
--tabling-statistics (runtime option) 140
--tag-switch-size 123
--target 111
--target-code-only 99
--target-debug 126
--target-env-type 135
--term-err-limit 108
--term-path-limit 108
--term-single-arg *limit* 107
--termination-error-limit 108
--termination-norm 108
--termination-path-limit 108
--termination-single-argument-analysis .. 107
--thread-pinning (runtime option) 137
--threadscope 81, 115
--time-profiling 113
--trace-count (runtime option) 140
--trace-count-if-exec (runtime option) ... 140
--trace-count-output-file (runtime option) 141
--trace-count-summary-file (runtime option) 141
--trace-count-summary-max (runtime option) 141
--trace-init-file 129
--trace-level *level* 101
--trace-optimized 101
--trad-passes 96, 116
--trail-segment-size (runtime option) 138
--trail-segment-size-kwords (runtime option) 139
--trail-segments 111, 114
--trans-intermod-opt 14, 118
--transitive-intermodule-optimization 4, 118
--try-switch-size 124
--tty (mdb option) 24
--type-inference-iteration-limit 107
--type-layout 116
--type-specialization 120
--typecheck-only 99
--unneeded-code 120
--unneeded-code-copy-limit 120
--use-atomic-cells 124
--use-grade-subdirs 134
--use-local-vars 125
--use-macro-for-redo-fail 125
--use-opt-files 118
--use-search-directories-for-intermod ... 133
--use-subdirs 1, 134
--use-trail 111, 114
--use-trans-opt-files 118
--user-guided-type-specialization 120
--verb-check-term 107
--verb-chk-term 107
--verbose 95
--verbose-check-termination 107
--verbose-commands 96
--verbose-dump-mlds 105
--verbose-error-messages 95
--verbose-recompilation 96
--very-verbose 95
--warn-accumulator-swaps 90
--warn-dead-predicates 93
--warn-dead-preds 93
--warn-dead-procedures 93
--warn-dead-procs 93
--warn-det-decls-too-lax 90
--warn-duplicate-calls 92
--warn-implicit-stream-calls 92
--warn-inconsistent-pred-order 91
--warn-inconsistent-pred-order-clauses ... 91
--warn-inconsistent-pred-order-foreign-procs 91
--warn-inferred-erroneous 90
--warn-insts-with-functors-without--type 91
--warn-insts-without-matching-type 90
--warn-interface-imports 91
--warn-interface-imports-in-parents 91
--warn-missing-det-decls 90
--warn-missing-module-name 92
--warn-missing-opt-files 91
--warn-missing-trans-opt-files 91
--warn-non-contiguous-foreign-procs 92
--warn-non-stratification 92
--warn-non-tail-recursion 93
--warn-nothing-exported 91
--warn-obvious-non-tail-recursion 93
--warn-potentially-ambiguous-pragma 95
--warn-simple-code 92
--warn-singleton-variables 90
--warn-singleton-vars 90
--warn-smart-recompilation 92

--warn-stubs ... 93
--warn-suspicious-foreign-code ... 94
--warn-suspicious-foreign-procs ... 94
--warn-suspicious-recursion ... 92
--warn-unknown-format-calls ... 94
--warn-unused-args ... 91
--warn-unused-imports ... 91
--warn-up-to-date ... 93
--warn-wrong-module-name ... 92
--window (mdb option) ... 25
--window-command (mdb option) ... 25
-? ... 135
-c ... 4, 100
-C ... 99
-C (runtime option) ... 136
-d ... 103
-D (runtime option) ... 136
-e ... 99
-e (runtime option) ... 139
-E ... 95
-F ... 131
-h ... 135
-H ... 113
-i ... 98
-i (runtime option) ... 139
-I ... 133
-j ... 131
-k ... 131
-l ... 128
-L ... 128
-m ... 131
-m (runtime option) ... 139
-M ... 98
-n ... 102
-N ... 96
-o ... 4, 127
-o (runtime option) ... 139
-O ... 117
-p (runtime option) ... 136
-P ... 99
-P (runtime option) ... 136
-r ... 131
-R ... 128
-s ... 108
-S ... 96
-T (runtime option) ... 137
-track-flags ... 131
-v ... 95
-V ... 95
-w ... 89

.

.debug (grade modifier) ... 109
.decldebug (grade modifier) ... 109
.gc (grade modifier) ... 109
.mdbrc ... 24
.memprof (grade modifier) ... 109
.mm (grade modifier) ... 109
.par (grade modifier) ... 109
.prof (grade modifier) ... 109
.profdeep (grade modifier) ... 109
.spf (grade modifier) ... 109
.stseg (grade modifier) ... 109
.threadscope (grade modifier) ... 109
.tr (grade modifier) ... 109
.trseg (grade modifier) ... 109

/

/debug ... 126

A

ABI (Application Binary Interface) ... 108
Accumulator introduction ... 121
alias (mdb command) ... 56
all_class_decls (mdb command) ... 65
all_procedures (mdb command) ... 65
all_regs (mdb command) ... 63
all_type_ctors (mdb command) ... 64
Allocation profiling ... 80, 81, 82, 85, 114
ambiguity (mdb command) ... 65
Application Binary Interface (ABI) ... 108
AR ... 14
ARFLAGS ... 14
asm_fast (compilation grade) ... 108
Automatic garbage collection ... 114
Automatic parallelism ... 114, 135

B

Backtrackable destructive update ... 114
Boehm (et al) conservative garbage collector ... 114
break (mdb command) ... 44
break points ... 25
break_print (mdb command) ... 49
Breakpoints ... 44
browse (mdb command) ... 38
Building profiled applications ... 81
Building programs ... 5

C

c2init ... 4
C compiler ... 126
C compiler options ... 127
C compilers ... 142
C debugging ... 126
C# compiler ... 127
C# compiler options ... 127
C# libraries ... 16
C2INITARGS ... 9
C2INITFLAGS ... 8
call (trace event) ... 19

Call graph profile ... 83
cc_query (mdb command) ... 31
CFLAGS ... 8
CIL interpreter ... 127
CLANG_FLAGS ... 11
class_decl (mdb command) ... 65
classpath ... 127
Clauses, procedures without ... 106
clear_histogram (mdb command) ... 59
Code generation options ... 116
Compilation grades ... 108
Compilation model options ... 108
Compilation models ... 108
Compilation speed ... 117
cond (trace event) ... 20
condition (mdb command) ... 48
Conservative garbage collection ... 114
Constraint solving ... 114
consumer (mdb command) ... 62
context (mdb command) ... 52
continue (mdb command) ... 35
control-C ... 26
Coverage Profiling ... 114
Coverage profiling ... 114
Creating profiles ... 82
Cross-compiling ... 115, 117
Cross-module optimization ... 117, 118
Cross-module optimization, transitive ... 118
csharp (compilation grade) ... 108
current (mdb command) ... 41
Cut stack size ... 139
cut_stack (mdb command) ... 62

D

Data representation ... 115
Dead function elimination ... 121
Dead predicate elimination ... 121
Dead procedure elimination ... 121
debug_vars (mdb command) ... 63
debugger break points ... 25
debugger interrupt ... 26
debugger print level ... 26
debugger procedure specification ... 28
debugger trace events ... 19
debugger trace level ... 22
Debugging ... 16, 113
Debugging the generated C code ... 126
Debugging Threads ... 140
Deep profiling ... 80, 81, 82, 87, 114
deep tracing ... 22
delete (mdb command) ... 50
dependencies ... 98
depth (mdb command) ... 55
Destructive update, backtrackable ... 114
Determinism inference ... 107
dice (mdb command) ... 59
diff (mdb command) ... 43
Directories ... 126, 127, 133, 134, 136
Directories for libraries ... 128, 129, 131
disable (mdb command) ... 49
disj (trace event) ... 20
document (mdb command) ... 57
document_category (mdb command) ... 56
down (mdb command) ... 40
dump (mdb command) ... 43
Duplicate call optimization ... 121

E

echo (mdb command) ... 52
Efficiency ... 80
else (trace event) ... 20
Emacs ... 17
enable (mdb command) ... 50
Environment variables ... 136
exception (mdb command) ... 34
exit (trace event) ... 19
EXTRA_C2INITARGS ... 9
EXTRA_C2INITFLAGS ... 8
EXTRA_CFLAGS ... 8
EXTRA_GRADEFLAGS ... 7
EXTRA_JAVACFLAGS ... 8
EXTRA_LD_LIBFLAGS ... 8
EXTRA_LDFLAGS ... 8
EXTRA_LIB_DIRS ... 9, 15
EXTRA_LIBRARIES ... 9, 15
EXTRA_MC_MAKE_FLAGS ... 10
EXTRA_MCFLAGS ... 8
EXTRA_MGNUCFLAGS ... 8
EXTRA_MLFLAGS ... 8
EXTRA_MLLIBS ... 8
EXTRA_MLOBJS ... 8

F

Fact tables ... 116
fail (trace event) ... 19
fail_trace_counts (mdb command) ... 54
File extensions ... 1, 127
file names ... 52
File names ... 1, 134
finish (mdb command) ... 33
flag (mdb command) ... 62
Flat profile ... 83
format (mdb command) ... 55
format_param (mdb command) ... 55
forward (mdb command) ... 34

G

Garbage collection ... 114
Garbage collection, profiling ... 83
GCC_FLAGS ... 11
gen_stack (mdb command) ... 62
Generator stack size ... 139

GNU C 142
GNU Emacs 17
goal path 20
`goal_path (mdb command)` 52
`goto (mdb command)` 33
`GRADEFLAGS` 7, 9, 108
Grades 108, 134
Grades and grade components 108

H

Heap overflow 5
Heap profiling 80, 81, 82, 86, 114
Heap size 5
held variables (in mdb) 27
`held_vars (mdb command)` 37
`help (mdb command)` 57
Help option 135
Higher-order specialization 120
`histogram_all (mdb command)` 59
`histogram_exp (mdb command)` 59
hlc (compilation grade) 108
HLDS 119, 123
`hold (mdb command)` 42

I

`ignore (mdb command)` 48
Include directories 126
Inference 106
Inference of determinism 107
Inference of modes 106, 107
Inference of types 106, 107
Inlining 118, 119
`INSTALL` 7, 9, 15
`INSTALL_MKDIR` 7, 9, 15
`INSTALL_PREFIX` 9, 15
Intermodule optimization 117, 118
Intermodule optimization, transitive 118
interrupt, in debugger 26
`io_query (mdb command)` 31

J

jar files 16
java (compilation grade) 108
Java compiler 127
Java compiler options 127
Java interpreter 127
Java libraries 16
`JAVACFLAGS` 8

L

Language semantics options 106
Last call optimization 121
`LD_BIND_NOW` 88
`LD_LIBFLAGS` 8
`LDFLAGS` 8
`level (mdb command)` 41
`LIB_LINKAGES` 7, 9
`LIBGRADES` 7, 9, 15
Libraries 11
Libraries, linking with 12, 13, 15, 128, 129
line numbers 52
`lines (mdb command)` 55
Link errors 108
Link options 127
`LINKAGE` 8
Linker options 127
`list (mdb command)` 43
`list_cmd (mdb command)` 54
`list_context_lines (mdb command)` 53
`list_path (mdb command)` 54
LLDS 123, 124
LLDS back-end compilation model options 113

M

Mac OS X, Darwin, Install name 128
Mac OS X, Using Frameworks 130, 131
`MAIN_TARGET` 7
`make --- see Mmake` 5
`max_io_actions (mdb command)` 55
`maxdepth (mdb command)` 35
`MC` 7
`MC_BUILD_FILES` 10
`MC_MAKE_FLAGS` 10
`MCFLAGS` 8, 108
`mdb` 16
mdbrc 24
`mdprof` 80, 81, 82, 87
Measuring performance 80
Memory attribution 86
Memory profiling 80, 81, 82, 85, 114
Memory retention 86
'`Mercury`' subdirectory 134
Mercury.options 10, 133
`MERCURY_COMPILER` 141
`MERCURY_DEBUGGER_INIT` 24, 142
`MERCURY_DEFAULT_GRADE` 111, 136, 142
`MERCURY_LINKAGE` 8
`MERCURY_MAIN_MODULES` 10
`MERCURY_MKINIT` 142
`MERCURY_OPTIONS` 5, 136
`MERCURY_STDLIB_DIR` 136
`mgnuc` 4
`MGNUC` 8
`MGNUCFLAGS` 8, 108
Microsoft Management Console 2
`mindepth (mdb command)` 35
`ml` 4
`ML` 8, 14
MLDS 122
MLDS back-end compilation model options 113
`MLFLAGS` 8, 14, 108

MLLIBS ... 8, 14
MLOBJS ... 8, 14
MLPICOBJS ... 14
mm_stacks (mdb command) ... 62
mmake ... 5
Mmake variables ... 7
mmc ... 2
mmc_options (mdb command) ... 51
Mode inference ... 107
modules (mdb command) ... 50
mprof ... 80, 81, 82, 85, 86, 88
MSVC_FLAGS ... 11

N

neg_enter (trace event) ... 20
neg_fail (trace event) ... 20
neg_success (trace event) ... 20
next (mdb command) ... 33
No clauses, procedures with ... 106
nondet_stack (mdb command) ... 63
none (compilation grade) ... 108

O

Object files, linking with ... 128
Optimization ... 80, 106
Optimization levels ... 117
Optimization options ... 117
Optimizing code size ... 118
Optimizing space ... 118
Optional features compilation model options ... 113
Options files ... 10, 133
Order of execution ... 106

P

Parallel execution ... 115
Parallel execution profiling ... 88
Parallel performance ... 80
Parallel runtime profiling ... 81
pass_trace_counts (mdb command) ... 54
path ... 20
paths in terms ... 27
PIC (position independent code) ... 13
Pneg stack size ... 139
pneg_stack (mdb command) ... 62
pop_list_dir (mdb command) ... 54
Position independent code ... 13
pragma fact_table ... 116
print (mdb command) ... 37
print level ... 26
print_optionals (mdb command) ... 63
printlevel (mdb command) ... 51
procedure specification (in mdb) ... 28
procedures (mdb command) ... 50
Procedures with no clauses ... 106
Profiler feedback ... 114, 135
Profiling ... 80, 82, 113, 114
Profiling and shared libraries ... 88
Profiling interrupts ... 83
Profiling memory allocation ... 80, 85
push_list_dir (mdb command) ... 54

Q

query (mdb command) ... 31
quit (mdb command) ... 59

R

RANLIB ... 14
RANLIBFLAGS ... 14
Recompiling ... 5
redo (trace event) ... 19
reg (compilation grade) ... 108
register (mdb command) ... 50
Reordering ... 106
retry (mdb command) ... 35
return (mdb command) ... 34

S

save (mdb command) ... 59
scope (mdb command) ... 52
scroll (mdb command) ... 51
Search path ... 133, 136
Search path for libraries ... 128, 129, 131
Semantics options ... 106
shallow tracing ... 22
Shared libraries ... 13
Shared libraries and profiling ... 88
Shared objects ... 13
show-ghc-events ... 88
size (mdb command) ... 55
source (mdb command) ... 58
Specialization of higher-order calls ... 120
spy points ... 25
stack (mdb command) ... 40
Stack overflow ... 5
Stack segments ... 113
Stack size ... 5
stack_default_limit (mdb command) ... 52
stack_regs (mdb command) ... 63
Static libraries ... 13
stats (mdb command) ... 63
step (mdb command) ... 32
strict debugger commands ... 26
Stubs ... 106
Subdirectories ... 134
subgoal (mdb command) ... 62
switch (trace event) ... 20

T

table (mdb command) ... 64

`table_io (mdb command)` ... 50
Tags ... 115
Tail recursion optimization ... 121
Target code compilation options ... 126
Target options ... 112, 117
term navigation ... 27
Termination analysis ... 118
Termination analysis options ... 107
then (trace event) ... 20
Threads, Debugging ... 140
`threadscope` ... 81, 88
ThreadScope profiling ... 88
Threadscope profiling ... 81, 115
Time profiling ... 80, 81, 82, 113
trace events ... 19
trace level ... 22
Tracing ... 16, 19
`track (mdb command)` ... 36
Trail size ... 138, 139
`trail_details (mdb command)` ... 66
Trailing ... 114
Transitive intermodule optimization ... 118
`trust (mdb command)` ... 58
`trusted (mdb command)` ... 58
Type inference ... 107
Type specialization ... 120
Type specialization, user guided ... 120
`type_ctor (mdb command)` ... 64
`typecheck-ambiguity-error-limit` ... 135
`typecheck-ambiguity-warn-limit` ... 135

U

`unalias (mdb command)` ... 56
Undefined symbol ... 108
`unhide_events (mdb command)` ... 64
`untrust (mdb command)` ... 58
Unused arguments ... 120
Unused outputs ... 120
Unused procedure elimination ... 121
`up (mdb command)` ... 40
`user (mdb command)` ... 34
user defined event attributes (in mdb) ... 27
user defined events (in mdb) ... 27
`user_event_context (mdb command)` ... 53
Using a different C compiler ... 142

V

`var_details (mdb command)` ... 62
Variables, environment ... 136
Variables, Mmake ... 7
`vars (mdb command)` ... 37
Verbosity options ... 95
`view (mdb command)` ... 41

W

Warning options ... 89
`web_browser_cmd (mdb command)` ... 55
`width (mdb command)` ... 55