

The Mercury User's Guide

Version rotd-2026-06-09

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	Introduction to compiling Mercury programs	1
2.1	Mmc arguments.....	1
2.1.1	Option arguments.....	2
2.1.2	@filename arguments.....	3
2.1.3	File name and module name arguments.....	3
2.2	Compiling single-module programs.....	3
2.3	Compiling multi-module programs.....	4
2.3.1	Managing connections between modules.....	4
2.3.2	Introduction to mmake.....	5
2.3.3	Introduction to mmc -make.....	7
3	Mercury grades	8
3.1	The Mercury backends.....	8
3.2	The importance of consistency.....	9
3.3	Base grades.....	10
3.4	Grade modifiers.....	11
3.4.1	Grade modifiers for debugging Mercury programs.....	11
3.4.2	Grade modifiers for profiling Mercury programs.....	12
3.4.3	Grade modifiers for trailing.....	13
3.4.4	Grade modifiers for parallelism.....	13
3.4.5	Grade modifiers for minimal model tabling.....	14
3.4.6	Grade modifiers for flexible stack sizes.....	15
3.4.7	Grade modifiers for garbage collection.....	16
3.4.8	Grade modifiers for single precision floats.....	16
3.4.9	Grade modifiers for target language debugging.....	17
3.4.10	Grade modifiers for pregenerated source distributions.....	17
3.4.11	Compatibility of grade modifiers.....	18
4	Running Mercury programs	18
5	The Mercury compilation process in detail ..	19
5.1	Creating interface files.....	19
5.2	Creating optimization files.....	19
5.3	Creating target and object code files.....	23
5.4	Creating executables.....	25

6	File naming conventions	26
6.1	Interface files	26
6.2	Optimization files	27
6.3	Timestamp files	27
6.4	Mmakefile fragment files	27
6.5	Files generated when targeting C	28
6.6	Files generated when targeting Java	28
6.7	Files generated when targeting C#	28
7	Using Mmake	28
8	Libraries	34
8.1	Writing libraries	34
8.2	Building with <code>mmc -make</code>	35
8.2.1	Building and installing libraries with <code>mmc -make</code>	35
8.2.2	Using installed libraries with <code>mmc -make</code>	35
8.2.3	Using non-installed libraries with <code>mmc -make</code>	36
8.3	Building with Mmake	36
8.3.1	Building libraries with Mmake	36
8.3.2	Installing libraries with Mmake	38
8.3.3	Using libraries with Mmake	38
8.4	Libraries and the Java grade	39
8.5	Libraries and the C# grade	39
9	Debugging	39
9.1	Quick overview	39
9.2	GNU Emacs interface	41
9.3	Tracing of Mercury programs	42
9.4	Preparing a program for debugging	45
9.5	Tracing optimized code	47
9.6	Mercury debugger invocation	47
9.7	Mercury debugger concepts	48
9.8	User defined events	52
9.9	I/O tabling	53
9.10	Debugger commands	54
9.10.1	Interactive query commands	54
9.10.2	Forward movement commands	55
9.10.3	Backward movement commands	59
9.10.4	Browsing commands	60
9.10.5	Breakpoint commands	67
9.10.6	I/O tabling commands	73
9.10.7	Parameter commands	74
9.10.8	Help commands	80
9.10.9	Declarative debugging <code>mdb</code> commands	80
9.10.10	Miscellaneous commands	82
9.10.11	Experimental commands	82
9.10.12	Developer commands	85

9.11	Declarative debugging.....	90
9.11.1	Overview	90
9.11.2	Concepts.....	91
9.11.3	Oracle questions	91
9.11.4	Commands.....	92
9.11.5	Diagnoses.....	95
9.11.6	Search modes	96
9.11.6.1	Top-down mode.....	96
9.11.6.2	Divide and query mode.....	97
9.11.6.3	Suspicion divide and query mode.....	97
9.11.6.4	Binary search mode	97
9.11.7	Improving the search.....	97
9.11.7.1	Tracking suspicious subterms	97
9.11.7.2	Trusting predicates, functions and modules	98
9.11.7.3	When different search modes are used	99
9.12	Trace counts.....	99
9.12.1	Generating trace counts	99
9.12.2	Combining trace counts	100
9.12.3	Slicing.....	101
9.12.4	Dicing.....	102
9.12.5	Coverage testing.....	104
10	Profiling.....	105
10.1	Profiling introduction.....	105
10.2	Building profiled applications	105
10.3	Creating profiles.....	106
10.4	Using mprof for time profiling.....	107
10.5	Using mprof for profiling memory allocation.....	110
10.6	Using mprof for profiling memory retention.....	110
10.7	Using mdprof.....	111
10.8	Using threadscope	113
10.9	Profiling and shared libraries.....	113
11	Invocation.....	113
11.1	Invocation overview.....	113
11.2	Help options.....	114
11.3	Options for modifying the command line	114
11.4	Options that give the compiler its overall task	114
11.5	Grade options	119
11.5.1	Grades and grade components.....	119
11.5.2	Target options	119
11.5.3	LLDS backend grade options	120
11.5.4	MLDS backend grade options	121
11.5.5	Debugging grade options	121
11.5.5.1	Mdb debugging grade options	121
11.5.6	Profiling grade options	121
11.5.6.1	Mprof profiling grade options.....	121
11.5.6.2	Deep profiling grade options.....	122

11.5.7	Optional feature grade options	122
11.5.8	Developer grade options	123
11.6	Options that control inference	123
11.7	Options specifying the intended semantics	124
11.8	Verbosity options	124
11.9	Diagnostics options	125
11.9.1	Options that control diagnostics	125
11.9.2	Options that control color in diagnostics	126
11.10	Warning options	126
11.10.1	Warnings about possible incorrectness	126
11.10.1.1	Warnings about possible module incorrectness	126
11.10.1.2	Warnings about possible inst incorrectness	127
11.10.1.3	Warnings about possible predicate incorrectness ..	128
11.10.1.4	Warnings about possible pragma incorrectness	129
11.10.1.5	Warnings about possible goal incorrectness	129
11.10.1.6	Warnings about missing files	131
11.10.2	Warnings about possible performance issues	131
11.10.3	Warnings about programming style	132
11.10.3.1	Warnings about style issues with modules	132
11.10.3.2	Warnings about style issues with predicates	132
11.10.3.3	Warnings about style issues with goals	133
11.10.3.4	Warnings about missing order	135
11.10.3.5	Warnings about missing contiguity	135
11.10.4	Options that control warnings	136
11.10.5	Options about halting for warnings	136
11.11	Options that request information	137
11.12	Options that ask for informational files	137
11.13	Controlling trace goals	139
11.14	Preparing code for mdb debugging	139
11.15	Preparing code for mdprof profiling	140
11.16	Optimization options	140
11.16.1	Overall control of optimizations	140
11.16.2	Source-to-source optimizations	141
11.16.3	Optimizations during code generation	145
11.16.4	Optimizations specific to high level code	147
11.16.5	Optimizations specific to low level code	147
11.17	Intermodule optimization	151
11.17.1	Non-transitive intermodule optimization	151
11.17.2	Transitive intermodule optimization	151
11.18	Program analyses	152
11.18.1	The termination analyser based on linear inequality constraints	152
11.18.2	Other program analyses	153
11.19	Options that ask for modified output	153
11.20	Options for controlling mmc -make	154
11.21	Options for target language compilation	155
11.21.1	General options for compiling target language code	155
11.21.2	Options for compiling C code	155

11.21.3	Options for compiling Java code.....	156
11.21.4	Options for compiling C# code	157
11.22	Options for linking.....	157
11.22.1	General options for linking.....	157
11.22.2	Options for linking C or C# code.....	158
11.22.3	Options for linking just C code.....	158
11.22.4	Options for linking just Java code.....	161
11.22.5	Options for linking just C# code.....	161
11.23	Options controlling searches for files.....	161
11.24	Options controlling the library installation process	162
11.25	Options specifying properties of the environment	164
11.26	Options that record autoconfigured parameters.....	165
11.27	Options for developers only	165
11.27.1	Operation selection options for developers only	165
11.27.2	Dumping out internal compiler data structures	166
11.27.3	Options intended for internal use by the compiler only	166
11.28	Now-unused former options kept for compatibility	166
12	Environment variables.....	167
12.1	Environment variables affecting the Mercury compiler	167
12.2	Environment variables affecting the Mercury debugger.....	167
12.3	Environment variables affecting the Mercury runtime system	168
13	Diagnostic output.....	173
13.1	Verbose vs nonverbose messages.....	174
13.2	Ordering diagnostics.....	174
13.3	Color schemes	175
13.4	Enabling the use of color.....	177
14	Using a different C compiler.....	178
15	Foreign language interface	179
16	Stand-alone interfaces	179
Index.....		181

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 Introduction to compiling Mercury programs

The Mercury compiler is called `mmc`, which stands for “Melbourne Mercury Compiler”.

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 its alternative name ‘`mercury`’, or adjust your `PATH` to ensure that the Mercury ‘`bin`’ directory precedes the Windows system directory.

2.1 Mmc arguments

Any useful invocation of `mmc` will specify one or more command line arguments. How the compiler treats each argument depends on its form.

- Any command line argument that starts with ‘`-`’ specifies one or more *options*.
- Any command line argument that starts with ‘`@`’ is a shorthand for a list of other arguments.
- Any command line argument that does not start with either ‘`-`’ or ‘`@`’ will be treated either as the argument of an option (if it immediately follows an option that takes an argument), or as a *non-option argument* (if it does not immediately follow an option that takes an argument).

In the absence of the `--make` option, whose description we will defer until [Section 2.3.3 \[Introduction to `mmc --make`\], page 7](#), all non-option arguments should be either the name of a file, or the name of a module. The compiler assumes that non-option arguments ending in ‘`.m`’ are file names, while all other non-option arguments are module names. Both file names and module names tell the compiler what code it should operate on.

2.1.1 Option arguments

The Mercury compiler follows the usual Unix conventions around options. Some of its options (e.g. `-c`, `-o`, and `-I`) have a similar meaning to that in compilers for other languages, though of course most are specific to Mercury.

Like many other programs, the Mercury compiler supports both short (single-character) and long (multi-character) option names.

On command lines, an option argument that starts with `--` specifies a single long option, while an option argument that starts with only a single `-` specifies one or more short options. For example, `-v` asks the compiler to be verbose, printing a message each time it starts a new phase of the compilation process, while `-w` asks it not to print any warnings.

Some options do not take arguments (we call such options *flags*), while some do. Single-letter flags may be grouped with a single `-`, so that e.g. users can ask for both verbose output and no warnings either by giving each short option in separate arguments, as in `-v -w`, or by giving them together in one argument, as in `-vw`. Such grouping cannot be done with long option names: a single argument can specify just one long option name.

All options that have a short name have a long name as well, and it does not matter which one is used. For example, `-v` is interchangeable with `--verbose`, and `-w` is interchangeable with `--inhibit-warnings`.

In addition, many options have two or more long names. There are two common reasons for these synonyms. The reason why the option `intermodule-optimization` can also be specified as `intermodule-optimisation` is to allow users to choose either American or British spelling as they wish. And the reason why that same option can also be specified as `intermod-opt` is to reduce the amount of typing required.

Flag options specify a boolean value: they either ask for something specific to be done, or they ask for that specific something not to be done. If you want to negate a single-letter flag, you can do so by appending a trailing `-` after the short option name, as in e.g. `-v-`. (You cannot both group *and* negate single-letter flags at the same time.) Long flags may be negated by preceding them with `no-`, as in e.g. `--no-verbose`.

Other options specify integer or string values. For these options, the value can be specified in the same argument as the option name, following it immediately (if the option name is a short name), or separated from it by an equal sign (if the option name is a long name). Or it can be in the next argument. For example, the requested optimization level, which is an integer, can be specified as either `-O3` or as `-O 3`, using the short name of that option, or as `--optimization-level=3` or as `--optimization-level 3` using the long name of that option. The name of the color scheme to be used for diagnostics, which is a string, can be specified either as `--color-scheme=light16` or as `--color-scheme light16`.

Some options specify an *optional* integer or string value. For example, you can specify the maximum line width for error messages as `--max-error-line-width=71`, or you can ask for error messages to have *no* maximum line width with a `no-` prefix instead of a value, as in `--no-max-error-line-width`.

Some options let users specify a *list* of strings, one string at a time, with each occurrence of the option adding one string to the end of the list. For example, the options

`--search-directory=foo --search-directory=bar` tell the compiler to add the named directories to the list of directories to be searched when looking for e.g. interface files [Section 6.1 \[Interface files\], page 26](#). (The default value of that list consists of just the current directory.) Negating the option name, e.g. with `--no-search-directory`, clears the list, i.e. sets it to the empty list.

All option names, both short and long, are case-sensitive. For example, the meanings of `-e` and `-E` are totally unrelated.

2.1.2 @filename arguments

Any command line argument that has the form `@filename` will be replaced with the contents of the named file. The contents should be a list of arguments, with one argument per line. This argument processing is done recursively, so that some of these arguments may also have the form `@file`.

Fairly obviously, a file named in an `@file` argument should never include itself, either directly or indirectly. The compiler will generate an error message for any file that violates this rule.

Since `@filename` arguments are just shorthand for a list of other arguments, in the rest of this user guide we will usually classify all command line arguments as either option or non-option arguments, with the latter including only file names and module names (in the absence of the `--make` option).

2.1.3 File name and module name arguments

In the absence of the `--make` option, non-option arguments can be either file names (if they end in `.m`) or module names (if they don't).

The compiler converts module names to file names in one of two ways. In the absence of a file named `Mercury.modules` in the current directory, it relies on each module being stored in a file whose name is just the module name followed by `.m`. For example, given a module named e.g. `foo.bar.baz`, the compiler expects to find it in `foo.bar.baz.m`.

If any module of the program does not meet this expectation, then the user must create a `Mercury.modules` file, which contains a map from module names to file names. This can be created using a command such as `mmc -f *.m` if all the modules of the program are in the current directory (see [Section 11.4 \[Options that give the compiler its overall task\], page 114](#)). In the presence of the `Mercury.modules` file, the compiler will of course get the file name corresponding to a module name by looking up the module name in this file.

2.2 Compiling single-module programs

Given a Mercury program that consists of a single module, which is stored in a file named e.g. `prog.m`, you can compile it using the command

```
mmc prog.m
```

or, more generally,

```
mmc option ... prog.m
```

with one concrete example being

```
mmc --inhibit-warnings -O3 prog.m
```

If the single module inside ‘prog.m’ is also named ‘prog’, then you can also compile it using

```
mmc option ... prog
```

which specifies the module name, not the (matching) file name.

2.3 Compiling multi-module programs

Compiling a program that consists of more than one module is a more complex task than compiling a program that consists of just one module, because it requires managing the connections between modules, and it requires minimizing the number of files needing to be recompiled after the user changes some but not all of the modules in the program.

2.3.1 Managing connections between modules

A Mercury module’s source file contains both the interface and the implementation sections of that module. Given a module *A* which imports module *B*, when compiling module *A*, the compiler needs to know what entities (types, predicates, functions etc) module *B* exports. It could do that by reading module *B*’s source file and extracting its interface part. However, given that the interface of a module tends to be much more stable than its implementation, that would be incredibly wasteful: it would compute *B*’s interface, many, many times, getting the same result between each pair of changes to *B*’s interface. It is far more efficient to store *B*’s interface in a file, which only ever needs to be updated when *B*’s source module changes, and does so in a way that *affects its interface*.

In C and in some programming languages based on it, hand-written header files (‘.h’ files) usually store the interface of the source file (‘.c’ file) with the same base name. Mercury programming uses a similar file, the ‘.int’ file, with the difference being that a module’s ‘.int’ file is *not* hand-written, but is derived from the module’s source file by the Mercury compiler.

In fact, due to Mercury’s module system being substantially more expressive than C’s (not hard when C does not actually *have* an explicit module system), each Mercury module has three or four *interface* files, not just one.

- ‘.int’ files play the same role as C’s ‘.h’ files: they list the entities (types, insts, modes, typeclasses, instances, predicates, functions, and some others) that this module makes available to the other modules that import it.
- ‘.int2’ files contain a subset of the information in ‘.int’ files, the subset which the compiler needs from indirectly imported modules. (If module *A* imports module *B* and module *B* imports module *C*, but module *A* contains no `:- import_module` or `:- use_module` declaration for module *C*, then module *A* imports module *C* *indirectly*.)
- ‘.int3’ files contain just the names of the types, insts, modes, typeclasses and instances exported by the module. The compiler needs these files to construct ‘A.int’ and ‘A.int2’ in the frequent case that module *A*’s interface uses a type named *foo* (maybe as the type of a predicate argument) that it does not define. If module *A*’s interface imports two modules, *B* and *C*, the compiler needs to know for each of those modules whether they define a type named *foo*, so it can put either *B.foo* or *C.foo* as the type of that argument into e.g. ‘A.int’ if just one of them defines it, and so that it can generate an error if either neither or both define it.

- ‘.int0’ files contain the declarations of the entities that a module that includes submodules exports *just* to those submodules, and to nowhere else. This includes entities (such as predicates and functions) that are declared in the implementation section of the parent module. Only parent modules, i.e. modules that include submodules, have ‘.int0’ files; non-parent modules do not.

As you may guess from that description, managing Mercury interface files is not trivial. It is not hard either, as shown by [Chapter 5 \[Compilation details\], page 19](#), but it is even simpler to leave their management to automatic build tools. These build tools also take care of another issue: separate compilation.

When C programmers update a header file, they must also recompile all the C source files that depend on that header file. Manually keeping track of *which* source files these are is possible in theory, but very tedious, and extremely error-prone. This is even more true for Mercury. Yet the one simple but guaranteed-to-work method, recompiling everything, can slow down the think-edit-compile-test cycle to an unreasonable degree. Unlike humans, automatic build tools can be trusted to recompile only the modules that need recompilation, reusing the results of previous compilations wherever possible.

The Mercury implementation supports two automatic build tools: `mmake`, and `mmc --make`. As their names imply, both are intended to work like the traditional Unix build tool `make`. The next two subsections describe them in turn. They each have strengths and weaknesses (for example, `mmake` supports only compilation to C, not to Java or C#), and they support different subsets of the available functionality, though the overlap (the set of features they both support) is large. We recommend that projects using Mercury pick one, and use it consistently.

If you can, you should pick `mmc --make`, because this is the build tool that is likely to receive more development in the future.

You can also use both tools, each for different parts of the same project, such as using `mmc --make` for the Mercury parts and `mmake` for tying those parts to the non-Mercury parts, but in general, you should do this only if you have to.

2.3.2 Introduction to mmake

The Mercury build tool that was implemented first is `mmake`, whose name is short for “Mercury make”. If you have a Mercury program containing three modules, maybe called ‘`main_module.m`’, ‘`module_a.m`’ and ‘`module_b.m`’, with the `main` predicate being in ‘`main_module.m`’, you can use these commands to compile a program:

```
mmc -f *.m
mmake main_module.depend
mmake main_module
```

The first step creates the ‘`Mercury.modules`’ file that maps the name of each module inside each Mercury source file in the current directory to the name of the file containing it. It is not needed if all the modules are stored in files whose name is the module name plus the ‘`.m`’ suffix.

The second step creates some small files containing makefile fragments. These small files will include ‘`main_module.dep`’ and ‘`main_module.dv`’, which record whole-program-level information, such as the list of modules in the program, and the intended executable name. In this case, that name will be ‘`main_module`’, either without an extension, if the platform

does not require one for executables, or with the required extension (such as `.exe` on Windows).

The second step will also create one file with the suffix `.d` for each module in the program, with e.g. `module_a.d` recording which other modules `module_a.m` imports.

One important aspect of `.d` files is that every compilation of a Mercury module will implicitly update that module's `.d` file if its old contents are not up-to-date. This is not true for `.dep` and `.dv` files; those are updated only if you explicitly ask for it, e.g. via the second step command above.

The third step then uses those small files to create all the necessary interface files in the proper order, to compile each Mercury module, and to then link the resulting object files together to yield the desired executable.

In outline, `mmake` works by concatenating the set of makefile rules for Mercury compilation built into it, all the makefile fragments in `.dep`, `.dv` and `.d` files in the current directory, and the file named `Mmakefile` if it exists, and invoking `gmake` on it, passing it the target or targets that it itself was given. (`gmake` is the GNU version of the standard Unix `make` program, though on many platforms it is installed as just plain `make`.)

This close relationship to `gmake` allows `mmake` to be used to control not just the compilation of Mercury code, but also the building of any other file, provided only that one can write makefile rules for their construction.

Many small Mercury projects don't really need an `Mmakefile`, but they have one anyway, usually looking something like this:

```
PROGRAM_NAME = prog

.PHONY: default_target
default_target: $(PROGRAM_NAME)

.PHONY: depend
depend: Mercury.modules $(PROGRAM_NAME).depend

Mercury.modules: $(wildcard *.m)
    mmc -f $(wildcard *.m)

.PHONY: install
install:
    test -d $(INSTALL_DIR) || mkdir -p $(INSTALL_DIR)
    cp $(PROGRAM_NAME) $(INSTALL_DIR)

.PHONY: clean
clean: $(PROGRAM_NAME).clean
```

If you have an `Mmakefile` like this, you can then type just

```
mmake depend
```

instead of

```
mmc -f *.m
mmake main_module.depend
```

and you can type just

```

mmake
instead of
mmake main_module

```

And obviously, the ‘Mmakefile’ also shortens the command you need to give to install the program, or to clean up the directory by deleting the automatically regenerable files.

For more information on how to use this tool, please see [Chapter 7 \[Using Mmake\]](#), page 28.

2.3.3 Introduction to `mmc --make`

The second Mercury build tool is `mmc --make`, which, as the name says, integrates the functionality of a make-like build tool into the Mercury compiler.

Like `mmake`, `mmc --make` needs access to the module-name-to-file-name map in ‘Mercury.modules’ if some module in the program is stored in a file whose name does not match the module name. This means that the first step in building an executable is still usually `mmc -f *.m`. Unlike `mmake`, when `mmc --make` is asked to build an executable for a program given the name of its main module, it can itself find out the dependencies (importer-imported relationships) between the modules of the program; it does not need previously-built files containing makefile fragments to give it this information. It therefore has no need for any equivalent of the `mmake main_module.depend` command.

This allows an executable to be built with just these two commands:

```

mmc -f *.m
mmc --make main_module

```

In this case, the `main_module` part of the second command is a file name: specifically, the name of the executable to be built. However, this non-option argument to `mmc --make` is not the name of a file to be *compiled*; it is the name of a file to be *built*. In general, non-option arguments to `mmc --make` should be the names of *targets*, in the same sense as `make` targets. Most targets are files to be built, but some are not, the same way as PHONY targets in `make` do not represent files. For example, `mmc --make main_module.realclean` asks the compiler to delete all the automatically-regenerable files that are part of the `main_module` program.

Note that `mmc --make` has a subtle advantage over `mmake`. While `mmake` uses (because it has to) module-to-module dependency information that was current when the relevant modules were last (re)compiled (because that is when their ‘.d’ files were last updated), `mmc --make` always uses *current* dependency information, because it gets that information from the current versions of the modules’ source files.

Most of the time, the difference does not matter, for one of several reasons:

- because most changes to Mercury modules do not affect their `:- import_module` and `:- use_module` declarations;
- because deleting an `:- import_module` or `:- use_module` declaration cannot lead to obsolete versions of interface files being read (since they lead to *fewer* interface files being read);
- because an `:- import_module` and `:- use_module` declaration is added, which *can* lead to obsolete versions of the interface files of the newly-imported modules being read, those interface files have a reasonable probability of being brought up to date

before being needed by the module that added the new import, due to some *other* module already depending on them; and

- because a nontrivial fraction of the time, the part of the interface file that bringing it up-to-date would change is of no relevance to the newly-importing module.

Most of the time, one or more of these mitigating circumstances will apply. However, sometimes none of them do. In such cases, using `mmake` to build the selected target will fail, with the failure being caused by `mmake`'s reliance on out-of-date dependency information. (This could mean e.g. `mmake` not knowing that it must update the `.int` file of a newly-imported module before generating target language code for the importing module.) In pretty much all of the usual scenarios, the attempt to build the target will cause the out-of-date dependency information to be updated, so the *next* attempt to build that same target will succeed. Nevertheless, the failure of the first attempt is annoying.

Such transient failures won't happen with `mmc --make`, because it *always* works with up-to-date dependency information.

For more information on `mmc --make`, please see *ZZZ*

3 Mercury grades

3.1 The Mercury backends

The Mercury compiler has two backends.

The first backend implemented by the Mercury team translates Mercury programs to what is effectively assembly-level code in C syntax. Most compilers generate assembly level code (either assembly code itself or binary machine code), so the first part is quite conventional. The second part, generating this assembly level code in C syntax, was rare at the time, though it has become more common since. We chose that approach because it allowed us to take advantage of the huge amount of work put into C compilers, both in terms of making C available on pretty much all commonly used platforms, and generating for them not just code that works, but *fast* code.

Like most compilers, the Mercury compiler has representations for both the source code and the target code. In compilers for imperative languages, these are usually called the abstract syntax tree (AST) and the intermediate representation (IR) respectively. The Mercury compiler's versions of these are substantially different, so we use different names for them: the high-level data structure (HLDS) and the low-level data structure (LLDS).

The low-level code generated by the original backend is about as far from the code that a human C programmer would write as it is possible to get. Later on, as part of a research project, we investigated translating Mercury code into *idiomatic* C code, with a view towards making the approach general enough to be able to produce idiomatic code in other imperative programming languages as well. We call the compiler's internal representation of this kind of code the medium-level data structure (MLDS), since it is clearly lower level than Mercury code, but higher level than assembly code.

From the names of these internal representations, Mercury calls the first backend the LLDS backend, and the second backend the MLDS backend. The first generates low (assembly) level C code, while the second generates either high level C code, Java code, or C# code.

3.2 The importance of consistency

In general, a Mercury program consists of many modules. These modules must be compiled in a manner that allows the resulting files codes to be linked together. For example if you compile e.g. module *module_a* to ‘*module_a.java*’ and module *module_b* to ‘*module_b.c*’, which the Java and C implementations compile further to ‘*module_a.class*’ and ‘*module_b.o*’. This is not just because JAR files and object files have different formats; the more fundamental reason is that the codes in them make fundamentally different assumptions about how the different modules of a program are supposed to communicate and cooperate with each other. This is why if you compile both module *module_a* and module *module_b* to C, but compile module *module_a* using the LLDS backend and *module_b* using the MLDS backend, any attempt to link ‘*module_a.o*’ and ‘*module_b.o*’ will still fail. It will fail because the two backends use different naming schemes for the C code they generate, so that the name of the C function that implements e.g. the predicate *module_b.q* will be different from the name by which a call from *module_a.p* will try to refer to *module_b.q*. This difference is deliberate. This is because any attempt to “cure” such link failures by having the two backends use the same C naming scheme would address only *incidental* incompatibilities; the *fundamental* incompatibilities, such as the LLDS backend managing its own stacks (two of them) while the MLDS backend relies on the standard C call stack, would still remain.

The `mmc` option ‘`--target`’ controls which of its target languages the Mercury compiler will generate code for. With ‘`--target c`’, Mercury will generate C code; with ‘`--target java`’, Mercury will generate Java code; and with ‘`--target csharp`’, Mercury will generate C# code. When generating C, the option ‘`--high-level-code`’ controls whether code generation will use the LLDS or the MLDS backend, generating assembly code in C syntax or idiomatic C code respectively. (Since you cannot write assembly-level code in either Java or C#, the compiler will always use the MLDS backend when generating Java or C# target code.) What the previous paragraph is saying is that if you compile all the modules of a program, an attempt to link the resulting files together can succeed *only if* all the modules were compiled using the exact same values of ‘`--target`’ and ‘`--high-level-code`’ options.

These are not the only options that have this property. There are other options as well, and in fact Mercury has more such options than most other languages. This is why unlike most other languages, Mercury has an explicit name for this concept: it calls each set of compatible values of those options a *grade*.

A Mercury grade succinctly specifies the value of roughly twenty options (depending on how exactly you count them, and when, since we add new ones from time to time). Each grade consists of a *base grade* (which must be present) followed by zero or more other *grade modifiers* (which are therefore optional). The names of the grade modifiers all start with a period; the names of the base grades do not. The name of a grade is a concatenation of the selected base grade and the selected grade modifiers (the grade modifiers may be given in any order). For example, `hlc.tr.gc` specifies the base grade `hlc` (meaning high level C code) and the grade modifiers `.tr` and `.gc` (which respectively call for the use of trailing and of the Boehm-Demers-Weiser garbage collector).

3.3 Base grades

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. There are three MLDS and three LLDS base grades.

The available MLDS base grades, and the option values they correspond to, are

```
hlc      '--target c' and '--high-level-code'
java    '--target java' and '--high-level-code'
csharp  '--target csharp' and '--high-level-code'
```

These base grades each call for the generation of relatively natural-looking code in the selected target language.

The available LLDS base grades, and the option values they correspond to, are

```
none    '--target c', '--no-high-level-code', '--no-gcc-global-registers', and
        '--no-gcc-non-local-gotos'
reg     '--target c', '--no-high-level-code', '--gcc-global-registers', and
        '--no-gcc-non-local-gotos'
asm_fast '--target c', '--no-high-level-code', '--gcc-global-registers',
        '--gcc-non-local-gotos' and '--asm-labels'
```

These base grades each call for the generation of assembly-like code in C syntax, but they differ in what GNU extensions to C, if any, the generated code will use. There are three extensions that they can potentially use:

`'--gcc-global-registers'`

specifies the use of a GNU extension that allows the generated code to tell `gcc` that a given global variable should be stored in a machine register. The Mercury implementation can use this mechanism to speed up access to the most frequently used virtual registers of the Mercury abstract machine, such as the return address register (which is used on every call) and the pointer to the top of the det stack (which is used at the start and end of the code of all predicates that cannot succeed more than once).

`'--gcc-non-local-gotos'`

specifies the use of a GNU extension that allows the generated code to simply take the address of a C label, store that address somewhere, and to later retrieve that address and jump to it. This speeds up transfers of control (conditional branches and unconditional jumps) within a single C function.

`'--asm-labels'`

specifies the use of a GNU extension that allows the address of a label to be made global, i.e. accessible from even outside the C function in which it occurs. With the appropriate precautions, which the Mercury system of course takes, this speeds up transfers of control from anywhere to anywhere else in the Mercury program.

The base grade `none` calls for no GNU extensions to be used. It is therefore the slowest of the LLDS base grades, but it is also the most portable. The base grade `reg` is faster

and less portable, and the base grade `asm_fast` is faster and less portable still. In general, using more GNU C extensions will make the program faster, but some platforms, compilers or compiler versions do not support specific extensions.

While it is in theory possible to combine `--gcc-non-local-gotos` with `--no-asm-labels`, in practice there is no advantage in doing so. And while it is also possible to combine `--gcc-non-local-gotos` `--asm-labels` with `--no-gcc-global-registers`, we do not know of any platforms on which `--gcc-non-local-gotos` and `--asm-labels` work but `--gcc-global-registers` does not. This is why we use only three out of the eight possible combinations of the values of these three options.

The default base grade is system dependent, but will be either `hlc` or `asm_fast`, as these are the two fastest.

3.4 Grade modifiers

Every grade modifier belongs to a *grade modifier group*. Each grade modifier group governs a single aspect of compilation. Because each modifier calls for handling that aspect differently from what the other modifiers in its group call for, a grade may contain at most one modifier from each group.

Note also that

- some grade modifiers are compatible only with some base grades (usually the ones that target C) and not others; and
- some grade modifiers in one group are incompatible not just with all other grade modifiers in the same group, but also with some grade modifiers from other groups.

Note also that the Mercury standard library will have been installed on your system in only a subset of the set of all possible grades.

Each of the following subsections except the last documents one group of grade modifiers. The last subsection documents which grade modifiers are compatible with grade modifiers from other groups.

3.4.1 Grade modifiers for debugging Mercury programs

In the absence of both of the grade modifiers in this group, the Mercury compiler generates executables that cannot be debugged, because they do not include the information needed for the operation of the Mercury debugger, whose name is `mdb` [Chapter 9 \[Debugging\]](#), [page 39](#).

`.debug`

`.decldebug`

Compiling programs in a grade that includes either the `.debug` or the `.decldebug` grade modifier generates executables that can be debugged using `mdb`. Invoking `mdb` on such executables lets users do the kinds of things that people do with C debuggers such as `gdb` (set breakpoints, step forward e.g. one call at a time, and print the values of variables, amongst others), as well as the kinds of things that people do with debug mode in Prolog. (such as jumping back from the end of a call to its beginning). The difference between

them is that the declarative debugging aspects of ‘`mdb`’ will work only with ‘`.decldebug`’. While both `.debug` and `.decldebug` increase the size of the executable and reduce its speed, both the size increase and the speed reduction are larger with `.decldebug` than with `.debug`.

Both of these grade modifiers are compatible only with LLDS base grades.

3.4.2 Grade modifiers for profiling Mercury programs

In the absence of all of the grade modifiers in this group, the Mercury compiler generates executables that cannot be profiled. Mercury actually supports two completely separate profilers: `mprof` and `mdprof`.

`mprof` is effectively a Mercury clone of one of the standard profilers for C, namely `gprof`. Like many (if not most) profilers, it operates by recording what part of the program was executing when its process is sent a periodic profiling signal by the OS. Like `gprof`, at each profiling signal, it records what predicate it was executing at the time and where it was called from, but no more than that.

This works well for C, where each function typically has one job, but not for Mercury, which has many polymorphic predicates that can be used in the implementation of many different jobs, with many different performance behaviors. This is why Mercury also has `mdprof`, the Mercury *deep* profiler, which at each profiling signal records the identity of not just the immediate parent of the currently executing predicate, but effectively the identities of *all* its other ancestors as well. (This is the “deep context” after which the profiler is named.)

The two profilers operate differently and have different strengths and weaknesses. You can read more about both of them in [Chapter 10 \[Profiling\]](#), page 105.

.prof Compiling programs in a grade that includes the `.prof` grade modifier generates executables that, when executed, generate files containing information that `mprof`, the `gprof`-style profiler, can use to tell you where the program is spending its time.

This grade modifier is compatible only with base grades that target C.

.memprof Compiling programs in a grade that includes the `.memprof` grade modifier generates executables that, when executed, generate files containing information that `mprof`, the `gprof`-style profiler, can use to tell you where the program is allocating memory.

This grade modifier is compatible only with base grades that target C.

.profdeep Compiling programs in a grade that includes the `.profdeep` grade modifier generates executables that, when executed, generate files containing information that `mdprof`, the Mercury deep profiler, can use to tell you both where the program is spending its time and where it is allocating memory.

This grade modifier is compatible only with LLDS base grades (which all target C).

3.4.3 Grade modifiers for trailing

In the absence of the `.tr` grade modifier, the Mercury compiler generates executables that do not support trailing.

Trailing is a technique that the implementations of logic programming languages use to support constraint solving. All Prolog implementations use it, because unification in standard Prolog is effectively a solver for constraints of the form “are these two terms unifiable?”. (Such constraints are usually called *Herbrand* constraints.) In Mercury, the mode system imposes a requirement that when two terms are unified, one of them must be ground, which means that such questions are always trivially answerable. In fact, the design of Mercury was driven in large part by the intention to *avoid* any requirement for constraint solving, because it makes programs harder to understand both by humans, and by the compiler. Unlike Prolog, Mercury thus has no constraint solver built into it. However, Mercury *can* be used to *implement* constraint solvers. While the act of posting a constraint in such a program is a declarative action, its implementation requires updating a constraint store. This in turn requires that when execution backtracks past the goal that posted the constraint, the update to the constraint store must be undone.

This is why in the right grades, Mercury can support a trail. (see [Section “Trailing” in *The Mercury Language Reference Manual*](#)). The trail is a stack-like data structure that consists of a sequence of entries, each of which effectively describes how to undo an update to a data structure. This can be used to implement constraint solvers like this:

- When execution is about to enter a disjunction or an if-then-else, the implementation records the address of the top of the trail, effectively as a snapshot.
- When the program adds a new constraint to the constraint store, it must also push a new entry on top of the trail that describes how to undo the addition of this new constraint. (That entry will consist of the address of the undo function, and an argument for it that identifies the specific addition to undo.)
- Whenever execution backtracks, either to a later disjunct in a disjunction or to the else part of an if-then-else, the implementation walks the trail from the current top of the trail down to the snapshot address that was recorded on entry to the current disjunction or if-then-else, and executes the undo actions that they encode. (For example, a trail entry can contain the address of a C function to call, and an argument to invoke that function with.) This is usually called *unwinding* the trail.

`.tr` Compiling programs in a grade that includes the `.tr` grade modifier generates executables that support trailing. This means that the compiler-generated code will create the snapshots and unwind the trail as needed. The addition of new constraints to the constraint store, and the addition of new entries to the trail to undo those additions, will be done by C code in `foreign_proc` pragmas written by the user. (Each trail entry consists of the address of a C function to call, and an argument to invoke that function with.)

This grade modifier is compatible only with base grades that target C.

3.4.4 Grade modifiers for parallelism

In the absence of the `.par` grade modifier, the Mercury compiler generates executables that do not support parallel execution in any form.

`.par` Compiling programs in a grade that includes the `.par` grade modifier generates executables that support parallel execution. What form of parallel execution is supported depends on the backend.

- With the MLDS backend, `.par` grades support user-managed parallelism, using whatever thread support is provided by the target language. Programmers can use the operations of the `thread` module of the Mercury standard library to spawn new threads, and to collect their results once they are done.
- With the LLDS backend, `.par` grades support compiler-managed parallelism. Programmers can ask the compiler to execute two goals in parallel simply by replacing the comma (the sequential conjunction operator) between them with an ampersand ‘&’, the parallel conjunction operator. The compiler will take care of the rest. (Note that ‘&’ has this effect *only* in LLDS `.par` grades; in all other grades, the compiler silently converts all ‘&’s into commas.)

Note: due to a code generation bug, at the moment LLDS parallelism does not work.

3.4.5 Grade modifiers for minimal model tabling

Mercury supports several forms of tabling, most of which must be explicitly enabled for each procedure that they are intended to apply to (see [Section “Tabled evaluation” in *The Mercury language Reference Manual*](#)). (Recall that a *procedure* is one mode of a predicate or function.) Tabling operates by recording (in a table, hence the name), for each vector of input values to a procedure, the set of vectors of output values the procedure returns as results. (For `det` code, the set will contain exactly one vector; for `semidet` code, the set will contain at most one vector.) For later calls to a tabled procedure, the Mercury implementation will automatically check whether the current vector of values of the input arguments has occurred before. If it has, it will return the recorded set of answers. If it has not, it will compute the result or results, and record it or them.

This system has a problem if the tabled procedure calls itself recursively, either directly or indirectly, with the exact same vector of input values, *before* its initial invocation has completed, because in that case, there will be an entry recording the input vector, but the corresponding set of output value vectors will not yet be available. With the most frequently used form of tabling, i.e. *memoization*, this is a fatal error, and the Mercury implementation will automatically abort the program. However, for `nondet` predicates, there is a way to avoid this abort, at the cost of a much more complicated execution model.

This solution gets its name, *minimal model tabling*, from the fact that it is intended to return, for each query, all the answers that match that query in a specific *minimal model* of the program, named the *perfect model*. (We call it “minimal model tabling” instead of “perfect model tabling” because the former is the standard terminology for this kind of tabling in the logic programming literature.)

Minimal model tabling consists of

- recording the state of the computation at the point of the recursive call;
- continuing execution as if the call failed;

- when the top-level call to the procedure with the given vector of input argument values finds a new solution, *restoring* the recorded state of the computation, and restarting its execution, but this time behaving as if the recursive call just returned the new solution.
- This restarted execution may obviously return even more solutions, which have to be fed back to the recursive call (or in general, the recursive calls, plural). The process ends when all restarted calls lead either to no solution, or to solutions that have all previously been not just recorded, but also fed back to all recursive call sites.

The execution mechanisms required for this are quite complicated, and impose significant overheads in both time and space, even on the parts of the program that do not benefit from it. This is why in the absence of the following grade modifiers, the Mercury compiler generates executables that do not support minimal model tabling.

- .mm** Compiling programs in a grade that includes the `.mm` grade modifier generates executables that support minimal model tabling.
This grade modifier is compatible only with LLDS base grades.
- .mmsc** The grade modifier `.mmsc` is a synonym for `.mm`, standing for “minimal model via stack copying”, since the standard implementation works by copying parts of the stack. 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.

3.4.6 Grade modifiers for flexible stack sizes

In MLDS grades, the code generated by the Mercury compiler uses the native call stack of the target language. In LLDS grades, on the other hand, the generated code manages its own two stacks: the *det stack*, which stores the data of procedures that *cannot* succeed more than once, and the *nondet stack*, which stores the data of procedures that *can* succeed more than once. (The two stacks are separate because for predicates that can succeed more than once, we cannot throw away the values of their input arguments and local variables when they succeed, while for predicates that cannot succeed more than once, we can throw them away, and for good performance, we *must* do so.)

By default, Mercury reserves a fixed-size block of memory for each stack, and never increases their size during execution. To make this work even for programs that use relatively deep recursion, the default size of the det stack is 16 megabytes on 32-bit machines, and 32 megabytes on 64-bit machines. These default sizes can be overridden using runtime options (see [Section 12.3 \[Environment variables affecting the Mercury runtime system\], page 168](#)), but whatever pair of sizes one picks, for some programs, they will be wasteful overkill, while for some other programs, hopefully rarely, they nevertheless will not be enough. If the platform allows it, we make the top page of each stack 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.

We therefore have a grade modifier that makes the sizes of the two stacks dynamic, and able to respond to the actual requirements of the program being executed.

- .stseg** Compiling programs in a grade that includes the `.stseg` grade modifier generates executables that use *stack segments*, small memory areas whose size is

16 or 32 kilobytes on 32- and 64-bit systems respectively. In `.stseg` grades, the `det` and `nondet` stacks both consist of a linked list of one or more stack segments. Each stack grows by adding a new segment at a time on demand, and shrinks by putting segments back on a free list once they are no longer needed.

Using stack segments does have a small performance penalty, because it requires code for checking

- whether the creation of a new stack frame requires the allocation of a new stack segment and adding it to the linked list, and
- whether a stack frame being popped off the stack was the last frame in its segment, allowing the segment to be removed from the stack's linked list, and made available for future allocations.

However, for many applications, this is a more than acceptable price to pay for not imposing any limit on stack sizes other than the amount of available memory.

This grade modifier is relevant only for LLDS base grades, and is compatible only with them.

3.4.7 Grade modifiers for garbage collection

By default, the Mercury system provides 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` usually include a grade modifier that specifies which garbage collector to use.

`.gc` Compiling programs in a grade that includes the `.gc` grade modifier generates executables that use the standard Mercury garbage collector, which is the Boehm-Demers-Weiser conservative collector for `C`.

This grade modifier is relevant only for base grades that target `C`, and is compatible only with them.

The reason why this is an explicit grade modifier is that over the years, we have tried out several collectors for `C`. These had their own grade modifiers, to allow users to specify their use. However, only the Boehm-Demers-Weiser collector has stood the test of time.

3.4.8 Grade modifiers for single precision floats

Many imperative languages have at least two floating point types, one single precision and one double precision, which usually means they are 32 bits and 64 bits in size respectively. Mercury has only one floating point type, `float`, which by default is implemented as an IEEE 754 double-precision 64-bit floating point number.

When targeting `Java` or `C#`, this default cannot be overridden: values of the Mercury type `float` are always represented by values of type `double` (which exists in both those languages). When targeting `C`, the default *can* be overridden by using the following grade modifier.

`.spf` Compiling programs in a grade that includes the `.spf` grade modifier (which stands for “single-precision float”) generates executables that represent values of the Mercury `float` type as 32-bit values of C’s `float` type.

This grade modifier was intended to be used on 32-bit platforms. On those platforms, 64-bit floats are twice the size of a machine word, which means that they cannot be stored the same way as all other values, and must instead be stored on the heap. This means that every operation that generates a new 64-bit floating point value must allocate a heap cell for it, and write those 64 bits to that cell. (This is usually called *boxing* the 64-bit value.) Representing Mercury `floats` as 32-bit single-precision values avoids this overhead, improving both memory consumption and speed. This made this grade modifier useful for programs that did not require the extra precision (or range) offered using 64-bits.

`.spf` grades provide no performance advantage at all on 64-bit platforms, because on those platforms, 64-bit values do not require boxing. On the other hand, they may simplify the use of C APIs that exclusively use single precision floats.

This grade modifier is relevant only for base grades that target C, and is compatible only with them.

3.4.9 Grade modifiers for target language debugging

Mercury programs are intended to be debugged using `mdb`, the Mercury debugger. However, `mdb` treats non-Mercury code as a black box, and cannot give any insights into its behavior.

`.target_debug`

Compiling programs in a grade that includes the `.target_debug` grade modifier generates executables that are intended to be debuggable with usual debuggers for the target language selected by the base grade.

This grade modifier is intended mainly to help the implementors of Mercury itself debug interactions between compiler-generated target code and the Mercury runtime system. However, in certain rare cases, it may also be useful to other users in debugging interactions between compiler-generated target code and the contents of `foreign_proc` and/or `foreign_code` pragmas.

This grade modifier could be applicable to all base grades, but is intentionally restricted to MLDS grades. This is because in LLDS grades, the assembler-like C code generated by `mmc` will probably confuse debuggers such as `gdb`, and it will definitely confuse any Mercury programmer who is not a Mercury implementor. The more idiomatic target language code that `mmc` generates in MLDS grades is still far from trivial for non-implementors to understand, but at least they have a fighting chance.

3.4.10 Grade modifiers for pregenerated source distributions

There is one grade modifier that users will probably encounter that they will probably never need to use themselves. This is the grade modifier that the Mercury team itself uses only for a single purpose: making source distributions. Such distributions include not just the code of the Mercury system, which is mostly written in Mercury itself, but also the C code

generated for each Mercury module. This allows people who do not have a working Mercury installation on their machine to create one.

`.pregen` Compiling programs in a grade that includes the `.pregen` grade modifier switches off all optional features, and tells the compiler to make the fewest possible assumptions about the platform on which the generated code will run. (For example, the generated code will work on both 32-bit and 64-bit platforms.)

Source distributions use grades that include this modifier (the grade usually being `hlc.gc.pregen`) because it makes the C code in them portable to as many platforms as possible. However, it has no advantages beyond this, and, on 64-bit machines, the requirement to use a data representation scheme that also works on 32-bit platforms imposes a nontrivial cost in performance. This is why for pretty much any purpose other than source distributions, other grades are a better choice.

This grade modifier is applicable only to the base grades `hlc` and `none`.

3.4.11 Compatibility of grade modifiers

ZZZ TODO

4 Running Mercury programs

ZZZ TODO

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.md'` 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'`. It will also generate a launcher that invokes the program using the Java interpreter. If you are using the Windows command line interpreter `'cmd.exe'`, this launcher will be a batch file called `'filename.bat'`. Otherwise, the launcher will be a shell script called `'filename'`. Java runtime flags can be set using `mmc`'s `'--java-runtime-flags'` or `'--java-runtime-flag'` options. Such Java runtime flags will be included in the generated launcher shell script or batch file. You may override any runtime flags set at (Mercury) compile time by setting the variable `MERCURY_JAVA_OPTIONS` in the environment. Classpath settings made using `mmc`'s `'--java-classpath'` option will also be included in the generated launcher shell script or batch file.

5 The Mercury compilation process in detail

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 this chapter.

5.1 Creating interface files

The first step in building a multi-module Mercury program is creating files for each module that describe the interface that this module presents to all the other modules of the program. (A reminder: the roles and contents of these interface files were described in [Section 2.3.1 \[Managing connections between modules\]](#), page 4.)

You can create the interface files for one or more source files using the following commands:

```
mmc --make-short-interface module_1.m module_2.m ...
mmc --make-private-interface module_1.m module_2.m ...
mmc --make-interface module_1.m module_2.m ...
```

- The first command builds (or rebuilds) the ‘`.int3`’ file of each module contained in the named source files. (*Not* just the top module of each source file.)
- 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.

There are constraints on the order in which these interface files can be built.

- Each ‘`.int3`’ file depends only on the source code of the module it is for. Since ‘`.int0`’, ‘`.int`’ and ‘`.int2`’ files all depend on ‘`.int3`’ files, the building process must start off by building ‘`.int3`’ files. The order in which they are built does not matter.
- Each ‘`.int0`’ file depends only on the ‘`.int3`’ files that the module it is for imports, directly or indirectly, *and* on the ‘`.int0`’ files of its ancestor modules (if any). Therefore the ‘`.int0`’ files of the program must be built next, with the ‘`.int0`’ files of ancestor modules being built before the ‘`.int0`’ files of their descendant modules.
- Each ‘`.int`’ file, and its specialized ‘`.int2`’ version, depends on the ‘`.int3`’ files that the module it is for imports, directly or indirectly, and on the ‘`.int0`’ files of its ancestor modules (if any). Therefore these files will in general be the last interface files built. The order in which they are built does not matter.

As the description above shows, interface files will in general be built in an order with the ‘`.int3`’ files first, then the ‘`.int0`’ files, and then the ‘`.int`’/‘`.int2`’ files. However, there is no hard-and-fast separation between these phases. For example, if neither *module_a* nor any of its ancestors import *module_z* either directly or indirectly, then it is ok

- to build ‘`module_a.int0`’ before ‘`module_z.int3`’, and
- to build ‘`module_a.int`’ before ‘`module_z.int0`’ or ‘`module_z.int3`’.

5.2 Creating optimization files

Intermodule optimization

By default, when `mmc` compiles a module, say *module_a*, the only code it has access to is the code of *module_a* itself. The only source of information that `mmc` has about the modules imported by *module_a* are their `.int` files. Beyond the definitions of types, insts and modes, these contain the *declarations* of predicates and functions, but not their *definitions*.

However, the compiler's usual optimizations could do a better job if they *did* have access to the definitions of those predicates and functions. This is why the Mercury compiler has a mechanism for providing that access. This mechanism, intermodule optimization, has two faces: recording extra information about the nominally-private parts of each module in a file, and making use of that information while compiling other modules.

Commands that do the first part look like this:

```
mmc --make-optimization-interface module_1.m module_2.m ...
```

This command will build `module_1.opt`, `module_2.opt`, and generally the `.opt` file of each named module. These files contain information that is normally private to the module that the `.opt` file is for, but which may be useful for the optimization of other modules. Mostly, this includes the definitions (i.e. the code) of both public and private predicates and functions of the module, if those definitions match one or more from a list of criteria, which include (but are not limited to) the following.

- Predicates and function definitions that are so simple that inlining calls to them (meaning replacing the call with an appropriately-renamed copy of the callee's definition) is likely to result in a speedup.
- Predicates and function definitions that contain switches on the values of arguments, meaning that after inlining calls to them at call sites that know the values of those arguments, the switch can be eliminated.
- Predicates and function definitions that have higher-order arguments, meaning that after inlining calls to them at call sites that know the values of those higher order arguments, the higher-order calls in the inlined version can be replaced by first order calls.

Beside such code, `.opt` files also contain definitions and declarations needed to make sense of that code, such as the definitions of the types, insts and modes they involve, and the declarations of both the predicates and functions they define, and the predicates and functions they call.

After `.opt` files have been created, any invocation of `mmc` to compile say *module_1* with the `--intermodule-optimization` option (or `--intermod-opt` for short), will read in, and use, the `.opt` files of the modules that *module_1* imports.

Transitive intermodule optimization

In some cases, when compiling e.g. *module_1*, an optimization would like access to information that is derived not just from a module that *module_1* imports, call it *module_2*, but also from modules that *module_2* imports, and they import, and so on. The information that these optimizations need is not so much the code of e.g. predicates defined in modules that *module_2* imports, but their effect on the properties of the predicates and functions of *module_2* itself.

Consider a conjunction such as

... $p(\dots)$, $q(\dots)$, $r(\dots)$, ...

where the definition of r traverses a data structure created by p . The Mercury compiler contains an optimization that can fuse two traversals into one. The optimization is called *deforestation*, because it can eliminate the intermediate data structure created by p and consumed by r , and in logic programming languages, data structures are terms, which can be viewed as trees.

Deforestation can fuse two traversals only if they are next to each other, and in this case, the two calls to be fused are *not* next to each other. The first step is therefore to replace

... $p(\dots)$, $q(\dots)$, $r(\dots)$, ...

with

... $p(\dots)$, $r(\dots)$, $q(\dots)$, ...

However, this is safe only in certain circumstances.

One situation in which it is unsafe occurs when q is semidet, meaning it can fail, and r can throw an exception. This is because in this case, the reordering above can replace code that simply fails with code that throws an exception. This would have an *observable effect* on the execution of the program, which optimizations are not allowed to make.

To perform the above reordering, the compiler needs to know that r can never throw an exception. (It would also need to be able to rule out other situations that could cause the reordering to have an observable effect, but in this example, we are focusing on just this one.) For this, it needs to know not just that the code of r (which must be available if we are considering fusing it with the code of p) contains no code to throw an exception, but also that the same is true for the predicates and functions it calls, and the predicates and functions they call, directly or indirectly. In effect, we need to know that no predicate or function in the call tree of r can throw an exception.

To make this possible in at least some cases, Mercury has a mechanism to make such information available: `.trans_opt` files. These files contain analysis results, with compiler options specifying the set of analyses whose results they contain. Note that `mmc --make` does *not* support `.trans_opt` files; only `mmake` does.

One of these analyses is exception analysis, which computes safe approximations to the set of exceptions that each predicate or function can possibly throw. These approximations do not refer to the identities of specific exceptions, but they are nevertheless useful, because if this approximation is the empty set, then we know for sure that the predicate or function cannot throw any exception. (An approximation can overestimate the set of actions that the predicate or function may perform, but it is *safe* only if it will *never underestimate* that set.)

Consider a call chain between functions where f calls g , g calls h , and h calls i . with f , g , h , i being defined in `module_f`, `module_g`, `module_h` and `module_i` respectively. Suppose none of these functions contain any calls other than the ones listed here, and none of these modules contain anything else. In that case,

- to know whether i can throw exceptions, we need only the code of i ;
- to know whether h can throw exceptions, we need the code of h and the results of the analysis for i ;
- to know whether g can throw exceptions, we need the code of g and the results of the analysis for h ;

- to know whether `f` can throw exceptions, we need the code of `f` and the results of the analysis for `g`.

These dependencies transfer to the files involved:

- We record the result of exception analysis for `i` in `'module_i.trans_opt'`. Creating `'module_i.trans_opt'` does not require reading any other `'trans_opt'` files.
- We record the result of exception analysis for `h` in `'module_h.trans_opt'`. Creating `'module_h.trans_opt'` requires reading `'module_i.trans_opt'`.
- We record the result of exception analysis for `g` in `'module_g.trans_opt'`. Creating `'module_g.trans_opt'` requires reading `'module_h.trans_opt'`, which makes `'module_g.trans_opt'` depend on `'module_h.trans_opt'` directly, and on `'module_i.trans_opt'` indirectly.
- We record the result of exception analysis for `f` in `'module_f.trans_opt'`. Creating `'module_f.trans_opt'` requires reading `'module_g.trans_opt'`, which makes `'module_f.trans_opt'` depends on `'module_g.trans_opt'` directly, and on `'module_h.trans_opt'` and `'module_i.trans_opt'` indirectly.

While we can build each `'opt'` file independently of any other `'opt'` file, this is not true for `'trans_opt'` files. Not only is it the case that `'trans_opt'` files *can* depend on other `'trans_opt'` files, that in fact is the *reason for their existence*. In this case, if e.g. `'module_f.trans_opt'` records that `f` cannot throw any exception, it can do so only because `'module_i.trans_opt'`, `'module_h.trans_opt'`, and `'module_g.trans_opt'` all record that `g`, `h` and `i` respectively also have this property. And if code that can possibly throw exceptions is ever added to any one of `'module_i.m'`, `'module_h.m'`, and `'module_g.m'`, then both the `'trans_opt'` file of the updated module, and all the `'trans_opt'` files that depend on it, must all be rebuilt.

If the programmer adds a function named `f2` to `'module_f.m'` and a function named `h2` to `'module_h.m'`, with `h2` calling `f2`, then the compiler can fully analyze `h2` and put the results into `'module_h.trans_opt'` only if, when it is building `'module_h.trans_opt'`, it has access to the contents of `'module_f.trans_opt'`. However, we cannot make `'module_h.trans_opt'` depend on `'module_f.trans_opt'`. `'module_f.trans_opt'` already depends on `'module_h.trans_opt'`, because if we did that, we would make the dependency chain circular. This would mean that in order to bring e.g. `'module_h.trans_opt'` up to date, we would first have to bring `'module_f.trans_opt'` up to date, which means that we would first have to bring `'module_h.trans_opt'` up to date, which means that we would first have to bring `'module_f.trans_opt'` up to date, and so on forever. This is why software build systems, including `mmake` and `mmc --make`, require the absence of circular dependencies between files.

Mercury's first step towards avoiding circular dependencies between `'trans_opt'` files is to impose an order on the modules of the program when the `mmc --generate-dependencies` command is executed for it.

Normally `mmc --generate-dependencies` puts the parts of the order relevant to each module in that module's `'d'` file (and specifically into the `trans_opt_deps` rule in that file). However, you can ask `mmc` to put the order for the whole program (e.g. `prog`) into e.g. `'prog.module_order'` with a command such as

```
mmc --generate-dependencies --also-output-module-order prog.m.
```

After this command, `prog.module_order` will contain the strongly connected components (SCCs) of the module dependency graph of the program. The SCCs will be listed in a top-down order, with different SCCs separated by blank lines, and with the different modules in each SCC being listed one per line with no blank lines between them. The order of the modules of the program is exactly the order in which they appear in this file.

To actually construct the `.trans_opt` files for e.g. `module_1.m` and `module_2.m`, use a command such as

```
mmc --make-transitive-optimization-interface --analyse-exceptions module_1.m module_2.m
```

Beside `--analyse-exceptions`, the following options also record their results in `.trans_opt` files.

`--enable-termination`

This option turns on Mercury’s first termination analyser, which uses linear inequalities.

`--enable-termination2`

This option turns on Mercury’s second termination analyser, which uses convex constraints.

`--analyse-trail-usage`

This option turns on an analysis that identifies which predicates and functions definitely do not touch the trail, enabling the compiler to reduce the overhead of trailing. (This analysis is applicable only in trailing grades.)

`--analyse-mm-tabling`

This option turns on an analysis that identifies which predicates and functions have call trees that do not involve minimal model tabled procedures at all, enabling the compiler to reduce the overhead of minimal model tabling. (This analysis is applicable only in minimal model grades.)

The results of both termination analysis systems can be used by the user to spot potential performance problems. (That performance problem *may* be non-termination, but the analyser’s inability to prove termination may also indicate less drastic performance problems.) The compiler can also perform more optimizations on calls for which it knows that the called predicate and function always terminates.

After `.trans_opt` files containing analysis results have been created, any invocation of `mmc` to compile say `module_3` with the `--transitive-intermodule-optimization` option (or `--trans-intermod-opt` for short), will read in, and use, a subset of the `.trans_opt` files of the modules that `module_3` imports; specifically, the subset that come after `module_3` in the module order.

5.3 Creating target and object code files

Compiling Mercury programs in C grades

To compile a Mercury source file to C code, use a command such as

```
mmc --grade asm_fast.gc -C module_name.m
```

On that command line, the `--grade asm_fast.gc` part obviously specifies the grade, while the `-C` option, whose long name is `--target-code-only`, tells the compiler to stop after generating the target language file (in this case, `'module_name.c'`).

You can then compile `'module_name.c'` to `'module_name.o'` yourself, or you can tell the Mercury compiler to do this for you.

The command for doing it yourself is a command such as

```
mgnuc --grade asm_fast.gc -- -c module_name.c -o module_name.o
```

The part of the command line between the command name (`mgnuc`) and to the double dash (`--`) contains the options interpreted by `mgnuc`, while the part after the double dash contains options for the configured C compiler (usually `gcc`, `clang`, or `MSVC`).

The command that tells the Mercury compiler to generate both the `'c'` and `'o'` file of a module is

```
mmc --grade asm_fast.gc -c module_name.m
```

The `-c` option, whose long name is `--compile-only`, tells the compiler to stop after compiling the Mercury code into object code.

If the source file contains nested modules, then both the main module in the file, and all the submodules nested inside it, directly or indirectly, will all get compiled first to separate `'c'` files, and then to separate `'o'` files.

Compiling Mercury programs in Java grades

In grades that target Java, `-C` (`--target-code-only`) tells the compiler to stop after generating `'module_name.java'`:

```
mmc --grade java -C module_name.m
```

while `-c` (`--compile-only`) tells the compiler to stop after generating `'module_name.class'`:

```
mmc --grade java -c module_name.m
```

since `'class'` files are the Java equivalents of object files for C.

If the source file contains nested modules, then both the main module in the file, and all the submodules nested inside it, directly or indirectly, will all get compiled first to separate `'java'` files, and then to separate `'class'` files.

Compiling Mercury programs in C# grades

In grades that target C#, `-C` (`--target-code-only`) tells the compiler to stop after generating `'module_name.cs'`:

```
mmc --grade csharp -C module_name.m
```

Since C# does not really have any equivalent of object files, `-c` and `-C` are equivalent in C# grades.

If the source file contains nested modules, then both the main module in the file, and all the submodules nested inside it, directly or indirectly, will all get compiled to separate `'cs'` files.

5.4 Creating executables

ZZZ TODO

After you have made all the interface files, and any `.opt` and maybe `.trans_opt` files needed by your intermodule optimization options, you are ready to create an executable.

One way to create an executable for a multi-module program is to compile all the modules at the same time, using a command such as

```
mmc 'main_module.m' 'module_1.m' 'module_2.m' ...
```

where `'main_module.m'` should be the module that defines the predicate that serves as the program entry point, `main/2`. This command will put the resulting executable in the file named `'main_module'` by default, though you can use the `'-o filename'` option to specify a different name for it, if you wish.

The other way to create an executable for a multi-module program is to compile each module separately. When targeting C,

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:

```
m1 -o main_module module1.o module2.o ... main_module_init.o
```

`'m1'`, the Mercury linker, is another shell script that invokes a C compiler with options appropriate for Mercury, this time for linking. `'m1'` 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 `'m1'` 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 11 \[Invocation\]](#), page 113.

ZZZ TODO add pointer, or text, about creating libraries

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 12 \[Environment\]](#), page 167.)

In MLDS grades, stack management is the responsibility of the target language's compiler. In LLDS grades, stack management is the responsibility of 'mmc' and of the Mercury runtime system for C. 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.

6 File naming conventions

Mercury source files must have a name that ends with the '.m' extension. Each other kind of file (documented below) is identified by its own extension.

For source files, the part of the filename before the '.m' extension can be anything. It is simplest if it is the full name of the module contained in the file, because if it isn't, for any module in a program, then you will need to tell the compiler to construct a module-name-to-filename map. You can do this using a command such as `mmc -f *.m` (if all the source files in the program are in the current directory).

For all the other kinds of files mentioned in this chapter, which are always constructed automatically by the Mercury compiler, the part of the filename before the suffix will be the fully qualified name of the module that the file belongs to.

For historical reasons, the default behaviour is for these files to be created in the current directory. However, 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. (With 'mmc --make', '--use-subdirs' is the default.)

The rest of this chapter lists the extensions used by the files automatically generated by the Mercury compiler. It also briefly introduces their roles. This is just in case you are interested. You don't actually need to know anything about them; if you want, you can skip now to the next chapter.

6.1 Interface files

Files whose names end in '.int', '.int2', '.int0' and '.int3' are interface files. In the order of their usual creation,

- '.int3' files are created by invoking 'mmc' with the '--make-short-interface' (or '--make-short-int') option. Roughly, they contain basic information about the entities defined in the interface section of the module, and exported from the module to all

other modules importing this one. Their purpose is to provide the information needed for the generation of the other kinds of interface files below.

- `.int0` files are created by invoking `mmc` with the `--make-private-interface` (or `--make-priv-int`) option. They contain a list of the entities defined in the implementation section of the module that are available only to its own submodules. Modules that do not have any submodules won't have `.int0` files.
- `.int` and `.int2` are both created, at the same time, by invoking `mmc` with the `--make-interface` (or `--make-int`) option. The difference between them is that e.g. `module_a.int` is intended to contain everything that is needed by another module, such as `module_b`, that imports `module_a` *directly*, while `module_a.int2` is intended to contain what is needed by another module, such as `module_c`, that imports `module_a` *indirectly*.

6.2 Optimization files

Files whose names end in `.opt` and `.trans_opt` are files used for optimization; these are generated automatically by the compiler.

- `.opt` files are used for intermodule optimization, and are created using the `--make-optimization-interface` (or `--make-opt-int`) option.
- `.trans_opt` files are used for transitive intermodule optimization, and are created using the `--make-transitive-optimization-interface` (or `--make-trans-opt`) option.

6.3 Timestamp files

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: they each record the last time when the file or files it represents has last been checked for being up to date. (`.date` files serve as the last-time-this-was-checked timestamp for both `.int` and `.int2` files.)

The Mercury implementation uses these timestamp files when deciding whether the files they represent need to be rebuilt.

6.4 Mmakefile fragment files

The `mmake` build tool gets the information it needs about the program being built from Makefile fragments automatically generated by the Mercury compiler. These fragments come from files with the following extensions.

- `.dep` files are automatically generated Makefile fragments which contain the rules for an entire program.
- `.dv` files are automatically generated Makefile fragments which contain variable definitions for an entire program.
- `.d` files are automatically generated Makefile fragments which contain the dependencies for a module.

‘.dep’ and ‘.dv’ files are built as part of the initial setup of the program, when ‘mmc’ is invoked with ‘--generate-dependencies’ (or some related options). ‘.d’ files contain information about a single module, and they are updated (if needed) by every recompilation of that module.

6.5 Files generated when targeting C

- ‘.c’ files are C source code.
- ‘.o’ files are object code (generated by C compilers other than MSVC).
- ‘.obj’ files are object code (generated by MSVC).
- ‘.pic_o’ files are object code files that contain position-independent (PIC) code.

‘.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.

- ‘.mh’ files contain the declarations of C function versions of Mercury predicates and/or functions that are exported to C code by ‘foreign_export’ pragmas written by programmers. They are intended to be #included by the user-written C code that the ‘foreign_export’ pragmas were intended for.
- ‘.mih’ files are C header files that are part of the Mercury implementation. They are intended to be #included only by C source files that are generated automatically by the Mercury compiler.

6.6 Files generated when targeting Java

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

6.7 Files generated when targeting C#

- ‘.cs’ files are C# source code,
- ‘.dll’ files are library assemblies, and
- ‘.exe’ files are process assemblies.

7 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 intermodule optimization performed (see [Section 11.16.1 \[Overall control of optimizations\]](#), page 140).

`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.

`'libmain_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 8 \[Libraries\]](#), page 34.

`'libmain_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 11.5 \[Grade options\]](#), page 119). This target is also recommended whenever you wish to change the level of intermodule optimization performed (see [Section 11.16.1 \[Overall control of optimizations\]](#), page 140) 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'`, `'libmain_module.a'`, `'libmain_module.so'`, `'libmain_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.MemWords'` and `'Prof.MemCells'`.

`'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 11.5 \[Grade options\]](#), page 119) 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 8.2.2 \[Using installed libraries with `mmc -make`\]](#), page 35.

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 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 8.3.3 \[Using libraries with Mmake\]](#), page 38. See [Section 8.2.2 \[Using installed libraries with mmc -make\]](#), page 35.

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` (`m1` 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 8.3.3 \[Using libraries with Mmake\]](#), page 38. See [Section 8.2.2 \[Using installed libraries with mmc -make\]](#), page 35.

EXTRA_LIB_DIRS

A list of extra Mercury library directory hierarchies to search when looking for extra libraries. See [Section 8.3.3 \[Using libraries with Mmake\]](#), page 38. See [Section 8.2.2 \[Using installed libraries with mmc -make\]](#), page 35.

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.

8 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.

8.1 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.

8.2 Building with `mmc --make`

8.2.1 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.

8.2.2 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`’.

8.2.3 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 \
...

```

8.3 Building with Mmake

8.3.1 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 ‘`libfoo`’ is a builtin target for creating a library whose top-level module is ‘`foo.m`’. The automatically generated Mmake rules for the target ‘`libfoo`’ 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)`.

8.3.2 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 `libfoo`, use the `mmake` target `libfoo.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 `libfoo` 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.

8.3.3 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
```

8.4 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*’.

8.5 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 compiled to a dynamic link library assembly whose name has the form ‘*library-name.dll*’.

9 Debugging

9.1 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 9.3 \[Tracing of Mercury programs\]](#), page 42). 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 are a GNU Emacs user, 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 9.10 \[Debugger commands\]](#), page 54.

9.2 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 `.emacs` file 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 9.1 \[Quick overview\]](#), [page 39](#) or [Section 9.4 \[Preparing a program for debugging\]](#), [page 45](#). 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 breakpoint 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.

9.3 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

<i>call</i>	A call event occurs just after a procedure has been called, and control has just reached the start of the body of the procedure.
<i>exit</i>	An exit event occurs when a procedure call has succeeded, and control is about to return to its caller.
<i>redo</i>	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.
<i>fail</i>	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, each of which is followed by a semicolon. Each path component is one of the following:

<i>cnum</i>	The <i>num</i> 'th conjunct of a conjunction.
<i>dnum</i>	The <i>num</i> 'th disjunct of a disjunction.
<i>snum</i>	The <i>num</i> '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 does not 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 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.

9.4 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`, `m1`, 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 9.11.7 \[Improving the search\], page 97](#)). 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 9.11 \[Declarative debugging\], page 90](#).
- `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 9.11 \[Declarative debugging\], page 90](#).
- `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
```

9.5 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.)

9.6 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’`.

9.7 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), "²²" refers to the tail of the tail of the list, and "²²¹" 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 9.8 \[User defined events\]](#), page 52.

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.

9.8 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'.

9.9 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 9.10 \[Debugger commands\]](#), page 54 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.

9.10 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.

9.10.1 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 8 \[Libraries\]](#), page 34 for details of how to build with shared libraries.

9.10.2 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’.

9.10.3 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
```

9.10.4 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.

If the ‘-f’ option is present, it 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 draw boxes next to them to 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 term 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 [-pqx] 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 ‘-p’ (or ‘--prettyprint’) causes the goal to be output in a pretty-printed form. The option ‘-x’ (or ‘--xml’) causes the goal to be output in XML format.

`dump [-pqx] 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 ‘-p’ (or ‘--prettyprint’) causes the exception value to be output in a pretty-printed form. The option ‘-x’ (or ‘--xml’) causes the output to be in XML.

`dump [-pqx] name [termpath] filename`

`dump [-pqx] num [termpath] 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 ‘-p’ (or ‘--prettyprint’) causes the variable’s value to be output in a pretty-printed form. The option ‘-x’ (or ‘--xml’) causes the output to be in XML. If the name or number is followed by a term path such as “^2”, then only the specified subterm of the given variable is dumped.

`open term`

Save *term* to a temporary file and open the file in an editor. The name of the editor to be used is taken from the environment variable `EDITOR` if this is set;

if it is not set, then the editor will be ‘vi’. *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.

9.10.5 Breakpoint commands

**break [-PS] [-Eignore-count] [-Iignore-count] [-n] [-pprint-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 ‘-Eignore-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 ‘-Iignore-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 ‘-pprintspec’ 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] [-Eignore-count] [-Iignore-count] [-n] [-pprint-spec]*
proc-spec**

Puts a break point on the specified procedure.

The options ‘-A’ or ‘--select-all’, and ‘-0’ 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 ‘-0’ 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 ‘-Eignore-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 ‘-Iignore-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 ‘-pprintspec’ 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] [-Eignore-count] [-Iignore-count] [-n] [-pprint-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 ‘-0’ or ‘--select-one’ selects the action to be taken if the specification matches more than one procedure. If you have specified option ‘-0’ 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 `-Eignore-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 `-Iignore-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 `-pprintspec` 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] [-Eignore-count] [-Iignore-count] [-n] [-pprint-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 `-Eignore-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 `-Iignore-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 `-pprintspec` 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] [-Xignore-count] [-n] [-pprint-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 `'-Xignore-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 `'-pprintspec'` 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] [-Xignore-count] [-n] [-pprint-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 `'-Xignore-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 ‘`-pprintspec`’ 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 [-bbreak-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 [-Eignore-count] [-Iignore-count] num`

The options `'-Eignore-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 `'-Iignore-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 [-Eignore-count] [-Iignore-count]`

The options `'-Eignore-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 `'-Iignore-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.

9.10.6 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.

9.10.7 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*.

9.10.8 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.

9.10.9 Declarative debugging mdb commands

The following commands relate to the declarative debugger. See [Section 9.11 \[Declarative debugging\]](#), [page 90](#) for details.

`dd [-r] [-R] [-nnodes] [-ssearch-mode] [-ppassfile] [-ffailfile]`

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 ‘-nnodes’ 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 ‘-ssearch-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’.

9.10.10 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.

9.10.11 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 his-

togram 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 [-pfilename] [-ffilename] [-nnum] [-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 displayed text shows the goal path, except for NEGE, NEGS and NEGF events, where it contains both the goal path and the port.

- ‘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 three 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 $\text{Suspicion} = \text{Fail} / (\text{Pass} + \text{Fail})$, where Fail is the number of times the goal was executed in the failing test run, and Pass 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, users can give a separate ‘set pass_trace_counts *filename*’ command. See [Section 9.12 \[Trace counts\], page 99](#) for more information about trace counts.

The table is normally sorted on the identification of the event, meaning it is sorted

- first on the identification of the procedure containing the event (consisting of a pred/func indication, the name of the predicate or function, its arity, and its mode number),
- then on the port (if any),
- then on the path (if any),
- then on the file name, and
- then on the line number.

However, users can also ask for it to be sorted on the Pass, Fail and/or Suspicion columns, or on a combination of these. This can be done with the ‘-s’ or ‘--sort’ option. The argument of this option is a nonempty string consisting of any combination of the letters ‘pPfFdDsS’. The letters in the string indicate how the table should be sorted:

- ‘p’: Pass ascending

- ‘P’: Pass descending
- ‘f’: Fail ascending
- ‘F’: Fail descending
- ‘d’: Difference (Pass minus Fail) ascending
- ‘D’: Difference (Pass minus 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.

To limit the number of sorted lines displayed, users can specify the ‘-n’ option or its ‘--top’ synonym, both of which take an integer argument; this will cause only the specified number of the top lines of the table to be displayed.

The ‘-m’ or ‘--module’ option, if specified, limits the output to the given module and (if they exist) its submodules.

If the user specifies the ‘-o’ or ‘--output-to-file’ option, then the output will be written to the specified file instead of being displayed on the screen. Note that if the file already exists, it will be overwritten without warning.

9.10.12 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 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 names 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] [-ptfbs] [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 given one or more of the `'-p'`, `'-t'` and `'-f'` options, or their long equivalents `'--procedures'`, `'--types'`, and `'--functors'`, this command prints ambiguities

only for the indicated kinds of constructs. The default is to print ambiguities for all these three kinds of constructs.

This command does not normally report two kinds of ambiguities among procedures.

First, this command does not usually report operations that have both function and predicate forms, with the predicate version having a (usually output) argument in place of the function's return value. An example is `list.length` being both a function with arity one and a predicate with arity two. The reason for not reporting this by default is that this is usually an *intended* ambiguity, and this command is usually used to find *unintended* ambiguities, so that they can be eliminated by renaming. However, users can ask for these ambiguities to be printed by specifying either the option `'-b'`, or its long form `'--both-pred-and-func'`.

Second, this command does not usually report ambiguities involving procedures that were created by the compiler as a type specialized version of another procedure. The reason for not reporting this by default is that ambiguities among the names of type specialized procedures cannot arise without ambiguities among the names of the original, not-yet-type-specialized procedures, and eliminating the ambiguities among the original names will perforce eliminate the ambiguities among the specialized names as well. However, users can ask for these ambiguities to be printed by specifying either the option `'-s'`, or its long form `'--typespec'`.

`trail_details`

In grades that specify trailing, prints out low-level details of the state of the trail. In other grades, it reports an error.

9.11 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.

9.11.1 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 descendants 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.

9.11.2 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.

9.11.3 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.

9.11.4 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 9.11.3 \[Oracle questions\], page 91](#)), 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 9.11.7 \[Improving the search\], page 97](#)). 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 9.10.4 \[Browsing commands\], page 60](#) 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.

9.11.5 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.

9.11.6 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 9.10.9 \[Declarative debugging mdb commands\]](#), page 80) or with the ‘mode’ declarative debugger command (see [Section 9.11.4 \[Declarative debugging commands\]](#), page 92).

9.11.6.1 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.

9.11.6.2 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.

9.11.6.3 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.

9.11.6.4 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.

9.11.7 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.

9.11.7.1 Tracking suspicious subterms

An incorrect subterm can be tracked to the call that bound the subterm from within the interactive term browser (see [Section 9.11.4 \[Declarative debugging commands\]](#), page 92).

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.

9.11.7.2 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 9.10.9 \[Declarative debugging mdb commands\]](#), page 80 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 9.6 \[Mercury debugger invocation\]](#), page 47). Trusted predicates will also be exported with a ‘`save`’ command (see [Section 9.10.10 \[Miscellaneous commands\]](#), page 82).

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 9.11.4 \[Declarative debugging commands\]](#), page 92.

9.11.7.3 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.

9.12 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*.

9.12.1 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 `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 9.11.6 \[Search modes\]](#), page 96).

9.12.2 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.

9.12.3 Slicing

Once a slice has been generated, it can be viewed in various ways using the ‘mslice’ tool. The output of this 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 debugger event 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 9.3 \[Tracing of Mercury programs\]](#), page 42 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 the first `N` lines.

The `-m` or `--module` option limits the output to events only from the given module.

The `-n` or `--max-name-column-width` option's argument gives the maximum width of the column containing predicate and function names. If the argument is zero, there is no maximum width.

The `-p` or `--max-path-column-width` 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-width` 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.

9.12.4 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 9.12.1 [Generating trace counts], page 99). Once this has been done, and the union of each set (the set of passing runs and the set of failing runs) has been computed using `mtc_union`, invoking `mdice` will 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` event 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 9.3 [Tracing of Mercury programs], page 42 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 passing 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. This number is calculated as $\text{Suspicion} = \text{Fail} / (\text{Pass} + \text{Fail})$ where Fail is the number of times the goal was executed in failing runs and Pass 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
```

where

- ‘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, and
- ‘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 is normally sorted on the identification of the event, meaning it is sorted

- first on the identification of the procedure containing the event (consisting of a pred/func indication, the name of the predicate or function, its arity, and its mode number),
- then on the port (if any),
- then on the path (if any),
- then on the file name, and
- then on the line number.

However, users can also ask for it to be sorted on the Pass, Fail and/or Suspicion columns, or on a combination of these. This can be done with the ‘-s’ or ‘--sort’ option. The argument of this option is a nonempty string consisting of any combination of the letters ‘pPfFdDsS’. The letters in the string indicate how the table should be sorted:

- ‘p’: Pass ascending
- ‘P’: Pass descending
- ‘f’: Fail ascending
- ‘F’: Fail descending
- ‘d’: Difference (Pass minus Fail) ascending

- ‘D’: Difference (Pass minus 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 "S", meaning to only sort descending by suspicion. (Note that this is different from the default for the ‘dice’ command in ‘mdb’.)

To limit the number sorted lines displayed, users can specify the ‘-n’ option or its ‘--top’ synonym, which take an integer argument; this will cause only specified number of the top lines of the table to be displayed.

The ‘-m’ or ‘--module’ option, if specified, limits the output to the given module and (if they exist) its submodules.

The ‘-n’ or ‘--max-name-column-width’ option’s argument gives the maximum width of the column containing predicate and function names. If the argument is zero, there is no maximum width.

The ‘-p’ or ‘--max-path-column-width’ 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-width’ 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.

9.12.5 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.

10 Profiling

10.1 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](#).

10.2 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 11.5 \[Grade options\]](#), page 119.)

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.

10.3 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 two files whose names will have form ‘`programname_on_date_at_time.data`’ and ‘`programname_on_date_at_time.procrep`’. (On Windows, the `.exe` suffix will be omitted from `programname`.) 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 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.

10.4 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 descendants (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 descendants 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 descendants 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.

index	%time	self	descendants	called/total called+self called/total	parents name index children
[1]	100.0	0.00	0.75	0	<spontaneous>
		0.00	0.75	1/1	call_engine_label [1]
					do_interpreter [3]

[2]	100.0	0.00	0.75	1/1	do_interpreter [3]
		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]

[3]	100.0	0.00	0.75	1/1	call_engine_label [1]
		0.00	0.75	1	do_interpreter [3]
		0.00	0.75	1/1	io.run/0(0) [2]

[4]	99.9	0.00	0.74	1/1	io.run/0(0) [2]
		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]

[5]	99.9	0.00	0.74	1/1	main/2(0) [4]
		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 descendants. 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 descendant time: the time spent in the current procedure and all its descendants. 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 descendant time have slightly different meanings. For the parent procedures the self and descendant time represent the proportion of the current procedure’s self and descendant 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.

10.5 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.

10.6 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 backend, 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.

10.7 Using mdprof

The user interface of the deep profiler is a browser. To display the information contained in a deep profiling data file (whose name will have the form ‘*programname_date_time.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/*programname_date_time.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’. (The deep profiler will replace this suffix with ‘.procrep’ to get access to the other file generated by the profiling run.)

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, including the following.

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 be 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
```

10.8 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.

10.9 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.

11 Invocation

This section contains a brief description of all the options available for `mmc`, the Melbourne Mercury compiler. Sometimes this list is a little out-of-date; use `mmc --help` to get the most up-to-date list.

11.1 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. The

compiler will convert module names to file names by looking up the module name in the module-name-to-file-name map in the `Mercury.modules` file if it exists. (It can be created using a command such as `mmc -f *.m`.) If `Mercury.modules` does not exist, then the compiler will search for a module named e.g. `foo.bar.baz` in the 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-`. (You cannot both group *and* negate single-letter flags at the same time.) Long flags may be negated by preceding them with `no-`, e.g. `--no-verbose`.

11.2 Help options

```
-?
-h
--help    Print a usage message.

--version
          Print the compiler version.
```

11.3 Options for modifying the command line

```
--flags-file filename
--flags filename
          Take options from the specified file, and handle them as if they were specified
          on the command line.

--filenames-from-stdin
          Read in from standard input a newline-terminated module name or file name,
          compile that module or file, and repeat until reaching end-of-file. (This allows a
          program or user to interactively compile several modules without the overhead
          of creating a process for each one.)
```

11.4 Options that give the compiler its overall task

```
-f
--generate-source-file-mapping
          Output to Mercury.modules the module-name-to-file-name mapping for the
          list of source files given as non-option arguments to mmc. This mapping is
          needed if either for some modules, the file name does not match the module
          name, or if some of the modules are outside the current directory. In such cases,
          the mapping must be generated before invoking any one of mmc --generate-
dependencies, mmc --make, or mmake depend.
```

-M

`--generate-dependencies`

Output “Make”-style dependencies for the given main module, and all the other modules in the program (i.e. all the other modules in this directory that the main module imports either directly or indirectly) to ‘`module.dep`’, to ‘`module.dv`’, and to the ‘`.d`’ files of all the modules in the program.

`--generate-dependencies-ints`

Do the same job as `--generate-dependencies`, but also output ‘`.int3`’, ‘`.int0`’, ‘`.int`’ and ‘`.int2`’ files for all the modules in the program.

`--generate-dependency-file`

Output “Make”-style dependencies for the given module to ‘`module.d`’.

`--make-short-interface`

`--make-short-int`

Write to ‘`module.int3`’ a list of the types, insts, modes, typeclasses and instances defined in the interface section of the named module. The compiler uses these files to create ‘`.int0`’, ‘`.int`’ and ‘`.int2`’ files.

`--make-private-interface`

`--make-priv-int`

Write to ‘`module.int0`’ the list of the entities (including types, insts, modes, predicates and functions) defined in the given module that its submodules have access to. (This will include even entities that are *not* exported from the module.) Besides the code of the module itself, the inputs to this task are the ‘`.int0`’ files of the given module’s own ancestor modules, and the ‘`.int3`’ files of the modules it imports directly or indirectly. Note that this command is unnecessary for modules that have no submodules.

-i

`--make-interface`

`--make-int`

Write to ‘`module.int`’ and to ‘`module.int2`’ a list of entities (including types, insts, modes, predicates and functions) that the given module exports for use by other modules. When generating code, the compiler reads the ‘`.int`’ file of every directly imported module, and the ‘`.int2`’ file of every indirectly imported module. (Each ‘`.int2`’ file is a slightly shorter version of the corresponding ‘`.int`’ file, because it is specialized for its intended use.)

`--make-optimization-interface`

`--make-optimisation-interface`

`--make-opt-int`

Write to ‘`module.opt`’ information about the semantically-private parts of the named module that can be useful when optimizing another module that imports this one. Note that ‘`.opt`’ files are used by ‘`--intermodule-optimization`’.

```

--make-transitive-optimization-interface
--make-transitive-optimisation-interface
--make-trans-opt
    Write to module.trans_opt information about the named module that can
    be useful when optimizing another module that imports this one. The dis-
    tinction from .opt files is that a .trans_opt file can include information
    not just from the source code of its module, but also from the .opt and
    .trans_opt files of other modules. Note that .trans_opt files are used by
    --transitive-intermodule-optimization.

-t
--typecheck-only
    Check the module's code only for syntax- and type errors. Do not execute any
    other semantic checks, and do not generate any code. (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's code for syntax- and semantic errors, but do not generate
    any code.

-C
--target-code-only
    Generate target code (meaning C code in module.c, C# code in module.cs,
    or Java code in module.java), but do not generate object code.

-c
--compile-only
    Generate C code in module.c and object code in module.o, but do not
    attempt to link the object files.

--generate-standalone-interface basename
--no-generate-standalone-interface
    Output a stand-alone interface. Use basename as the basename of any files gen-
    erated for the stand-alone-interface. (See Chapter 16 \[Stand-alone interfaces\],
    page 179 for further details.)

-P
--convert-to-mercury
--convert-to-Mercury
--pretty-print
    Output the code of the module to module.ugly in a standard format. 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.)

```

`-x`
`--make-xml-documentation`
`--make-xml-doc`
Output an XML representation of all the declarations in the module to `'module.xml'`. This XML file can then be transformed via a XSL transform into another documentation format.

`-m`
`--make` Treat the non-option arguments to `mmc` as files to make, rather than source files. Build or rebuild the specified files if they do not exist or are not up-to-date. Note that this option also enables `'--use-subdirs'`.

`-r`
`--rebuild`
A variant of the `'--make'` option, with the difference being that it always rebuilds the target files, even if they are up-to-date.

`--output-grade-string`
Print to standard output the canonical string representing the currently selected grade.

`--output-grade-defines`
Print to standard output the C compiler flags that define the macros which specify the selected compilation grade.

`--output-stdlib-grades`
Print to standard output the list of compilation grades in which the Mercury standard library is available with this compiler.

`--output-stdlib-modules`
Print to standard output the names of the modules in the Mercury standard library.

`--output-libgrades`
`--output-library-install-grades`
Print to standard output the list of compilation grades in which a library to be installed should be built.

`--output-target-arch`
Print to standard output the target architecture.

`--output-cc`
Print to standard output the command for invoking the C compiler.

`--output-c-compiler-type`
`--output-cc-type`
Print to standard output the C compiler's type.

`--output-cflags`
Print to standard output the flags with which the C compiler will be invoked.

`--output-c-include-directory-flags`
`--output-c-include-dir-flags`
Print to standard output the C compiler flags that specify which directories to search for C header files. This includes the C header files from the standard library.

`--output-link-command`
Print to standard output the link command used to create executables.

`--output-shared-lib-link-command`
Print to standard output the link command used to create shared libraries.

`--output-library-link-flags`
Print to standard output the flags that must be passed to the linker in order to link against the current set of libraries. This includes the Mercury standard library, as well as any other libraries specified via either the `'--ml'` or `'-l'` option.

`--output-csharp-compiler`
Print to standard output the command for invoking the C# compiler.

`--output-csharp-compiler-type`
Print to standard output the C# compiler's type.

`--output-java-class-directory`
`--output-class-directory`
`--output-java-class-dir`
`--output-class-dir`
Print to standard output the name of the directory in which the compiler will place any generated Java class files.

`--output-optimization-options`
`--output-opt-opts`
Print to standard output a list of the optimizations enabled at each optimization level.

```
--output-optimization-options-upto max_level
--output-opt-opts-upto max_level
    Print to standard output a list of the optimizations enabled at each optimization
    level up to the given maximum.
```

11.5 Grade options

11.5.1 Grades and grade components

```
-s grade
--grade grade
```

Select the compilation model. This model, which Mercury calls a *grade*, specifies what properties the resulting executable or library should have. Properties such as ‘generates profiling data for mprof when executed’ and ‘can be debugged with the Mercury debugger’. As such, it controls decisions that must be made the same way in all the modules of a program. For example, it does not make sense to compile some modules of a program to C and some to Java; nor does it make sense to compile some modules to support profiling and some to not support profiling.

The *grade* should consist of one of the base grades ‘none’, ‘reg’, ‘asm_fast’, ‘hlc’, ‘java’, or ‘csharp’, followed by zero or more of the grade modifiers in the following options. The names of all grade modifiers start with a period, so a complete grade name consists of a list of name components (the base grade and some grade modifiers) separated by periods.

Note that not all combinations of components are allowed, and that the Mercury standard library will have been installed on your system in only a subset of the set of all possible grades.

Attempting to build a program in a grade which has not been installed or to link together modules that have been compiled in different grades, will result in an error.

11.5.2 Target options

```
--target c (grades: ‘none’, ‘reg’, ‘asm_fast’, ‘hlc’)
--target csharp (grades: ‘csharp’)
--target java (grades: ‘java’)
    Specify the target language: C, C# or Java. The default is C. Targets other
    than C imply ‘--high-level-code’ (see below).
```

```

--compile-to-c
--compile-to-C
    An abbreviation for '--target c' '--target-code-only'. Generate C code in
    'module.c', but do not generate object code.

--java
--Java    An abbreviation for '--target java'.

--java-only
--Java-only
    An abbreviation for '--target java' '--target-code-only'. Generate Java
    code in 'module.java', but do not generate Java bytecode.

--csharp
--C#      An abbreviation for '--target csharp'.

--csharp-only
--C#-only
    An abbreviation for '--target csharp' '--target-code-only'. Generate C#
    code in 'module.cs', but do not generate CIL bytecode.

```

11.5.3 LLDS backend grade options

```

--gcc-global-registers (grades: 'reg', 'asm_fast')
--no-gcc-global-registers (grades: 'none')
    Specify whether to use GNU C's global register variables extension. This option
    is used only when targeting C with '--no-high-level-code'.

--gcc-non-local-gotos (grades: 'asm_fast')
--no-gcc-non-local-gotos (grades: 'none', 'reg')
    Specify whether to use GNU C's "labels as values" extension. This option is
    used only when targeting C with '--no-high-level-code'.

--asm-labels (grades: 'asm_fast')
--no-asm-labels (grades: 'none', 'reg')
    Specify whether to use GNU C's asm extensions for inline assembler labels.
    This option is used only when targeting C with '--no-high-level-code'.

```

11.5.4 MLDS backend grade options

`-H` (grades: ‘hlc’, ‘java’, ‘csharp’)

`--high-level-code`

`--high-level-c`

`--high-level-C`

`--highlevel-code`

`--highlevel-c`

`--highlevel-C`

Use the MLDS backend, which generates idiomatic high-level-language code, rather than the LLDS backend, which generates assembly language code in C syntax.

`--target-debug-grade` (grades: ‘hlc’, ‘java’, ‘csharp’)

Require that all modules in the program be compiled to object code (for C), ‘.class’ files (for Java), or ‘.dll’ files (for C#) in a way that allows the program executable to be debuggable with debuggers for the target language, such as gdb for C. This option is intended mainly for the developers of Mercury, though it can also help to debug foreign language code included in Mercury programs.

11.5.5 Debugging grade options

11.5.5.1 Mdb debugging grade options

`--debug` (grade modifier: ‘.debug’)

Enable Mercury-level debugging. See [Chapter 9 \[Debugging\]](#), page 39 for details. This option is supported only when targeting C with ‘--no-high-level-code’.

`--decl-debug` (grade modifier: ‘.decldebug’)

Enable full support for declarative debugging. This allows subterm dependency tracking in the declarative debugger. See [Chapter 9 \[Debugging\]](#), page 39 for details. This option is supported only when targeting C with ‘--no-high-level-code’.

11.5.6 Profiling grade options

11.5.6.1 Mprof profiling grade options

`-p` (grade modifier: ‘.prof’)

`--profiling`

Prepare the generated code for time profiling by Mercury’s version of the standard Unix profiler `gprof`, which is a tool called `mprof`. In ‘.prof’ grades, the compiler will insert profiling hooks into the generated code (e.g. to count calls), and will also output the static call graph of the module to ‘`module.prof`’ for use

by `mprof`. Please see the [Section 10.2 \[Building profiled applications\]](#), page 105 in the Mercury User’s Guide for details.

This option is supported only when targeting C.

`--memory-profiling` (grade modifier: `‘.memprof’`)

Prepare the generated code for profiling of memory usage and retention by `mprof`. Please see [Section 10.5 \[Using mprof for profiling memory allocation\]](#), page 110 and [Section 10.6 \[Using mprof for profiling memory retention\]](#), page 110 in the Mercury User’s Guide for details.

This option is supported only when targeting C.

11.5.6.2 Deep profiling grade options

`--deep-profiling` (grade modifier: `‘.profdeep’`)

Prepare the generated code for deep profiling. The Mercury deep profiling tool `mdprof` (note the “d” in the name) associates much more context with each measurement than `mprof`, making it much more suitable for handling polymorphic code and higher order code, both of which are much more common in typical Mercury code than in typical C code. This option is supported only when targeting C with `‘--no-high-level-code’`.

`--no-coverage-profiling`

Do not gather deep profiling information that is useful only for coverage profiling.

`--profile-for-feedback`

`--profile-for-implicit-parallelism`

Select deep profiling options that are suitable for profiler directed implicit parallelism. `‘--profile-for-implicit-parallelism’` is a deprecated synonym for this option.

11.5.7 Optional feature grade options

`--garbage-collection {none, Boehm, automatic}`

`--gc {none, Boehm, automatic}`

Specify which method of garbage collection to use. When targeting Java or C#, the only possible choice is `‘automatic’`, which means the garbage collector built into the target language. When targeting C, the usual choice is `‘Boehm’`, which is Hans Boehm et al’s conservative collector. The use of the Boehm collector is indicated by the `‘.gc’` grade component. The other alternative when targeting C is `‘none’`, meaning there is no garbage collector. This works only for programs with very short runtimes.

- `--stack-segments` (grade modifier: `‘.stseg’`)
Specify the use of 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 supported only when targeting C with `‘--no-high-level-code’`.
- `--use-trail` (grade modifier: `‘.tr’`)
Enable use of a trail. This is necessary for interfacing with constraint solvers, or for backtrackable destructive update. This option is supported only when targeting C.
- `--single-precision-float` (grade modifier: `‘.spf’`)
`--single-prec-float`
Use single precision floats so that, on 32-bit machines, floating point values don’t need to be boxed. The default is to use double precision floats. This option is supported only when targeting C.
- `--parallel` (grade modifier: `‘.par’`)
Enable concurrency and parallel conjunction support for the low-level C grades. Enable concurrency (via pthreads) for the high-level C grades.
- `--maybe-thread-safe {yes, no}`
Specify how the compiler should treat the `maybe_thread_safe` foreign code attribute. `‘yes’` means that a foreign procedure with the `maybe_thread_safe` attribute is treated as if it has a `thread_safe` attribute. `‘no’` means that the foreign procedure is treated as if it has a `not_thread_safe` attribute. The default is `‘no’`.

11.5.8 Developer grade options

- `--num-ptag-bits N`
- `--num-tag-bits N`
Use `‘N’` primary tag bits. Note that the value of this option is normally auto-configured; its use should never be needed except for cross-compilation to an architecture where autoconfiguration would yield a different value.

11.6 Options that control inference

- `--infer-all`
This option is an abbreviation for the combination of `‘--infer-types’`, `‘--infer-modes’`, and `‘--infer-det’`.
- `--infer-types`
If there is no type declaration for a predicate or function, try to infer its type, instead of just reporting an error.

--infer-modes

If there is no mode declaration for a predicate, try to infer its mode (or modes), instead of just reporting an error.

--no-infer-determinism**--no-infer-det**

If there is no determinism declaration for a procedure (a mode of a predicate or of a function), just report an error; do not try to infer its determinism.

--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).

--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 11.10 \[Warning options\]](#), page 126.)

11.7 Options specifying the intended semantics

--strict-sequential

This option is an abbreviation for the combination of ‘**--no-reorder-conj**’, ‘**--no-reorder-disj**’, and ‘**--fully-strict**’.

--no-reorder-conj

Execute conjunctions left-to-right. Do not reorder conjuncts, except where the modes require it (to put the producer of each variable before all its consumers).

--no-reorder-disj

Execute disjunctions strictly left-to-right; do not reorder disjuncts.

--no-fully-strict

Allow infinite loops, and goals whose determinism is erroneous, to be optimised away.

11.8 Verbosity options

-v**--verbose**

Output progress messages at each stage in the compilation.

- V**
--very-verbose
Output very verbose progress messages.

- S**
--statistics
Output messages about the compiler's time/space usage at the boundaries between phases of the compiler.

- no-verbose-make**
Disable messages about the progress of builds when using `mmc --make`.

- output-compile-error-lines *N***
--no-output-compile-error-lines
With '`--make`', output the first *N* lines of the '`.err`' file after compiling a module (default: 100). Specifying '`--no-output-compile-error-lines`' removes the limit.

- verbose-commands**
Output each external command before it is run. Note that some commands will only be printed with '`--verbose`'.

11.9 Diagnostics options

11.9.1 Options that control diagnostics

- E**
--verbose-error-messages
--verbose-errors
Some error messages have two versions: a standard version intended for experienced users, and a verbose version intended for new users. The default is to print the first version. This option tells the compiler to print the second version, which will offer a more detailed explanation of any errors it finds in your code.

- reverse-error-order**
Print error messages in descending order of their line numbers, instead of the usual ascending order. This is useful if you want to work on the last errors in a file first.

- max-error-line-width *N***
--no-max-error-line-width
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.

`--limit-error-contexts filename:minline1-maxline1,minline2-maxline2`

`--no-limit-error-contexts`

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-contexts`’ 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.

`--error-files-in-subdir`

This option causes `mmc --make` to put ‘`.err`’ files into the ‘Mercury’ subdirectory instead of the current directory. (This option has no effect on `mmake`.)

`--typecheck-ambiguity-warn-limit N`

Set the number of type assignments required to generate a warning about highly ambiguous overloading to *N*. (Default: 50.)

`--typecheck-ambiguity-error-limit N`

Set the number of type assignments required to generate an error about excessively ambiguous overloading to *N*. If this limit is reached, the typechecker will not process the predicate or function any further. (Default: 3000.)

11.9.2 Options that control color in diagnostics

`--color-diagnostics`

`--colour-diagnostics`

Disable the use of colors in diagnostic messages. Please see [Section 13.4 \[Enabling the use of color\]](#), page 177 for the details.

`--color-scheme ColorScheme`

`--colour-scheme ColorScheme`

Specify the color scheme to use for diagnostics, if the use of color in diagnostics is enabled. For information about how the compiler uses colors in diagnostic messages, and about the syntax of color scheme specifications, please see [Section 13.3 \[Color schemes\]](#), page 175.

11.10 Warning options

11.10.1 Warnings about possible incorrectness

11.10.1.1 Warnings about possible module incorrectness

`--no-warn-nothing-exported`

Do not warn about modules which export nothing.

- `--warn-unused-imports`
Warn about modules that are imported but not used.

- `--no-warn-unused-interface-imports`
`--no-warn-interface-imports`
Do not warn about modules imported in the interface, but which are not used in the interface.

- `--warn-interface-imports-in-parents`
Warn about modules that are imported in the interface of a parent module, but not used in the interface of that module.

- `--warn-unused-types`
Warn about types that are neither used in their defining module nor exported to other modules.

- `--no-warn-stdlib-shadowing`
Do not generate warnings for module names that either duplicate the name of a module in the Mercury standard library, or contain a subsequence of name components that do so.

- `--no-warn-duplicate-abstract-instances`
Do not warn about duplicate abstract typeclass instances. (Duplicate concrete typeclass instances are errors, and are always reported.)

- `--warn-too-private-instances`
An instance declaration has to be private if it is for a private type class (meaning a type class that is not visible outside the current module), if some of its member types refer to private type constructors, or if its class constraints refer to private type classes or private type constructors. Such instances can only be relevant in the current module. Generate a warning if an instance declaration that can be relevant outside the current module is not exported.

- `--no-warn-subtype-ctor-order`
Do not warn about a subtype definition that lists its data constructors in a different order than its supertype.

11.10.1.2 Warnings about possible inst incorrectness

- `--no-warn-insts-without-matching-type`
Do not warn about insts that are not consistent with any of the 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.

`--no-warn-exported-insts-for-private-type`

Do not warn about exported insts that match only private types.

11.10.1.3 Warnings about possible predicate incorrectness

`--no-warn-det-decls-too-lax`

Do not warn about determinism declarations which could be stricter.

`--no-warn-inferred-erroneous`

Do not warn about procedures whose determinism is inferred to be ‘erroneous’, but whose determinism declarations are looser.

`--no-warn-unresolved-polymorphism`

Do not warn about unresolved polymorphism, which occurs when the type of a variable contains a type variable that is not bound to an actual type, even though it should be.

`--no-warn-stubs`

Do not warn about procedures for which there are no clauses. Note that this option is meaningful only if the ‘`--allow-stubs`’ option is enabled.

`--no-warn-cannot-table`

Do not warn about tabling pragmas which for some reason cannot be applied to the specified procedure.

`--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 meaningful only if termination analysis is enabled.

`--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 through negation through some path in its call graph.

`--no-warn-unneeded-purity-pred-decl`

Do not warn about predicate and function declarations that specify a purity level that is less pure than the predicate or function definition.

`--no-warn-typecheck-ambiguity-limit`

Do not generate a warning when the number of type assignments needed to process the definition of a predicate or function reaches or exceeds the typechecker’s ambiguity warn limit.

11.10.1.4 Warnings about possible pragma incorrectness

`--no-warn-ambiguous-primas`

`--no-warn-ambiguous-pragma`

Do not warn about primas that do not specify whether they are for a predicate or a function, even when there is both a predicate and a function with the given name and arity.

`--warn-potentially-ambiguous-primas`

`--warn-potentially-ambiguous-pragma`

Warn about primas that do not specify whether they are for a predicate or a function.

`--no-warn-table-with-inline`

Do not warn about tabled procedures that also have a `pragma inline` declaration. (This combination does not work, because inlined copies of procedure bodies cannot be tabled.)

`--no-warn-unneeded-purity-pragma`

Do not warn about purity promise primas that specify a purity level that is less pure than the definition of the predicate or function that they apply to.

`--no-warn-nonexported-pragma`

Do not warn about non-exported primas that declare something about an exported predicate or function (such as an assertion that it terminates).

11.10.1.5 Warnings about possible goal incorrectness

`--no-warn-simple-code`

Do not warn about constructs which are so simple that they are likely to be programming errors. (One example is if-then-elses whose condition always succeeds.)

`--no-warn-singleton-variables`

`--no-warn-singleton-vars`

Do not warn about variables which only occur once in a clause, but whose names do not start with an underscore.

`--no-warn-repeated-singleton-variables`

`--no-warn-repeated-singleton-vars`

Do not warn about variables which occur more than once in a clause, but whose names do start with an underscore.

`--no-warn-unification-cannot-succeed`

Do not warn about unifications which cannot succeed.

- no-warn-known-bad-format-calls**
Do not warn about calls to `string.format`, `io.format`, or `stream.string_writer.format` that contain mismatches between the format string and the supplied values.

- no-warn-obsolete**
Do not warn about calls to predicates and functions that have been marked as obsolete.

- no-warn-overlapping-scopes**
Do not warn about variables which occur in overlapping scopes.

- no-warn-suspected-occurs-check-failure**
- no-warn-suspected-occurs-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 the test called the *occurs check*.

- warn-suspicious-recursion**
Warn about recursive calls which are likely to have problems, such as leading to infinite recursion.

- warn-unused-args**
Warn about predicate or function arguments which are not used.

- no-warn-unneeded-purity-indicator**
Do not warn about purity indicators on goals that specify a purity level that is less pure than the declaration of the called predicate or function.

- no-warn-missing-state-var-init**
Do not print warnings about state variables that are initialized on some but not all paths through a disjunction or if-then-else.

- no-warn-moved-trace-goal**
Do not print warning about trace goals that were moved after goals that follow them in the text of the program. Such reordering may mean that the trace goal will not be executed if the goal moved before it fails, and even if it does get executed, it may be executed in a different context than the one expected by the programmer.

- no-warn-disj-fills-partial-inst**
Do not print warnings about disjunctions that further instantiate some variables that enter the disjunction in a partially-instantiated state. While such disjunctions work fine in most contexts, if they constitute the body of a predicate or a function, that predicate or function will not work when passed to

an all-solutions predicate. This is because while the solutions that the different disjuncts in the disjunction can generate distinct values for the affected variables, those values will be represented as terms that have the exact same address, namely the address of the initial partially-instantiated term. This fact will lead the all-solutions predicate to consider them to be the **same** solution.

`--no-warn-unknown-warning-name`

Do not report unknown warning names in the list of warnings to disable in `disable_warning(s)` scopes.

11.10.1.6 Warnings about missing files

`--no-warn-undefined-options-variables`

`--no-warn-undefined-options-vars`

Do not warn about references to undefined variables in options files with `'--make'`.

`--no-warn-missing-descendant-modules`

Do not warn about modules which cannot be found, even though some of their ancestor modules exist in the current directory.

`--no-warn-missing-opt-files`

Do not warn about `'.opt'` files which cannot be opened.

`--warn-missing-trans-opt-files`

Warn about `'.trans_opt'` files which cannot be opened.

`--no-warn-missing-trans-opt-deps`

Do not generate a warning when the information required to allow `'.trans_opt'` files to be read when creating other `'.trans_opt'` files has been lost. The information can be recreated by running `mmake 'mainmodule.depend'`.

11.10.2 Warnings about possible performance issues

`--no-warn-accumulator-swaps`

Do not warn about argument order rearrangements done by `'--introduce-accumulators'`.

`--no-warn-unneeded-final-statevars`

Do not warn about `!:S` state variables in clause heads whose value will always be the same as `!.S`.

`--no-warn-unneeded-final-statevars-lambda`

Do not warn about `!:S` state variables in lambda expressions whose value will always be the same as `!.S`.

```
--no-warn-no-auto-parallelisation
--no-warn-no-auto-parallelization
    Do not warn about procedures and goals that could not be automatically parallelised.

--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.

--warn-non-tail-recursion {none,self,self-and-mutual}
--no-warn-non-tail-recursion
    Specify when the compiler should warn about recursive calls that are not tail calls.

--no-warn-no-recursion
    Do not generate a warning when a predicate or function in which the programmer requests warnings about non-tail recursion has no recursive calls at all.
```

11.10.3 Warnings about programming style

11.10.3.1 Warnings about style issues with modules

```
--warn-include-and-non-include
    Warn about modules that contain both “include_module” declarations and other kinds of entities, such as types or predicates.
```

When a module contains both “include_module” declarations and other code, changes to that code will often cause the recompilation of all the included submodules. This is because those submodules have access to the entities declared in their parent module, so if the set of those entities changes in any way, the submodules must be checked to see whether they relied on some entity in the parent module that is not there anymore.

Modules that contain “include_module” declarations and nothing else, which are often called “packages”, do not have this problem.

11.10.3.2 Warnings about style issues with predicates

```
--warn-dead-predicates
--warn-dead-preds
    Warn about predicates and functions that have no procedures which are ever called.
```

`--warn-dead-procedures`

`--warn-dead-procs`

Warn about procedures which are never called.

`--warn-can-fail-function`

Warn about functions that can fail. (Such functions should be replaced by semidet predicates.)

`--no-warn-unnneeded-mode-specific-clause`

Do not warn about clauses that unnecessarily specify the modes of their arguments.

11.10.3.3 Warnings about style issues with goals

`--no-warn-redundant-code`

Do not warn about redundant constructs in Mercury code. (One example is importing a module more than once.)

`--warn-ite-instead-of-switch`

`--inform-ite-instead-of-switch`

Generate warnings for if-then-elses that could be replaced by switches.

`--warn-incomplete-switch`

`--inform-incomplete-switch`

Generate warnings for switches that do not cover all the function symbols that the switched-on variable could be bound to.

`--warn-incomplete-switch-threshold N`

`--inform-incomplete-switch-threshold N`

Have the ‘`--warn-incomplete-switch`’ option generate its messages only for switches that *do* cover at least ‘*N*%’ of the function symbols that the switched-on variable could be bound to.

`--warn-duplicate-calls`

Warn about multiple calls to a predicate or function with the same input arguments.

`--no-warn-redundant-coerce`

Do not warn about redundant type coercions, which occur when the type of the result of the `coerce` expression is the same as the type of its argument.

`--no-warn-requested-by-code`

Do not generate warnings that are specifically requested by the code being compiled, such as `require_switch_arms_in_type_order` pragmas.

--no-warn-requested-by-option

Do not generate warnings that are specifically requested by compiler options, such as “`--enable-termination`”.

--no-warn-state-var-shadowing

Do not warn about one state variable shadowing another.

--no-warn-unnneeded-initial-statevars

Do not warn about state variables in clause heads that could be ordinary variables.

--no-warn-unnneeded-initial-statevars-lambda

Do not warn about state variables in the heads of lambda expressions that could be ordinary variables.

--warn-implicit-stream-calls

Warn about calls to I/O predicates that could take explicit stream arguments, but do not do so.

--warn-unknown-format-calls

Warn about calls to `string.format`, `io.format` or `stream.string_writer.format` for which the compiler cannot tell whether there are any mismatches between the format string and the supplied values.

--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.

--warn-suspicious-foreign-procs

Warn about possible errors in the bodies of `foreign_proc` pragmas. 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 it should not contain, such as `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.

11.10.3.4 Warnings about missing order

`--warn-unsorted-import-blocks`

`--warn-unsorted-import-block`

Generate a warning if two `import_module` and/or `use_module` declarations occur on the same line, or if a sequence of such declarations on consecutive lines are not sorted on module name.

`--warn-inconsistent-pred-order-clauses`

`--warn-inconsistent-pred-order`

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.

11.10.3.5 Warnings about missing contiguity

`--no-warn-non-contiguous-decls`

Do not generate a warning if the mode declarations of a predicate or function do not all immediately follow its `pred` or `func` 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.

`--allow-non-contiguity-for name1,name2,...`

`-no-allow-non-contiguity-for`

Allow the clauses (or, with `'--warn-non-contiguous-foreign-procs'`, the clauses and/or `foreign_proc` pragmas) of the named predicates and/or functions to be intermingled with each other, but not with those or any other predicates or functions. This option may be specified more than once, with each option value specifying a distinct set of predicates and/or function names that may be intermingled. Each name must uniquely specify a predicate or a function.

11.10.4 Options that control warnings

`-w`

`--inhibit-warnings`

Disable all warning messages.

`--inhibit-style-warnings`

Disable all warning messages about programming style.

`--warn-all-format-string-errors`

If a format string has more than one mismatch with the supplied values, generate a warning for all mismatches, not just the first. (The default is to report only the first, because later mismatches may be avalanche errors caused by earlier mismatches.)

`--no-warn-up-to-date`

Do not warn if targets specified on the command line with `mmc --make` are already up-to-date.

11.10.5 Options about halting for warnings

`--halt-at-warn`

This option causes the compiler to treat all warnings as if they were errors when intending to generate target code. This means that if the compiler issues any warnings, it will not generate target code; instead, it will return a non-zero exit status.

`--halt-at-warn-make-interface`

`--halt-at-warn-make-int`

This option causes the compiler to treat all warnings as if they were errors when intending to generate 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 intending to generate 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-errors`

This option causes the compiler to halt immediately, without doing 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 print an error message for each problem, but will then stop.

Users can prevent this stop, and thus allow the generation of invalid interface files, by specifying ‘`--no-halt-at-invalid-interface`’. (In this case, the problems in the invalid information files will be reported when compiling the modules that import them.)

11.11 Options that request information

--inform-inferred

Do not print inferred types or modes.

--no-inform-inferred-types

Do not print inferred types.

--no-inform-inferred-modes

Do not print inferred modes.

--inform-suboptimal-packing

Generate messages if the arguments of a data constructor could be packed more tightly if they were reordered.

--show-pred-moveability *pred_or_func_name***-no-show-pred-moveability****--show-pred-movability** *pred_or_func_name***-no-show-pred-movability**

Write out a short report on the effect of moving the code of the named predicate or function (or the named several predicates and/or functions, if the option is given several times) to a new module. This includes listing the other predicates and/or functions that would have to be moved with them, and whether the move would cause unwanted coupling between the new module and the old.

11.12 Options that ask for informational files

--show-definitions**--show-defns**

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`

`--show-defn-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.

The line counts are only approximate because the compiler does not need, and therefore does not keep around, information such as the context of a line that contains only a close parenthesis ending a clause.

`--show-definition-extents`

`--show-defn-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.

The line numbers are only approximate because the compiler does not need, and therefore does not keep around, information such as the context of a line that contains only a close parenthesis ending a clause.

`--show-local-call-tree`

Construct the local call tree of the predicates and functions defined in the module. Each node of this tree is a local predicate or function, and each node has edges linking it to the nodes of the other local predicates and functions it directly refers to. Write out to `'module.local_call_tree'` a list of these nodes. Put these nodes into the order in which they are encountered by a depth-first left-to-right traversal of the bodies (as reordered by mode analysis), of the first procedure of each predicate or function, starting the traversal at the exported predicates and/or functions of the module. List the callees of each node in the same order.

Write a flattened form of this call tree, containing just the predicates and functions in the same traversal order, to `'module.local_call_tree_order'`.

Construct another call tree of the predicates and functions defined in the module in which each entry lists not just the local predicates/functions directly referred to, but all directly or indirectly referenced predicates/functions, whether or not they are defined in the current module. The one restriction is that we consider only references that occur in the body of the current module. Write out this tree to `'module.local_call_full'`.

```

--show-local-type-representations
--show-local-type-repns
    Write out information about the representations of all types defined in the
    module being compiled to 'module.type_repns'.

--show-all-type-representations
--show-all-type-repns
    Write out information about the representations of all types visible in the mod-
    ule being compiled to 'module.type_repns'.

--show-dependency-graph
    Write out the dependency graph to 'module.dependency_graph'.

--show-imports-graph
--imports-graph
    If '--generate-dependencies' is specified, then write out the imports graph
    to 'module.imports_graph' in a format that can be processed by the graphviz
    tools. The graph will contain an edge from the node of module A to the node
    of module B if module A imports module B.

```

11.13 Controlling trace goals

```

--trace-flag keyword
-no-trace-flag
    Enable the trace goals that depend on the keyword trace flag.

```

11.14 Preparing code for mdb debugging

```

--trace {minimum, shallow, deep, rep, default}
    Generate code that includes the specified level of execution tracing. See
    Chapter 9 \[Debugging\], page 39.

--exec-trace-tail-rec
    Generate TAIL events for self-tail-recursive calls instead of EXIT events. This
    allows these recursive calls to reuse their parent call's stack frame, but it also
    means that the debugger won't have access to the contents of the reused stack
    frames.

--trace-optimized
--trace-optimised
    Do not disable optimizations that can change the trace.

--event-set-file-name filename
    Get the specification of user-defined events from filename.

```

--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 so if the user specifies ‘**--no-delay-death**’. This may be necessary if without it, the stack frames of some procedures grow too big.

--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.

11.15 Preparing code for mdprof profiling

--profile-optimized**--profile-optimised**

Do not disable optimizations that can distort deep profiles.

11.16 Optimization options

11.16.1 Overall control of optimizations

-O N**--optimization-level N****--optimisation-level N****--opt-level N**

Set optimization level to *N*. Optimization level -1 means no optimization while optimization level 6 means full optimization. The option ‘**--output-optimization-options**’ lists the optimizations enabled at each level.

Note that some options are not enabled automatically at *any* optimization level. These include options that are too new and experimental for large scale use, and options that generate speedups in some use cases, but slowdowns in others, require situation-specific consideration of their use.

Likewise, if you want the compiler to perform intermodule optimizations, where the compiler exploits information about the non-public parts of the modules it imports (which it gets from their ‘.opt’ files) for optimization purposes then you must enable that separately, partially because they affect the compilation process in ways that require special treatment by **mmake**. This goes double for *transitive* intermodule optimizations, where the compiler exploits information

about the non-public parts of not just the modules it imports, but also from the modules that *they* import, directly or indirectly. (It gets this info from the `.trans_opt` files of the directly or indirectly imported modules.)

`--optimize-space`

`--optimise-space`

`--opt-space`

Turn on optimizations that reduce code size, and turn off optimizations that significantly increase code size.

11.16.2 Source-to-source optimizations

`--optimize-dead-procs`

`--optimise-dead-procs`

Delete all procedures that are never called.

Optimization levels 0 to 6 automatically set `--optimize-dead-procs`.

`--unneeded-code`

Remove goals from computation paths where their outputs are not needed, provided the semantics options allow the deletion or movement of the goal.

`--unneeded-code-copy-limit copy_limit`

Specify 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-unused-args`

`--optimise-unused-args`

Delete unused arguments from predicates and functions. This will cause the compiler to generate more efficient code for many polymorphic predicates.

Optimization levels 3 to 6 automatically set `--optimize-unused-args`.

`--intermod-unused-args`

Delete unused arguments from predicates and functions even when the analysis required crosses module boundaries. This option implies `--optimize-unused-args` and `--intermodule-optimization`.

`--no-optimize-format-calls`

Do not optimize calls to `string.format`, `io.format`, and `stream.string_writer.format` at compile time. The default is to interpret the format string in such calls at compile time, replacing those calls with the sequence of more primitive operations required to implement them.

`--optimize-constant-propagation`

`--optimise-constant-propagation`

Given calls to some frequently used library functions and predicates, mainly those that do arithmetic, evaluate them at compile time, if all their input arguments are constants.

Optimization levels 2 to 6 automatically set `-optimize-constant-propagation`.

`--optimize-duplicate-calls`

`--optimise-duplicate-calls`

Given multiple calls to a predicate or function with the same input arguments, optimize away all but one.

`--inlining`

Ask the compiler to inline procedures using its usual heuristics.

`--inline-single-use`

Inline procedures which are called only from one call site.

Optimization levels 2 to 6 automatically set `-inline-single-use`.

`--inline-simple`

Inline all simple procedures.

Optimization levels 2 to 6 automatically set `-inline-simple`.

`--inline-simple-threshold threshold`

With `'--inline-simple'`, inline a procedure if its size is less than the given threshold.

Optimization levels 4 to 6 automatically set `-inline-simple-threshold=8`.

`--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-compound-threshold threshold`

Inline a procedure if its size (measured roughly in terms of the number of connectives in its internal form) less the assumed call cost, multiplied by the number of times it is called, is below the given threshold.

Optimization levels 2 to 3 automatically set `-inline-compound-threshold=10`.

Optimization level 4 automatically sets `-inline-compound-threshold=20`.

Optimization levels 5 to 6 automatically set `-inline-compound-threshold=100`.

`--inline-call-cost cost`

Assume that the cost of a call is the given parameter. Used only in conjunction with `'--inline-compound-threshold'`.

`--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.

`--optimize-higher-order`

`--optimise-higher-order`

Create specialized variants of higher-order predicates and functions for call sites where the values of the higher-order arguments are known.

Optimization levels 3 to 6 automatically set `-optimize-higher-order`.

`--type-specialization`

`--type-specialisation`

Enable specialization of polymorphic predicates where the polymorphic types are known.

`--user-guided-type-specialization`

`--user-guided-type-specialisation`

Enable specialization of polymorphic predicates for which there are `:- pragma type_spec` declarations. See the "Type specialization" section of the "Pragmas" chapter of the Mercury language Reference Manual for more details.

Optimization levels 2 to 6 automatically set `-user-guided-type-specialization`.

`--higher-order-size-limit max_size`

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.

Optimization level 4 automatically sets `-higher-order-size-limit=30`.

Optimization levels 5 to 6 automatically set `-higher-order-size-limit=40`.

`--higher-order-arg-limit max_size`

Set the maximum size of higher-order arguments to be specialized by `'--optimize-higher-order'` and `'--type-specialization'`.

--loop-invariants

Hoist loop invariant computations out of loops.

Optimization levels 5 to 6 automatically set `-loop-invariants`.

--introduce-accumulators

Attempt to make procedures tail recursive by introducing accumulator variables into them.

--optimize-constructor-last-call**--optimise-constructor-last-call**

Enable the optimization of almost-last calls that are followed only by constructor application.

--no-split-switch-arms

When a switch on a variable has an inner switch on that same variable inside one of its arms, the default is to split up that arm of the outer switch along the same lines, effectively inlining the inner switch. '`--no-split-switch-arms`' prevents this split.

Optimization levels 2 to 6 automatically set `-split-switch-arms`.

--no-const-struct

By default, the compiler will gather constant ground structures in a separate table, with each such structure being stored in this table just once, even if it occurs in many different procedures. '`--no-const-struct`' prevents this behavior.

--common-struct

Replace two or more occurrences of the same term in a conjunction with just one copy.

Optimization levels 2 to 6 automatically set `-common-struct`.

--optimize-saved-vars**--optimise-saved-vars**

Minimize the number of variables saved across calls.

--constraint-propagation

Perform the constraint propagation transformation, which attempts to ensure that goals which can fail are executed as early as possible.

Optimization levels 3 to 6 automatically set `-constraint-propagation`.

--local-constraint-propagation

Perform the constraint propagation transformation, but only rearrange goals within each procedure. Specialized versions of procedures will not be created.

Optimization levels 3 to 6 automatically set `--local-constraint-propagation`.

--deforestation

Perform deforestation, which is a program transformation whose aims are to avoid the construction of intermediate data structures, and to avoid repeated traversals over data structures within a conjunction.

Optimization levels 3 to 6 automatically set `--deforestation`.

--deforestation-depth-limit *depth_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.

--delay-constructs**--delay-construct**

Reorder goals to move construction unifications after primitive goals that can fail.

Optimization levels 5 to 6 automatically set `--delay-constructs`.

--no-generate-trail-ops-inline

Normally, the compiler generates inline code for trailing operations. With `'--no-generate-trail-ops-inline'`, the compiler will implement them using calls to those operations in the standard library.

11.16.3 Optimizations during code generation

--smart-indexing

Implement switches using the fastest applicable implementation method, which may be e.g. binary search or a hash table. With `'--no-smart-indexing'`, the default is to implement switches as simple if-then-else chains.

Optimization levels 0 to 6 automatically set `--smart-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).
- lookup-switch-req-density *percentage***
The jump table generated for an atomic switch in which all the outputs are constant terms must have at least this percentage of full slots (default: 25).
- dense-switch-size *N***
The jump table generated for an atomic switch must have at least this many entries (default: 4).
- lookup-switch-size *N***
The lookup table generated for an atomic switch must have at least this many entries (default: 4).
- string-trie-switch-size *N***
--string-trie-size *N*
The trie generated for a string switch must have at least this many entries (default: 16).
- string-hash-switch-size *N***
--string-switch-size *N*
The hash table generated for a string switch must have at least this many entries (default: 8).
- string-binary-switch-size *N***
The binary search table generated for a string switch must have at least this many entries (default: 4).
- tag-switch-size *N***
The number of alternatives in a tag switch must be at least this number (default: 3).
- static-ground-terms**
Enable 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 always be created as static constants.
- Optimization levels 0 to 6 automatically set `--static-ground-terms`.
- use-atomic-cells**
Use the atomic variants of the Boehm gc allocator calls when the memory cell to be allocated cannot contain pointers.

11.16.4 Optimizations specific to high level code

`--mlds-optimize`

`--mlds-optimise`

Enable the MLDS->MLDS optimization passes.

Optimization levels 0 to 6 automatically set `--mlds-optimize`.

`--mlds-peephole`

Perform peephole optimization of the MLDS.

Optimization levels 0 to 6 automatically set `--mlds-peephole`.

`--optimize-tailcalls`

`--optimise-tailcalls`

Turn self-tailcalls into loops.

Optimization levels 1 to 6 automatically set `--optimize-tailcalls`.

`--optimize-initializations`

`--optimise-initializations`

Whenever possible, convert the first assignment to each local variable in the target code into an initializer on its declaration. Some target language compilers can generate faster code that way.

Optimization levels 2 to 6 automatically set `--optimize-initializations`.

`--eliminate-local-variables`

`--eliminate-local-vars`

Eliminate local variables with known values, where possible, by replacing occurrences of such variables with their values.

Optimization levels 5 to 6 automatically set `--eliminate-local-variables`.

11.16.5 Optimizations specific to low level code

`--try-switch-size N`

The number of alternatives in a try/retry chain switch must be at least this number (default: 3).

`--binary-switch-size N`

The number of alternatives in a binary search switch must be at least this number (default: 4).

`--middle-rec`

Enable the middle recursion optimization.

Optimization levels 1 to 6 automatically set `--middle-rec`.

`--simple-neg`

Generate simplified code for simple negations.

Optimization levels 2 to 6 automatically set `--simple-neg`.

`--llds-optimize`

`--llds-optimise`

Enable the LLDS->LLDS optimization passes.

Optimization levels 0 to 6 automatically set `--llds-optimize`.

`--optimize-repeat N`

`--optimise-repeat N`

Iterate most LLDS->LLDS optimizations at most *N* times (default: 3).

Optimization levels 0 to 1 automatically set `--optimize-repeat=1`.

Optimization level 2 automatically sets `--optimize-repeat=3`.

Optimization levels 3 to 4 automatically set `--optimize-repeat=4`.

Optimization levels 5 to 6 automatically set `--optimize-repeat=5`.

`--optimize-peep`

`--optimise-peep`

Enable local peephole optimizations.

Optimization levels 0 to 6 automatically set `--optimize-peep`.

`--optimize-labels`

`--optimise-labels`

Delete dead labels, and the unreachable code following them.

Optimization levels 0 to 6 automatically set `--optimize-labels`.

`--optimize-jumps`

`--optimise-jumps`

Enable the short-circuiting of jumps to jumps.

Optimization levels 0 to 6 automatically set `-optimize-jumps`.

`--optimize-fulljumps`

`--optimise-fulljumps`

Enable the elimination of jumps to ordinary code.

Optimization levels 2 to 6 automatically set `-optimize-fulljumps`.

`--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.

`--pessimize-tailcalls`

Disable the optimization of tailcalls. This option tries to minimize code size at the expense of speed.

`--optimize-delay-slot`

`--optimise-delay-slot`

Disable branch delay slot optimizations. This option is meaningful only if the target architecture has delay slots.

Optimization levels 1 to 6 automatically set `-optimize-delay-slot`.

`--optimize-frames`

`--optimise-frames`

Optimize the operations that maintain stack frames.

Optimization levels 1 to 6 automatically set `-optimize-frames`.

`--optimize-reassign`

`--optimise-reassign`

Optimize away assignments to memory locations that already hold the to-be-assigned value.

Optimization levels 3 to 6 automatically set `-optimize-reassign`.

`--use-local-vars`

Use local variables in C code blocks wherever possible.

Optimization levels 1 to 6 automatically set `-use-local-vars`.

`--optimize-dups`

`--optimise-dups`

Enable elimination of duplicate code within procedures.

Optimization levels 2 to 6 automatically set `-optimize-dups`.

`--optimize-proc-dups`

`--optimise-proc-dups`

Enable elimination of duplicate procedures.

`--common-data`

Enable optimization of common data structures.

Optimization levels 0 to 6 automatically set `-common-data`.

`--no-common-layout-data`

Disable optimization of common subsequences in layout structures.

`--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).

`--emit-c-loops`

Use C loop constructs to implement loops. With `'--no-emit-c-loops'`, use only gotos.

Optimization levels 1 to 6 automatically set `-emit-c-loops`.

`--procs-per-c-function N`

`--procs-per-C-function N`

Put the code for up to 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.

`--no-local-thread-engine-base`

Do not copy the thread-local Mercury engine base address into a local variable, even when this would be appropriate. This option is effective only in low-level parallel grades that do not use the GNU C global register variables extension.

`--inline-alloc`

Inline calls to `GC_malloc()`. This can improve performance a fair bit, but may significantly increase code size. This option is meaningful only if the selected garbage collector is boehm, and if the C compiler is gcc.

Optimization level 6 automatically sets `-inline-alloc`.

`--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. This produces slightly bigger but slightly faster code.

Optimization level 6 automatically sets `--use-macro-for-redo-fail`.

11.17 Intermodule optimization

11.17.1 Non-transitive intermodule optimization

`--intermodule-optimization`

`--intermodule-optimisation`

`--intermod-opt`

Perform inlining, higher-order specialization, and other optimizations using knowledge of the otherwise non-public data of directly imported modules. The compiler gets this data from the `'module.opt'` files of the directly imported modules.

This option must be set consistently throughout the compilation process, starting with `mmc --generate-dependencies`.

`--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.

`--no-read-opt-files-transitively`

Only read the inter-module optimization information for directly imported modules, not the transitive closure of the imports.

11.17.2 Transitive intermodule optimization

`--transitive-intermodule-optimization`

`--transitive-intermodule-optimisation`

`--trans-intermod-opt`

Perform inlining, higher-order specialization, and other optimizations using knowledge of the otherwise non-public data of both directly and indirectly imported modules. The compiler gets this data from the `'module.trans_opt'` files of the directly and indirectly imported modules.

This option must be set consistently throughout the compilation process, starting with `mmc --generate-dependencies`.

Note that `'--transitive-intermodule-optimization'` works only with `mmake`; it does not work with `mmc --make`.

`--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.

`--also-output-module-order`

`--generate-module-order`

If ‘`--generate-dependencies`’ is also specified, then output the strongly connected components of the module dependency graph in top-down order to ‘`module.module_order`’. If ‘`--generate-dependencies`’ is not specified, then this option does nothing.

11.18 Program analyses

11.18.1 The termination analyser based on linear inequality constraints

`--enable-termination`

`--enable-term`

Enable termination analysis, which analyses each mode of each predicate or function to see whether it terminates. The `terminates`, `does_not_terminate`, and `check_termination` pragmas have an effect only when termination analysis is enabled.

Note that both ‘`--intermodule-optimization`’ and ‘`--transitive-intermodule-optimization`’ greatly improve the accuracy of the analysis.

`--check-termination`

`--check-term`

`--chk-term`

Enable termination analysis, and emit warnings for some predicates or functions that cannot be proved to terminate.

In many cases where the compiler is unable to prove termination, the problem is either a lack of information about the termination properties of other predicates, or the use of language constructs (such as higher order calls) which are beyond the capabilities of the analyser. In these cases, the compiler does not emit a warning of non-termination, as it is likely to be spurious.

`--verbose-check-termination`

`--verb-check-term`

`--verb-chk-term`

Enable termination analysis, and emit warnings for all predicates or functions that cannot be proved to terminate.

`--termination-single-argument-analysis N`

`--term-single-arg N`

When performing termination analysis, try analyzing recursion on single arguments in strongly connected components of the call graph that have up to the

given number of procedures. Setting this limit to zero disables single argument analysis.

`--termination-norm {simple, total, num-data-elems}`

`--term-norm {simple, total, num-data-elems}`

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.

`--termination-error-limit N`

`--term-err-limit N`

Print at most this number of reasons for any single termination error (default: 3).

`--termination-path-limit N`

`--term-path-limit N`

Perform termination analysis only on predicates with at most this many paths (default: 256).

11.18.2 Other program analyses

`--analyse-exceptions`

Enable exception analysis. which tries to identify procedures that will not throw an exception. Some optimizations can make use of this information.

`--analyse-trail-usage`

Enable trail usage analysis, which tries to identify procedures that will not modify the trail. The compiler can use this information to reduce the overhead of trailing.

`--analyse-mm-tabling`

Identify those goals that do not call procedures that are evaluated using minimal model tabling. The compiler can use this information to reduce the overhead of minimal model tabling.

11.19 Options that ask for modified output

`-n`

`--line-numbers`

Put source line numbers into the generated code. The generated code may be in C, Java or C# (the usual cases), or in Mercury (with ‘`--convert-to-mercury`’).

`--no-line-numbers-around-foreign-code`
Do not put source line numbers into the generated target language file 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).

`--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`’ option.)

11.20 Options for controlling `mmc --make`

`-k`
`--keep-going`
With ‘`--make`’, keep going as far as possible even if an error is detected.

`--order-make-by-timestamp`
Tell `mmc --make` to compile more recently modified source files first.

`--show-make-times`
Report run times for commands executed by `mmc --make`.

`-j N`
`--jobs N` With ‘`--make`’, attempt to perform up to ‘`N`’ jobs concurrently.

`--track-flags`
`--track-options`
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 options, verbosity options, and build system options are not tracked.

`--pre-link-command command`
`-no-pre-link-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.

11.21 Options for target language compilation

11.21.1 General options for compiling target language code

`--target-debug`

`--c-debug`

`--java-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.

`--no-warn-target-code`

Disable warnings from the compiler (which may be e.g. gcc) that processes the target code generated by mmc.

11.21.2 Options for compiling C code

`--cc compiler-name`

Specify which C compiler to use.

`--c-optimize`

`--c-optimise`

Enable optimizations by the C compiler.

Optimization levels 1 to 6 automatically set `-c-optimize`.

`--c-include-directory directory`

`-no-c-include-directory`

`--c-include-dir directory`

`-no-c-include-dir`

Append *directory* to the list of directories to be searched for C header files. Note that if you want to override this list, instead of appending to it, then you can set the `MERCURY_MC_ALL_C_INCL_DIRS` environment variable to a sequence of ‘`--c-include-directory`’ options.

`--cflags <options>`

`-no-cflags` Specify options to be passed to the C compiler. These options will not be quoted when passed to the shell.

`--quoted-cflag option`

`--cflag option`

Specify a single word option to be passed to the C compiler. The word will be quoted when passed to the shell.

11.21.3 Options for compiling Java code

`--java-compiler javac`

`--javac javac`

Specify which Java compiler to use. The default is 'javac'.

`--java-interpreter java`

Specify which Java interpreter to use. The default is 'java'.

`--javac-flags options`

`-no-javac-flags`

`--java-flags options`

`-no-java-flags`

Specify options to be passed to the Java compiler. These options will not be quoted when passed to the shell.

`--quoted-javac-flag option`

`--quoted-java-flag option`

`--javac-flag option`

`--java-flag option`

Specify a single word option to be passed to the Java compiler. The word will be quoted when passed to the shell.

`--java-classpath path`

`-no-java-classpath`

Set the classpath for the Java compiler and interpreter.

`--java-runtime-flags options`

`-no-java-runtime-flags`

Specify options to be passed to the Java interpreter. These options will not be quoted when passed to the shell.

`--quoted-java-runtime-flag option`

`--java-runtime-flag option`

Specify a single word option to be passed to the Java interpreter. The word will be quoted when passed to the shell.

11.21.4 Options for compiling C# code

`--csharp-compiler csc`
Specify the name of the C# Compiler. The default is 'csc'.

`--cli-interpreter prog`
Specify the program that implements the Common Language Infrastructure (CLI) execution environment, e.g. 'mono'.

`--csharp-flags options`
`-no-csharp-flags`
Specify options to be passed to the C# compiler. These options will not be quoted when passed to the shell.

`--quoted-csharp-flag option`
`--csharp-flag option`
Specify a single word option to be passed to the C# compiler. The word will be quoted when passed to the shell.

`--mono-path-directory directory`
`-no-mono-path-directory`
`--mono-path-dir directory`
`-no-mono-path-dir`
Specify a directory to add to the runtime library assembly search path passed to the Mono CLR using the MONO_PATH environment variable.

11.22 Options for linking

11.22.1 General options for linking

`--mercury-library-directory directory`
`--mld 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 8.2.2 \[Using installed libraries with mmc -make\]](#), page 35.

`--search-library-files-directory directory`
`--search-lib-files-dir directory`
Search *directory* for Mercury library files that 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.

`--mercury-library library`

`--ml library`

Link with the specified Mercury library. See [Section 8.2.2 \[Using installed libraries with `mmc --make`\]](#), page 35.

`--mercury-standard-library-directory directory`

`--no-mercury-standard-library-directory`

`--mercury-stdlib-dir directory`

`--no-mercury-stdlib-dir`

Search *directory* for the Mercury standard library. Implies ‘`--mercury-library-directory directory`’ and ‘`--mercury-configuration-directory directory`’. The negative version, ‘`--no-mercury-standard-library-directory`’, tells the compiler not to use the Mercury standard library, and also implies ‘`--no-mercury-configuration-directory`’.

11.22.2 Options for linking C or C# code

`-L directory`

`-L-`

`--library-directory directory`

`--no-library-directory`

Append *directory* to the list of directories in which to search for libraries.

`-l library`

`-l-`

`--library library`

`--no-library`

Link with the specified library.

11.22.3 Options for linking just C code

`-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.) This option is ignored by `mmc --make`.

`--link-object filename`

`--no-link-object`

Link with the specified object file, or archive of object files.

`--ld-flags options`

`--no-ld-flags`

Specify options to be passed to the linker command that will create an executable. These options will not be quoted when passed to the shell. Use `mmc --output-link-command` to find out what the linker command is.

`--quoted-ld-flag option`

`--ld-flag option`

Specify a single word option to be passed to the linker command that will create an executable. The word will be quoted when passed to the shell. Use `mmc --output-link-command` to find out what the linker command is.

`--ld-libflags options`

`-no-ld-libflags`

Specify options to be passed to the linker command that will create a shared library. These options will not be quoted when passed to the shell. Use `mmc --output-shared-lib-link-command` to find out what the linker command is.

`--quoted-ld-libflag option`

`--ld-libflag option`

Specify a single word option to be passed to the linker command that will create a shared library. The word will be quoted when passed to the shell. Use `mmc --output-shared-lib-link-command` to find out what the linker command is.

`-R directory`

`-R-`

`--runtime-library-directory directory`

`-no-runtime-library-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.

`--init-file-directory directory`

`-no-init-file-directory`

Append *directory* to the list of directories to be searched for `.init` files by `c2init`.

`--init-file init-file`

`-no-init-file`

Append *init-file* to the list of `.init` files to be passed to `c2init`.

`--trace-init-file init-file`

`-no-trace-init-file`

Append *init-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`
Do not pipe the output of the linker through the Mercury demangler. The demangler usually makes it easier to understand the diagnostics for any link errors that involve code generated by the Mercury compiler.
- `--no-strip`
Do not invoke the `strip` command on executables. Stripping minimizes executables’ sizes, but also makes debugging them much harder.
- `--no-main`
Do not generate a C `main()` function. With ‘`--no-main`’, the user’s own 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`
`-no-runtime-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-written C code with `INIT` comments, rather than containing only C code that was automatically generated by the Mercury compiler.)
- `--framework framework`
`-no-framework`
Build and link against the specified framework. (Mac OS X only.)
- `-F directory`
`-F-`
`--framework-directory directory`
`-no-framework-directory`
Append the specified directory to the framework search path. (Mac OS X only.)

- `--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.
- `--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.
- `--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 above (such as Mac OS X).
- `--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.
- `--strip-executable-static-flags options`
Specify options to pass to the strip executable command when linking against Mercury static libraries.

11.22.4 Options for linking just Java code

- `--java-archive-command command`
Specify the command used to produce Java archive (JAR) files.

11.22.5 Options for linking just C# code

- `--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.)

11.23 Options controlling searches for files

- `--options-search-directory directory`
`-no-options-search-directory`
Add *directory* to the list of directories to be searched for options files.

--use-subdirs

Generate intermediate files in a ‘Mercury’ subdirectory, rather than generating them 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’ is supported only by `mmc --make`; it does not work with `mmake`.

-I *directory*

-I-

--search-directory *directory*

-no-search-directory

Append *directory* to the list of directories to be searched for ‘.int*’ and ‘.module_dep’ files.

--intermod-directory *directory*

-no-intermod-directory

Add *directory* to the list of directories to be searched for ‘.opt’ and ‘.trans_opt’ files.

--no-use-search-directories-for-intermod

With ‘--use-search-directories-for-intermod’, the compiler will add the arguments of ‘--search-directory’ options to the list of directories to search for ‘.opt’ files. With ‘--no-use-search-directories-for-intermod’, the compiler will use only the arguments of ‘--intermod-directory’ options.

11.24 Options controlling the library installation process

--install-prefix *directory*

The directory under which to install Mercury libraries.

--library-grade *grade*

-no-library-grade

--libgrade *grade*

-no-libgrade

The positive form adds *grade* to the list of compilation grades in which a library to be installed should be built. (The list is initialized to the set of grades in which the standard library is installed.) The negative form clears the list of compilation grades in which a library to be installed should be built.

`--libgrades-include-component grade_component`
`--no-libgrades-include-component`
`--libgrades-include grade_component`
`--no-libgrades-include`
Remove grades that do not contain the specified component from the set of library grades to be installed. (This option works only with `mmc --make`; it does not work with `mmake`.)

`--libgrades-exclude-component grade_component`
`--no-libgrades-exclude-component`
`--libgrades-exclude grade_component`
`--no-libgrades-exclude`
Remove grades that contain the specified component from the set of library grades to be installed. (This option works only with `mmc --make`; it does not work with `mmake`.)

`--library-install-linkage {shared, static}`
`--no-library-install-linkage`
`--library-linkage {shared, static}`
`--no-library-linkage`
`--lib-linkage {shared, static}`
`--no-lib-linkage`
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.

`--no-detect-stdlib-grades`
`--no-detect-libgrades`
Do not scan the installation directory to determine which standard library grades are available.

`--no-libgrade-install-check`
Do not check that libraries have been installed before attempting to use them. (This option is meaningful only with `mmc --make`.)

`--extra-init-command command`
`--no-extra-init-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.

`--extra-library-header filename`
`-no-extra-library-header`
`--extra-lib-header filename`
`-no-extra-lib-header`
 Install the specified C header file along with a Mercury library. (This option is supported only by `mmc --make`.)

11.25 Options specifying properties of the environment

`--mercury-configuration-directory directory`
`--mercury-config-dir directory`
 Search *directory* for Mercury system's configuration files.

`--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`.

`--options-file filename`
`-no-options-file`
 Add *filename* to the list of options files to be processed. If *filename* is '-', an options file will be read from the standard input. By default, the compiler will read the file named 'Mercury.options' in the current directory. Note that this option is intended to be used only on command lines. When specified in options files (i.e. in files named by other `-options-file` options), it has no effect.

`--config-file filename`
`-no-config-file`
 Read the Mercury compiler's configuration information from *filename*. 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 7 \[Using Mmake\]](#), page 28.

`--env-type {posix,cygwin,msys,windows}`
 Specify the environment type in which the compiler and generated programs will be invoked. This option is equivalent to setting all of '`--host-env-type`', '`--system-env-type`' and '`--target-env-type`' to the given environment type.

`--host-env-type {posix,cygwin,msys,windows}`
 Specify the environment type in which the compiler will be invoked.

`--system-env-type {posix,cygwin,msys,windows}`
 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 {posix,cygwin,msys,windows}`
Specify the environment type in which generated programs will be invoked.

`--restricted-command-line`
Enable this option if your shell does not support long command lines. This option uses temporary files to pass arguments to sub-commands. (This option is supported only by `mmc --make`.)

11.26 Options that record autoconfigured parameters

`--num-real-r-regs N`
Assume registers ‘*r1*’ up to ‘*rN*’ are real (i.e. not virtual) general purpose registers. Note that the value of this option is normally autoconfigured; its use should never be needed except for cross-compilation.

`--num-real-f-regs N`
Assume registers ‘*f1*’ up to ‘*fN*’ are real (i.e. not virtual) floating point registers. Note that the value of this option is normally autoconfigured; its use should never be needed except for cross-compilation.

`--num-real-r-temps N`
`--num-real-temps N`
Assume that ‘*N*’ non-float temporaries will fit into real machine registers. Note that the value of this option is normally autoconfigured; its use should never be needed except for cross-compilation.

`--num-real-f-temps N`
Assume that ‘*N*’ float temporaries will fit into real machine registers. Note that the value of this option is normally autoconfigured; its use should never be needed except for cross-compilation.

`--max-jump-table-size N`
Specify the maximum number of entries a jump table may 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.

11.27 Options for developers only

11.27.1 Operation selection options for developers only

`--control-granularity`
Don’t try to generate more parallelism than the machine can handle, which may be specified at runtime or detected automatically.

11.27.2 Dumping out internal compiler data structures

`-d` *stage number or name*

`-d-`

`--dump-hlds` *stage number or name*

`--no-dump-hlds`

`--hlds-dump` *stage number or name*

`--no-hlds-dump`

Dump the HLDS (high level intermediate representation) after the specified stage to `'module'.hlds_dump,'num'-'name'`. Stage numbers range from 1-599. Multiple dump options accumulate.

`--dump-hlds-pred-name` *name*

`--no-dump-hlds-pred-name`

Dump the HLDS only of the predicate/function with the given name.

`--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 option argument (see the Mercury User's Guide for details).

`--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.

11.27.3 Options intended for internal use by the compiler only

`--generate-mmc-make-module-dependencies`

`--generate-mmc-deps`

Generate dependencies for use by `mmc --make` even when using `mmake`. This is recommended when building a library for installation.

11.28 Now-unused former options kept for compatibility

`--no-ansi-c`

This option is deprecated and does not have any effect.

`--everything-in-one-c-function`

`--everything-in-one-C-function`

This option is deprecated and does not have any effect.

12 Environment variables

The various components of the Mercury compilation environment will use the following environment variables if they are set. There should be little need to use most of these, because the default values will generally work fine.

12.1 Environment variables affecting the Mercury compiler

MERCURY_COMPILER

Filename of the Mercury compiler.

MERCURY_MKINIT

Filename of the program to create the `*_init.c` file.

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_COLOR_SCHEME

The specification of the map from color roles to color shades that the Mercury compiler should use in diagnostics. For the details of the syntax, semantics, and effects of such specifications, please see [Section 13.3 \[Color schemes\]](#), page 175.

MERCURY_ENABLE_COLOR

This environment variable should contain either `always` or `1` to enable the use of color in diagnostics by the Mercury compiler, or either `never` or `0` to disable it. Note however that this effect can be overruled by command line options; for the details, please see [Section 13.4 \[Enabling the use of color\]](#), page 177.

`NO_COLOR` This environment variable being set to any nonempty string will disable the use of color in diagnostics by the Mercury compiler. Note however that this effect can be overruled by command line options and by the `MERCURY_ENABLE_COLOR` environment variable; for the details, please see [Section 13.4 \[Enabling the use of color\]](#), page 177.

12.2 Environment variables affecting the Mercury debugger

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.

12.3 Environment variables affecting the Mercury runtime system

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_prognam` apply only to programs named *prognam*. Note that *prognam* does *not* include the `.exe` extension on those systems (e.g. Windows) that use it. 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_prognam` 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-kwargs size`
Sets the size of the cut stack to *size* kilobytes multiplied by the word size in bytes.

`--pneystack-size size`
Sets the size of the pneg stack to *size* kilobytes.

`--pneystack-size-kwargs 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* (excluding any `'.exe'` extension on Windows). 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* (excluding any `'.exe'` extension on Windows). 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.

`--deep-std-name`

This option, which is meaningful only in deep profiling grades, asks deep profiling data to be stored in files named `Deep.data` and `Deep.procrep`. The default is that the runtime system puts such data into files whose names follow the pattern `progrname_on_date_at_time.data` and `progrname_on_date_at_time.procrep` respectively.

`--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.

13 Diagnostic output

Mercury’s strong type-, mode-, and determinism systems impose strict requirements on Mercury programs. These systems allow the compiler to guarantee that it will catch many of the most frequent kinds of errors that programmers tend to make. This improves not just program reliability, but also programmer productivity, since finding and fixing a bug is much easier if the compiler hands you the location of at least one part of the program involved in the error on a silver platter. However, it also means that a Mercury programmer

will typically have to see and act upon more error and warning messages (diagnostics) than programmers who write code in less strict languages. This is why the compiler tries hard to make each diagnostic as usable, informative and clear as it can.

13.1 Verbose vs nonverbose messages

Making a diagnostic sufficiently informative for programmers new to Mercury does in some cases require including a significant amount of explanation. On the other hand, such explanations are not needed by programmers who are already familiar with the issues involved, and indeed for them, such explanations are usually nothing more than clutter. This is why the Mercury compiler splits such error messages into two parts. The first part just reports the facts specific to the error being reported. The second part, which is intended specifically to help programmers new to Mercury, is a generic explanation of the background needed to understand the first part. The first part is always printed, while the second part is printed only if the programmer asks for verbose error messages. This can be done using the `'--verbose-error-messages'` compiler option, which can be abbreviated to just `'-E'`. If a compiler invocation prints any diagnostics that have a verbose component that is not printed because the user has not asked for it, the compiler will print the reminder about this fact: `'For more information, recompile with '-E'.'`

13.2 Ordering diagnostics

Normally, the compiler prints all the diagnostics for a module together, in ascending order of context. (With the exception of Mercury modules whose source files use the `'source_file'` pragma and/or the `'#line'` directive, this means that messages for lower numbered lines are printed before messages for higher numbered lines.)

Sometimes, the output will *look* unsorted without actually *being* unsorted. This can happen because some diagnostics report an inconsistency between two or more parts of the program. Such diagnostics will contain one component for each such part, and each component's text about that part will have before it the context (filename and line number) of that part. The sorting of diagnostics operates on just the first context in each diagnostic; any later contexts are ignored.

However, there are some situations in which the compiler does indeed intentionally deviate from the above default.

One such situation occurs when the programmer specifies the `'-v'` option, which asks for verbose compiler output. In such cases, the compiler will print progress messages that specify which phase of compilation the compiler is about to start, and if that phase can find errors in the program, it will print the diagnostics for the errors it finds *before* it goes on to the next phase.

The programmer can ask for the usual ordering of diagnostics to be reversed by giving the `'--reverse-error-order'` option. This option makes things easier for programmers when they want to work on the last errors in a file first. This can be a good idea for example if the module you are working on (e.g. by rearranging a data structure) has its primitive operations towards the end of the module, and you think the fixes needed to these primitives will require changes to their signatures. In such cases, fixing the lowest level primitives first allows you to fix the originally reported errors for higher level operations at the same time as you update their calls to the lower level primitives, and reversing the

order of the diagnostics allows you to get to those errors for the tail end of the file without having to wade through the errors for the code above.

The compiler also supports a more general mechanism for allowing programmers to focus on fixing errors in a particular part of the program first. This mechanism is the ‘`--limit-error-contexts`’ compiler option. This option has as its argument a comma-separated list of line number ranges. It tells the compiler to print only diagnostics that contain one or more contexts that fall into one of these ranges. (A printed diagnostic may contain some contexts that fall outside all these ranges, as long as it has at least one context that falls inside.) If a compiler invocation has this option specified but all the diagnostics it would print in its absence are held back by this option, it will print a reminder about this fact.

13.3 Color schemes

The compiler tries to make diagnostics usable more quickly by using color to direct people’s attention to the most relevant parts of diagnostics.

The compiler uses colors for the following five purposes. We must explain these first, before we can explain *color schemes*, which are simple mappings from purposes (which we also call *color roles*) to actual shades of color.

- ‘**subject**’ The compiler uses the ‘**subject**’ color for the entity that the diagnostic is about. Examples of such entities include a specific variable, a specific type, or a specific predicate.
- ‘**incorrect**’ The compiler uses the ‘**incorrect**’ color to emphasize either *the description* of what is wrong with the subject entity, or *the property* of the subject entity that is wrong.
- ‘**correct**’ The compiler uses the ‘**correct**’ color to emphasize the expected property of the subject entity, where that differs from its actual property. For example, when a call passes an argument of the wrong type, it will show the argument’s actual type using the ‘**incorrect**’ color, and the expected type using the ‘**correct**’ color. It does this because the type is far more likely to be wrong in the call than in the declaration of the called predicate or function. (Of course, situations do sometimes occur in which it is the type in the declaration that is wrong.)
- ‘**inconsistent**’ The compiler uses the ‘**inconsistent**’ color when it knows that two parts of the code are inconsistent with each other, but neither part is a priori more likely to be wrong than the other. For example, when unifying two values of different types, the compiler uses this color when printing the type of both values; the compiler knows that in all likelihood, one type is correct and one is incorrect, but it does not know which is which.
- ‘**hint**’ The compiler uses the ‘**hint**’ color when it reports observations that may or may not explain the root of the problem it reports, but which are worth looking into. For example, when it reports that two actual arguments of a call have types that do not match the types of the corresponding arguments in the declaration of the callee, it may give a hint that each of those actual arguments do in fact

each match the type of an argument in the callee, just not the ones in the right positions, implying that the actual problem may be passing the right arguments in the wrong order.

Programmers can specify the colors they want the compiler to use for each purpose either by setting the environment variable ‘MERCURY_COLOR_SCHEME’ to a string (let’s call it *ColorScheme*), or by specifying the compiler option ‘--color-scheme *ColorScheme*’. In both cases, *ColorScheme* can be either

- the name of a color scheme built into the Mercury compiler, or
- the assignment of specific colors for the roles that colors can play in its diagnostics.

The compiler currently has the following six builtin color schemes:

- ‘darkmode’
- ‘darkmode256’
- ‘darkmode16’
- ‘lightmode’
- ‘lightmode256’
- ‘lightmode16’

The “mode” in each name is optional, meaning that e.g. ‘dark’ names the same color scheme as ‘darkmode’.

The schemes whose names contain “dark” are intended to be used on terminal screens with dark backgrounds (including black backgrounds). The schemes whose names contain “light” are intended to be used on terminal screens with light backgrounds (including white backgrounds).

- The schemes ending without numbers use 24-bit RGB colors, which your terminal or terminal emulator *probably* supports.
- The schemes ending in “256” use the colors listed in the 8-bit color section under ‘https://en.wikipedia.org/wiki/ANSI_escape_code#Colors’, which your terminal or terminal emulator is *virtually certain* to support.
- The schemes ending in “16” use the first 16 color slots listed in that 8-bit color table. These 16 color slots differ from the other 240 colors in that table in that most terminal emulators allow users to select the actual colors that go into those slots. This means that output that uses color schemes ending in “16” may look different on terminals with different color palette settings, and will match the intended look only with the default color palette. (The builtin color schemes ending in “256” avoid using these reassignable color slots.)

The other way to specify a color scheme is to directly specify the colors you want the compiler to use using a string such as ‘specified@subject=87:correct=40:incorrect=203:inconsistent=171:h’. The rules for this form of color scheme specification are as follows.

- The string must start with the string ‘specified@’.
- The rest of the string must not contain any white space, but must consist of a colon-separated list of one or more color assignments.
- Each color assignment must have the form ‘*Role=Color*’, which assigns the given color to the given role.

- Each *Role* must be one of the five strings ‘subject’, ‘correct’, ‘incorrect’, ‘inconsistent’, and ‘hint’.
- Each *Color* must have one of the following two forms.
 - The first form is a decimal integer in the range 0 to 255 (both inclusive). This selects a slot in the table in the 8-bit color section of ‘https://en.wikipedia.org/wiki/ANSI_escape_code#Colors’. The caveat mentioned above about colors 0 to 15 apply here as well: in most terminal emulators, these slots are reassignable, so specifying one of these slots may *or may not* get you to the color in that slot in that table. The other 240 slots are not usually reassignable.
 - The second form is a seven-character string where the first character is ‘#’, and each of the following six characters is a hexadecimal digit. Each string ‘#RRGGBB’ specifies a 24-bit RGB color, with ‘RR’ specifying its red component, ‘GG’ specifying its green component, and ‘BB’ specifying its blue component.
- If the color scheme has no color assignment for a given role, then the compiler will not use color for that role.
- If the color scheme has two or more color assignments for the same role, only the last one counts; the others are ignored.

As mentioned above, programmers can specify color schemes using the environment variable ‘MERCURY_COLOR_SCHEME’ and by using the ‘--color-scheme’ compiler option. These can specify different schemes. The rules the compiler uses to choose the color scheme that it will use are as follows.

1. If the command line contains a ‘--color-scheme’ option, the compiler will use its color scheme. (If the command line contains more than one ‘--color-scheme’ option, the compiler will use the last one.)
2. If the command line contains no ‘--color-scheme’ option, but the environment variable ‘MERCURY_COLOR_SCHEME’ exists and contains a nonempty string, then the compiler will use its color scheme.
3. If the command line contains no ‘--color-scheme’ option, and the environment variable ‘MERCURY_COLOR_SCHEME’ does not exist or contains an empty string, but the files named in any ‘--options-file’ compiler options (or, in the absence of such options, the ‘Mercury.options’ file), include a ‘--color-scheme’ option among the module-specific ‘MCFLAGS’ for the current module, then the compiler will use its color scheme. (If the options file(s) contain more than one ‘--color-scheme’ option, the compiler will use the last one.)
4. In the absence of a ‘--color-scheme’ option on the command line, in the ‘MERCURY_COLOR_SCHEME’ environment variable *and* in the options files, the compiler will use its default color scheme, which is ‘light16’. This scheme works reasonably well even on dark backgrounds, even though (as its name indicates) it was designed for light backgrounds. (This assumes that the color slots it uses hold either their standard colors, or something reasonably close to them.)

13.4 Enabling the use of color

The Mercury compiler uses two separate mechanisms

- to specify what colors to use for each role, *if* the use of color in diagnostics is enabled, and
- to decide whether the use of color in diagnostics is enabled.

This arrangement allow programmers to turn off the use of color temporarily without having to specify the color scheme again in full when they want to turn it back on again.

As with color schemes, programmers can control whether color is enabled in diagnostics using either a compiler option, or by using environment variables.

The compiler option is a boolean option, which, like other boolean options, has two forms: a positive form ‘`--color-diagnostics`’, which enables the use of color, and a negative form ‘`--no-color-diagnostics`’, which disables its use.

There are two environment variables that the Mercury compiler consults in its decision: ‘`MERCURY_ENABLE_COLOR`’, and ‘`NO_COLOR`’. As their names show, the first is specific to Mercury, while the second is not. It is in fact a defacto standard for disabling color; see ‘<https://no-color.org>’.

The rules the compiler uses to decide whether the use of color in diagnostics is enabled are as follows.

1. If the command line contains a ‘`--color-diagnostics`’ option, in either its positive or negative form, then the compiler will obey it.
2. If the command line contains no ‘`--color-diagnostics`’ option, but the environment variable ‘`MERCURY_ENABLE_COLOR`’ exists and contains either ‘`never`’ or ‘`always`’, then the compiler will obey it. The compiler also allows ‘`0`’ as a synonym for ‘`never`’, and ‘`1`’ as a synonym for ‘`always`’.
3. If the command line contains no ‘`--color-diagnostics`’ option, and the environment variable ‘`MERCURY_ENABLE_COLOR`’ either does not exist, or contains a string other than the four listed above, but the environment variable ‘`NO_COLOR`’ exists and contains a nonempty string, then the compiler will obey it by disabling the use of color.
4. If the command line contains no ‘`--color-diagnostics`’ option and neither environment variable has a value that directs the compiler to obey it, but the files named in any ‘`--options-file`’ compiler options, (or, in the absence of such options, the ‘`Mercury.options`’ file), include a ‘`--color-diagnostics`’ option, in either its positive or negative form, among the module-specific ‘`MCFLAGS`’ for the current module, then the compiler will obey it.
5. If the preconditions of all four of the above rules are false, i.e. none of them decide the outcome, then the compiler will fall back to its default. The usual default decision is to enable the use of color, though this can be overridden for a given installation of Mercury.

14 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`.

15 Foreign language interface

Mercury allows the code of a procedure to be specified in a target language using a `pragma foreign_proc` declaration. A procedure may have more than one `pragma foreign_proc` declarations, provided they are in different languages. A procedure that has some `pragma foreign_proc` declarations may also have a Mercury definition.

If the language specified in a `foreign_proc` is not available for the selected backend, it will be ignored, and the implementation will use the procedure's Mercury definition. If there isn't one, the compiler will generate an error.

'C'	This is the only foreign language whose <code>pragma foreign_procs</code> are used when targeting C.
'C#'	This is the only foreign language whose <code>pragma foreign_procs</code> are used when targeting C#.
'Java'	This is the only foreign language when whose <code>pragma foreign_procs</code> are used targeting Java.

16 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.

You can create a stand-alone interface 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 8 \[Libraries\]](#), [page 34](#)). 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, like this:

```
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 itself attempt to determine the address of the base of the stack. 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-non-contiguity-for	135
--allow-stubs	124
--allow-undefined	160
--also-output-module-order	152
--analyse-exceptions	153
--analyse-mm-tabling	153
--analyse-trail-usage	153
--ansi-c	166
--asm-labels	120
--auto-comments	154
--binary-switch-size	147
-- Boehm GC calc-time (runtime option)	173
-- Boehm GC free-space-divisor (runtime option)	173
--c-debug	155
--c-include-dir	155
--c-include-directory	155
--c-optimise	155
--c-optimize	155
--C#	120
--C#-only	120
--cc	155, 178
--cflag	156
--cflags	155
--check-term	152
--check-termination	152
--checked-nondet-tailcalls	149
--chk-term	152
--cli-interpreter	157
--color-diagnostics	126
--color-scheme	126
--colour-diagnostics	126
--colour-scheme	126
--common-data	150
--common-layout-data	150
--common-struct	144
--compile-only	24, 116
--compile-to-c	120
--compile-to-C	120
--config-file	164
--const-struct	144
--constraint-propagation	144
--control-granularity	165
--convert-to-mercury	116
--convert-to-Mercury	116
--coverage-profiling	122
--coverage-test (runtime option)	172
--coverage-test-if-exec (runtime option)	172
--csharp	120
--csharp-compiler	157
--csharp-flag	157
--csharp-flags	157
--csharp-only	120
--cstack-reserve-size	161
--cutstack-size (runtime option)	170
--cutstack-size-kwargs (runtime option)	171
--debug	39, 121
--debug-threads (runtime option)	171
--decl-debug	121
--deep-profiling	122
--deep-std-name	173
--default-runtime-library-directory	159
--deforestation	145
--deforestation-depth-limit	145
--deforestation-size-threshold	145
--deforestation-vars-threshold	145
--delay-construct	145
--delay-constructs	145
--delay-death	140
--delay-death-max-vars	140
--demangle	107, 160
--dense-switch-req-density	146
--dense-switch-size	146
--detect-libgrades	163
--detect-stdlib-grades	163
--detstack-size	26
--detstack-size (runtime option)	169
--detstack-size-kwargs (runtime option)	169
--dump-hlds	166
--dump-hlds-options	166
--dump-hlds-pred-name	166
--dump-same-hlds	166
--eliminate-local-variables	147
--eliminate-local-vars	147
--emit-c-loops	150
--enable-term	152
--enable-termination	152
--env-type	164
--error-files-in-subdir	126
--errorcheck-only	116
--event-set-file-name	139
--everything-in-one-c-function	166
--everything-in-one-C-function	166
--exec-trace-tail-rec	139
--extra-init-command	163
--extra-initialization-functions	160
--extra-inits	160
--extra-lib-header	164
--extra-library-header	164
--filenames-from-stdin	114
--flags	114
--flags-file	114
--fp-rounding-mode (runtime option)	173
--framework	160
--framework-directory	160
--fully-strict	124
--garbage-collection	122

--gc	122	--inline-vars-threshold	143
--gcc-global-registers	120	--inlining	142
--gcc-non-local-gotos	120	--install-command	164
--generate-dependencies	115	--install-prefix	162
--generate-dependencies-ints	115	--intermod-directory	162
--generate-dependency-file	115	--intermod-inline-simple-threshold	142
--generate-mmc-deps	166	--intermod-opt	151
--generate-mmc-make-module-dependencies	166	--intermod-unused-args	141
--generate-module-order	152	--intermodule-optimisation	151
--generate-source-file-mapping	114	--intermodule-optimization	37, 38, 151
--generate-standalone-interface	116	--introduce-accumulators	144
--generate-trail-ops-inline	145	--java	120
--genstack-size (runtime option)	170	--Java	120
--genstack-size-kwargs (runtime option) ..	170	--java-archive-command	161
--grade	119	--java-classpath	156
--halt-at-invalid-interface	137	--java-compiler	156
--halt-at-syntax-errors	136	--java-debug	155
--halt-at-warn	136	--java-flag	156
--halt-at-warn-make-int	136	--java-flags	156
--halt-at-warn-make-interface	136	--java-interpreter	156
--halt-at-warn-make-opt	136	--java-only	120
--heap-size (runtime option)	169	--Java-only	120
--heap-size-kwargs (runtime option)	169	--java-runtime-flag	156
--help	113, 114	--java-runtime-flags	156
--high-level-c	121	--javac	156
--high-level-code	107, 121	--javac-flag	156
--high-level-C	121	--javac-flags	156
--higher-order-arg-limit	143	--jobs	154
--higher-order-size-limit	143	--keep-going	154
--highlevel-c	121	--layout-compression-limit	150
--highlevel-code	121	--ld-flag	159
--highlevel-C	121	--ld-flags	158
--hlds-dump	166	--ld-libflag	159
--host-env-type	164	--ld-libflags	159
--imports-graph	139	--lib-linkage	163
--infer-all	123	--libgrade	162
--infer-det	124	--libgrade-install-check	163
--infer-determinism	124	--libgrades-exclude	163
--infer-modes	124	--libgrades-exclude-component	163
--infer-types	123	--libgrades-include	163
--inform-incomplete-switch	133	--libgrades-include-component	163
--inform-incomplete-switch-threshold	133	--library	158
--inform-inferred	137	--library-directory	158
--inform-inferred-modes	137	--library-grade	162
--inform-inferred-types	137	--library-install-linkage	163
--inform-ite-instead-of-switch	133	--library-linkage	163
--inform-suboptimal-packing	137	--limit-error-contexts	126
--inhibit-style-warnings	136	--line-numbers	153
--inhibit-warnings	136	--line-numbers-around-foreign-code	154
--init-file	36, 159	--line-numbers-for-c-headers	154
--init-file-directory	159	--link-executable-command	161
--inline-alloc	150	--link-object	36, 158
--inline-call-cost	143	--link-shared-lib-command	161
--inline-compound-threshold	142	--linkage	159
--inline-simple	142	--llds-optimise	148
--inline-simple-threshold	142	--llds-optimize	148
--inline-single-use	142	--local-constraint-propagation	145
		--local-thread-engine-base	150

- lookup-switch-req-density 146
- lookup-switch-size 146
- loop-invariants 144
- main 160
- make 33, 117
- make-int 19, 27, 115
- make-interface 19, 27, 115
- make-opt-int 20, 115
- make-optimisation-interface 115
- make-optimization-interface 20, 27, 115
- make-priv-int 19, 27, 115
- make-private-interface 19, 27, 115
- make-short-int 19, 26, 115
- make-short-interface 19, 26, 115
- make-trans-opt 23, 27, 116
- make-transitive-optimisation-interface
..... 116
- make-transitive-optimization-interface
..... 23, 27, 116
- make-xml-doc 117
- make-xml-documentation 117
- max-contexts-per-thread (runtime option)
..... 168
- max-engines (runtime option) 168
- max-error-line-width 125
- max-jump-table-size 165
- maybe-thread-safe 123
- mdb-err (runtime option) 171
- mdb-in (runtime option) 171
- mdb-in-window (mdb option) 48
- mdb-out (runtime option) 171
- mdb-tty (runtime option) 171
- mem-usage-report (runtime option) 171
- memory-profiling 122
- mercury-config-dir 164
- mercury-configuration-directory 164
- mercury-library 35, 158
- mercury-library-directory 35, 157
- mercury-linkage 160
- mercury-standard-library-directory 158
- mercury-stdlib-dir 158
- middle-rec 148
- ml 35, 158
- mld 35, 157
- mlds-optimise 147
- mlds-optimize 147
- mlds-peephole 147
- mode-inference-iteration-limit 124
- mono-path-dir 157
- mono-path-directory 157
- no- 113
- no-allow-non-contiguity-for 135
- no-allow-undefined 160
- no-ansi-c 166
- no-asm-labels 120
- no-c-include-dir 155
- no-c-include-directory 155
- no-cflags 155
- no-color-diagnostics 126
- no-colour-diagnostics 126
- no-common-layout-data 150
- no-config-file 164
- no-const-struct 144
- no-coverage-profiling 122
- no-csharp-flags 157
- no-default-runtime-library-directory .. 159
- no-delay-death 140
- no-demangle 107, 160
- no-detect-libgrades 163
- no-detect-stdlib-grades 163
- no-dump-hlds 166
- no-dump-hlds-pred-name 166
- no-extra-init-command 163
- no-extra-lib-header 164
- no-extra-library-header 164
- no-framework 160
- no-framework-directory 160
- no-fully-strict 124
- no-gcc-global-registers 120
- no-gcc-non-local-gotos 120
- no-generate-standalone-interface 116
- no-generate-trail-ops-inline 145
- no-halt-at-invalid-interface 137
- no-hlds-dump 166
- no-infer-all 123
- no-infer-det 124
- no-infer-determinism 124
- no-inform-inferred 137
- no-inform-inferred-modes 137
- no-inform-inferred-types 137
- no-inhibit-style-warnings 136
- no-inhibit-warnings 136
- no-init-file 159
- no-init-file-directory 159
- no-inlining 142
- no-intermod-directory 162
- no-java-classpath 156
- no-java-flags 156
- no-java-runtime-flags 156
- no-javac-flags 156
- no-ld-flags 158
- no-ld-libflags 159
- no-lib-linkage 163
- no-libgrade 162
- no-libgrade-install-check 163
- no-libgrades-exclude 163
- no-libgrades-exclude-component 163
- no-libgrades-include 163
- no-libgrades-include-component 163
- no-library 158
- no-library-directory 158
- no-library-grade 162
- no-library-install-linkage 163
- no-library-linkage 163
- no-limit-error-contexts 126
- no-line-numbers-around-foreign-code ... 154

- no-link-object..... 158
- no-local-thread-engine-base..... 150
- no-main..... 160
- no-max-error-line-width..... 125
- no-mercury-standard-library-directory
..... 158
- no-mercury-stdlib-dir..... 158
- no-mono-path-dir..... 157
- no-mono-path-directory..... 157
- no-optimise-saved-vars..... 144
- no-optimize-format-calls..... 141
- no-optimize-saved-vars..... 144
- no-options-file..... 164
- no-options-search-directory..... 161
- no-output-compile-error-lines..... 125
- no-pre-link-command..... 154
- no-profiling..... 121
- no-read-opt-files-transitively..... 151
- no-reorder-conj..... 124
- no-reorder-disj..... 124
- no-runtime-flags..... 160
- no-runtime-library-directory..... 159
- no-search-directory..... 162
- no-show-pred-movability..... 137
- no-show-pred-moveability..... 137
- no-split-switch-arms..... 144
- no-strip..... 160
- no-trace-flag..... 139
- no-trace-init-file..... 159
- no-use-readline..... 160
- no-use-search-directories-for-intermod
..... 162
- no-verbose-make..... 125
- no-warn-accumulator-swaps..... 131
- no-warn-ambiguous-pragma..... 129
- no-warn-ambiguous-pragmas..... 129
- no-warn-cannot-table..... 128
- no-warn-det-decls-too-lax..... 128
- no-warn-disj-fills-partial-inst..... 130
- no-warn-duplicate-abstract-instances.. 127
- no-warn-exported-insts-for-private-type
..... 128
- no-warn-inferred-erroneous..... 128
- no-warn-insts-without-matching-type... 127
- no-warn-interface-imports..... 127
- no-warn-known-bad-format-calls..... 130
- no-warn-missing-descendant-modules.... 131
- no-warn-missing-opt-files..... 131
- no-warn-missing-state-var-init..... 130
- no-warn-missing-trans-opt-deps..... 131
- no-warn-moved-trace-goal..... 130
- no-warn-no-auto-parallelisation..... 132
- no-warn-no-auto-parallelization..... 132
- no-warn-no-recursion..... 132
- no-warn-non-contiguous-clauses..... 135
- no-warn-non-contiguous-decls..... 135
- no-warn-non-tail-recursion..... 132
- no-warn-non-term-special-preds..... 128
- no-warn-nonexported-pragma..... 129
- no-warn-nothing-exported..... 126
- no-warn-obsolete..... 130
- no-warn-overlapping-scopes..... 130
- no-warn-redundant-code..... 133
- no-warn-redundant-coerce..... 133
- no-warn-repeated-singleton-variables.. 129
- no-warn-repeated-singleton-vars..... 129
- no-warn-requested-by-code..... 133
- no-warn-requested-by-option..... 134
- no-warn-simple-code..... 129
- no-warn-singleton-variables..... 129
- no-warn-singleton-vars..... 129
- no-warn-state-var-shadowing..... 134
- no-warn-stdlib-shadowing..... 127
- no-warn-stubs..... 128
- no-warn-subtype-ctor-order..... 127
- no-warn-suspected-occurs-check-failure
..... 130
- no-warn-suspected-occurs-failure..... 130
- no-warn-table-with-inline..... 129
- no-warn-target-code..... 155
- no-warn-typecheck-ambiguity-limit..... 128
- no-warn-undefined-options-variables... 131
- no-warn-undefined-options-vars..... 131
- no-warn-unification-cannot-succeed.... 129
- no-warn-unknown-warning-name..... 131
- no-warn-unneeded-final-statevars..... 131
- no-warn-unneeded-final-statevars-lambda
..... 131
- no-warn-unneeded-initial-statevars.... 134
- no-warn-unneeded-initial-statevars-lambda
..... 134
- no-warn-unneeded-mode-specific-clause
..... 133
- no-warn-unneeded-purity-indicator..... 130
- no-warn-unneeded-purity-pragma..... 129
- no-warn-unneeded-purity-pred-decl..... 128
- no-warn-unresolved-polymorphism..... 128
- no-warn-unused-interface-imports..... 127
- no-warn-up-to-date..... 136
- nondetstack-size..... 26
- nondetstack-size (runtime option)..... 169
- nondetstack-size-kwargs (runtime option)
..... 169
- num-ptag-bits..... 123
- num-real-f-regs..... 165
- num-real-f-temps..... 165
- num-real-r-regs..... 165
- num-real-r-temps..... 165
- num-real-temps..... 165
- num-tag-bits..... 123
- opt-level..... 140
- opt-space..... 141
- optimisation-level..... 140
- optimise-constant-propagation..... 142
- optimise-constructor-last-call..... 144
- optimise-dead-procs..... 141

- optimise-delay-slot 149
- optimise-duplicate-calls 142
- optimise-dups 149
- optimise-frames 149
- optimise-fulljumps 149
- optimise-higher-order 143
- optimise-initializations 147
- optimise-jumps 148
- optimise-labels 148
- optimise-peep 148
- optimise-proc-dups 150
- optimise-reassign 149
- optimise-repeat 148
- optimise-saved-vars 144
- optimise-space 141
- optimise-tailcalls 147
- optimise-unused-args 141
- optimization-level 140
- optimize-constant-propagation 142
- optimize-constructor-last-call 144
- optimize-dead-procs 141
- optimize-delay-slot 149
- optimize-duplicate-calls 142
- optimize-dups 149
- optimize-format-calls 141
- optimize-frames 149
- optimize-fulljumps 149
- optimize-higher-order 143
- optimize-initializations 147
- optimize-jumps 148
- optimize-labels 148
- optimize-peep 148
- optimize-proc-dups 150
- optimize-reassign 149
- optimize-repeat 148
- optimize-saved-vars 144
- optimize-space 141
- optimize-tailcalls 147
- optimize-unused-args 141
- options-file 164
- options-search-directory 161
- order-make-by-timestamp 154
- output-c-compiler-type 118
- output-c-include-dir-flags 118
- output-c-include-directory-flags 118
- output-cc 117
- output-cc-type 118
- output-cflags 118
- output-class-dir 118
- output-class-directory 118
- output-compile-error-lines 125
- output-csharp-compiler 118
- output-csharp-compiler-type 118
- output-file 158
- output-grade-defines 117
- output-grade-string 117
- output-java-class-dir 118
- output-java-class-directory 118
- output-libgrades 117
- output-library-install-grades 117
- output-library-link-flags 118
- output-link-command 118
- output-opt-opts 118
- output-opt-opts-upto 119
- output-optimization-options 118
- output-optimization-options-upto 119
- output-shared-lib-link-command 118
- output-stdlib-grades 117
- output-stdlib-modules 117
- output-target-arch 117
- parallel 105, 123
- pessimize-tailcalls 149
- pneystack-size (runtime option) 171
- pneystack-size-kwargs (runtime option) 171
- pre-link-command 154
- pretty-print 116
- procs-per-c-function 150
- procs-per-C-function 150
- profile-for-feedback 122
- profile-for-implicit-parallelism 122
- profile-optimised 140
- profile-optimized 140
- profiling 121
- program-in-window (mdb option) 48
- quoted-cflag 156
- quoted-csharp-flag 157
- quoted-java-flag 156
- quoted-java-runtime-flag 156
- quoted-javac-flag 156
- quoted-ld-flag 159
- quoted-ld-libflag 159
- read-opt-files-transitively 151
- rebuild 117
- reorder-conj 124
- reorder-disj 124
- restricted-command-line 165
- reverse-error-order 125
- runtime-flags 160
- runtime-library-directory 159
- search-directory 162
- search-lib-files-dir 36, 157
- search-library-files-directory 157
- shlib-linker-install-name-path 161
- show-all-type-repns 139
- show-all-type-representations 139
- show-definition-extents 138
- show-definition-line-counts 138
- show-definitions 137
- show-defn-extents 138
- show-defn-line-counts 138
- show-defns 137
- show-dependency-graph 139
- show-imports-graph 139
- show-local-call-tree 138
- show-local-type-repns 139

- show-local-type-representations..... 139
- show-make-times 154
- show-pred-movability..... 137
- show-pred-moveability 137
- sign-assembly..... 161
- simple-neg..... 148
- single-prec-float..... 123
- single-precision-float..... 123
- small-detstack-size (runtime option) ... 170
- small-detstack-size-kwds (runtime option) 170
- small-nondetstack-size (runtime option) 170
- small-nondetstack-size-kwds (runtime option) 170
- smart-indexing..... 145
- solutions-heap-size (runtime option) ... 170
- solutions-heap-size-kwds (runtime option) 170
- split-switch-arms..... 144
- stack-segments..... 123
- stack-trace-higher-order..... 140
- static-ground-terms..... 146
- statistics..... 125
- strict-sequential..... 124
- string-binary-switch-size..... 146
- string-hash-switch-size..... 146
- string-switch-size..... 146
- string-trie-size..... 146
- string-trie-switch-size..... 146
- strip..... 160
- strip-executable-command..... 161
- strip-executable-shared-flags..... 161
- strip-executable-static-flags..... 161
- system-env-type..... 164
- tabling-statistics (runtime option) 171
- tag-switch-size..... 146
- target..... 119
- target-code-only..... 23, 116
- target-debug..... 155
- target-debug-grade..... 121
- target-env-type..... 165
- term-err-limit..... 153
- term-norm..... 153
- term-path-limit..... 153
- term-single-arg..... 152
- termination-error-limit..... 153
- termination-norm..... 153
- termination-path-limit..... 153
- termination-single-argument-analysis.. 152
- thread-pinning (runtime option)..... 169
- threadscope..... 105
- trace..... 139
- trace-count (runtime option)..... 172
- trace-count-if-exec (runtime option) ... 172
- trace-count-output-file (runtime option) 172
- trace-count-summary-file (runtime option) 172
- trace-count-summary-max (runtime option) 172
- trace-flag..... 139
- trace-init-file..... 159
- trace-optimised..... 139
- trace-optimized..... 139
- track-flags..... 154
- track-options..... 154
- trail-segment-size (runtime option).... 170
- trail-segment-size-kwds (runtime option) 170
- trans-intermod-opt..... 37, 151
- transitive-intermodule-optimisation ... 151
- transitive-intermodule-optimization ... 151
- try-switch-size..... 147
- tty (mdb option)..... 48
- type-inference-iteration-limit..... 124
- type-specialisation..... 143
- type-specialization..... 143
- typecheck-ambiguity-error-limit..... 126
- typecheck-ambiguity-warn-limit..... 126
- typecheck-only..... 116
- unneded-code..... 141
- unneded-code-copy-limit..... 141
- use-atomic-cells..... 146
- use-grade-subdirs..... 162
- use-local-vars..... 149
- use-macro-for-redo-fail..... 150
- use-opt-files..... 151
- use-readline..... 160
- use-search-directories-for-intermod ... 162
- use-subdirs..... 26, 162
- use-trail..... 123
- use-trans-opt-files..... 151
- user-guided-type-specialisation..... 143
- user-guided-type-specialization..... 143
- verb-check-term..... 152
- verb-chk-term..... 152
- verbose..... 124
- verbose-check-termination..... 152
- verbose-commands..... 125
- verbose-error-messages..... 125
- verbose-errors..... 125
- verbose-make..... 125
- version..... 114
- very-verbose..... 125
- warn-accumulator-swaps..... 131
- warn-all-format-string-errors..... 136
- warn-ambiguous-pragma..... 129
- warn-ambiguous-pragmas..... 129
- warn-can-fail-function..... 133
- warn-cannot-table..... 128
- warn-dead-predicates..... 132
- warn-dead-preds..... 132
- warn-dead-procedures..... 133
- warn-dead-procs..... 133

--warn-det-decls-too-lax	128	--warn-suspicious-foreign-code	134
--warn-disj-fills-partial-inst	130	--warn-suspicious-foreign-procs	134
--warn-duplicate-abstract-instances	127	--warn-suspicious-recursion	130
--warn-duplicate-calls	133	--warn-table-with-inline	129
--warn-exported-insts-for-private-type ..	128	--warn-target-code	155
--warn-implicit-stream-calls	134	--warn-too-private-instances	127
--warn-include-and-non-include	132	--warn-typecheck-ambiguity-limit	128
--warn-incomplete-switch	133	--warn-undefined-options-variables	131
--warn-incomplete-switch-threshold	133	--warn-undefined-options-vars	131
--warn-inconsistent-pred-order	135	--warn-unification-cannot-succeed	129
--warn-inconsistent-pred-order-clauses ..	135	--warn-unknown-format-calls	134
--warn-inconsistent-pred-order-foreign- procs	135	--warn-unknown-warning-name	131
--warn-inferred-erroneous	128	--warn-unneeded-final-statevars	131
--warn-insts-with-functors-without-type	127	--warn-unneeded-final-statevars-lambda ..	131
--warn-insts-without-matching-type	127	--warn-unneeded-initial-statevars	134
--warn-interface-imports	127	--warn-unneeded-initial-statevars-lambda	134
--warn-interface-imports-in-parents	127	--warn-unneeded-mode-specific-clause	133
--warn-ite-instead-of-switch	133	--warn-unneeded-purity-indicator	130
--warn-known-bad-format-calls	130	--warn-unneeded-purity-pragma	129
--warn-missing-descendant-modules	131	--warn-unneeded-purity-pred-decl	128
--warn-missing-opt-files	131	--warn-unresolved-polymorphism	128
--warn-missing-state-var-init	130	--warn-unsorted-import-block	135
--warn-missing-trans-opt-deps	131	--warn-unsorted-import-blocks	135
--warn-missing-trans-opt-files	131	--warn-unused-args	130
--warn-moved-trace-goal	130	--warn-unused-imports	127
--warn-no-auto-parallelisation	132	--warn-unused-interface-imports	127
--warn-no-auto-parallelization	132	--warn-unused-types	127
--warn-no-recursion	132	--warn-up-to-date	136
--warn-non-contiguous-clauses	135	--window (mdb option)	48
--warn-non-contiguous-decls	135	--window-command (mdb option)	48
--warn-non-contiguous-foreign-procs	135	-?	114
--warn-non-stratification	128	-c	24, 25, 116
--warn-non-tail-recursion	132	-C	23, 116
--warn-non-term-special-preds	128	-C (runtime option)	168
--warn-nonexported-pragma	129	-d	166
--warn-nothing-exported	126	-d-	166
--warn-obsolete	130	-D (runtime option)	168
--warn-obvious-non-tail-recursion	132	-e	116
--warn-overlapping-scopes	130	-e (runtime option)	171
--warn-potentially-ambiguous-pragma	129	-E	125
--warn-potentially-ambiguous-pragmas	129	-f	114
--warn-redundant-code	133	-F	160
--warn-redundant-coerce	133	-F-	160
--warn-repeated-singleton-variables	129	-h	114
--warn-repeated-singleton-vars	129	-H	121
--warn-requested-by-code	133	-i	115
--warn-requested-by-option	134	-i (runtime option)	171
--warn-simple-code	129	-I	162
--warn-singleton-variables	129	-I-	162
--warn-singleton-vars	129	-j	154
--warn-state-var-shadowing	134	-k	154
--warn-stdlib-shadowing	127	-l	158
--warn-stubs	128	-l-	158
--warn-subtype-ctor-order	127	-L	158
--warn-suspected-occurs-check-failure ...	130	-L-	158
--warn-suspected-occurs-failure	130	-m	117
		-m (runtime option)	171

- M..... 115
 - n..... 153
 - o..... 25, 158
 - o (runtime option)..... 171
 - O..... 140
 - p..... 121
 - p (runtime option)..... 168
 - P..... 121
 - P..... 116
 - P (runtime option)..... 168
 - r..... 117
 - R..... 159
 - R-..... 159
 - s..... 119
 - S..... 125
 - t..... 116
 - T (runtime option)..... 169
 - v..... 124
 - V..... 125
 - w..... 136
 - w-..... 136
 - x..... 117
-
- .debug (grade modifier)..... 121
 - .decldebug (grade modifier)..... 121
 - .gc (grade modifier)..... 122
 - .mdbrc..... 47
 - .memprof (grade modifier)..... 122
 - .par (grade modifier)..... 123
 - .prof (grade modifier)..... 121
 - .profdeep (grade modifier)..... 122
 - .stseg (grade modifier)..... 123
 - .tr (grade modifier)..... 123
- ## A
- alias (mdb command)..... 79
 - all_class_decls (mdb command)..... 89
 - all_procedures (mdb command)..... 89
 - all_regs (mdb command)..... 86
 - all_type_ctors (mdb command)..... 88
 - Allocation profiling..... 105, 106, 110, 122
 - ambiguity (mdb command)..... 89
 - AR..... 37
 - ARFLAGS..... 37
 - asm_fast (compilation grade)..... 119
 - Automatic parallelism..... 122
- ## B
- break (mdb command)..... 67
 - break points..... 48
 - break_print (mdb command)..... 72
 - Breakpoints..... 67
 - browse (mdb command)..... 62
 - Building profiled applications..... 105
- Building programs..... 28
- ## C
- c2init..... 25
 - C compiler..... 155
 - C compiler options..... 155
 - C compilers..... 178
 - C# compiler..... 157
 - C# compiler options..... 157
 - C# libraries..... 39
 - C2INITARGS..... 32
 - C2INITFLAGS..... 32
 - call (trace event)..... 42
 - Call graph profile..... 107
 - cc_query (mdb command)..... 54
 - CFLAGS..... 31
 - CIL interpreter..... 157
 - CLANG_FLAGS..... 34
 - class_decl (mdb command)..... 88
 - classpath..... 156
 - Clauses, procedures without..... 124
 - clear_histogram (mdb command)..... 83
 - Color schemes..... 175
 - Compilation speed..... 140
 - cond (trace event)..... 43
 - condition (mdb command)..... 71
 - consumer (mdb command)..... 86
 - context (mdb command)..... 75
 - continue (mdb command)..... 58
 - control-C..... 49
 - Controlling trace goals..... 139
 - Coverage profiling..... 122
 - Creating profiles..... 106
 - Cross-module optimization..... 140
 - csharp (compilation grade)..... 119
 - current (mdb command)..... 64
 - Cut stack size..... 170, 171
 - cut_stack (mdb command)..... 86
- ## D
- Data representation..... 123
 - debug_vars (mdb command)..... 87
 - debugger break points..... 48
 - debugger interrupt..... 49
 - debugger print level..... 49
 - debugger procedure specification..... 51
 - debugger trace events..... 42
 - debugger trace level..... 45
 - Debugging..... 39, 121
 - Debugging grade options..... 121
 - Debugging Threads..... 171
 - Deep profiling..... 105, 106, 111, 122
 - Deep profiling grade options..... 122
 - deep tracing..... 45
 - delete (mdb command)..... 73
 - Dependencies..... 115

depth (mdb command) 78
 Determinism inference 124
 Developer grade options 123
 Diagnostic output 173
 Diagnostics options 125
 dice (mdb command) 83
 diff (mdb command) 66
 Directories 126, 156, 162, 167
 Directories for libraries 157, 158, 160
 disable (mdb command) 73
 disj (trace event) 43
 document (mdb command) 80
 document_category (mdb command) 80
 down (mdb command) 64
 dump (mdb command) 66
 Dumping out internal compiler data structures
 166

E

echo (mdb command) 75
 Efficiency 105
 else (trace event) 43
 Emacs 41
 enable (mdb command) 73
 Enabling the use of color 177
 Environment variables 167
 Environment variables affecting the Mercury
 compiler 167
 Environment variables affecting the Mercury
 debugger 167
 Environment variables affecting the Mercury
 runtime system 168
 exception (mdb command) 57
 exit (trace event) 42
 EXTRA_C2INITARGS 32
 EXTRA_C2INITFLAGS 32
 EXTRA_CFLAGS 31
 EXTRA_GRADEFLAGS 31
 EXTRA_JAVACFLAGS 31
 EXTRA_LD_LIBFLAGS 31
 EXTRA_LDFLAGS 31
 EXTRA_LIB_DIRS 32, 38
 EXTRA_LIBRARIES 32, 38
 EXTRA_MC_MAKE_FLAGS 34
 EXTRA_MCFLAGS 31
 EXTRA_MGNUCLFLAGS 31
 EXTRA_MLFLAGS 31
 EXTRA_MLLIBS 32
 EXTRA_MLOBS 32

F

fail (trace event) 42
 fail_trace_counts (mdb command) 77
 File extensions 26
 File names 26, 126, 162
 file names 75

finish (mdb command) 56
 flag (mdb command) 85
 Flat profile 107
 format (mdb command) 78
 format_param (mdb command) 78
 forward (mdb command) 58

G

Garbage collection, profiling 108
 GCC_FLAGS 34
 gen_stack (mdb command) 86
 General options for compiling target language code
 155
 General options for linking 157
 Generator stack size 170
 GNU C 178
 GNU Emacs 41
 goal path 44
 goal_path (mdb command) 75
 goto (mdb command) 56
 Grade options 119
 GRADEFLAGS 31, 32
 Grades 162
 Grades and grade components 119

H

Heap overflow 26
 Heap profiling 105, 106, 110, 122
 Heap size 26
 held variables (in mdb) 50
 held_vars (mdb command) 60
 help (mdb command) 80
 Help option 114
 Help options 114
 Higher-order specialization 143
 histogram_all (mdb command) 82
 histogram_exp (mdb command) 82
 hlc (compilation grade) 119
 hold (mdb command) 66

I

ignore (mdb command) 72
 Inference 123
 Inference of determinisms 124
 Inference of modes 124
 Inference of types 123, 124
 INSTALL 30, 32, 38
 INSTALL_MKDIR 30, 32, 38
 INSTALL_PREFIX 32, 38
 Intermodule optimization 140, 151
 interrupt, in debugger 49
 io_query (mdb command) 54

J

jar files	39
java (compilation grade)	119
Java compiler	156
Java compiler flags	156
Java interpreter	156
Java libraries	39
Java runtime options	156
JAVACFLAGS	31

L

LD_BIND_NOW	113
LD_LIBFLAGS	31
LDFLAGS	31
level (mdb command)	64
LIB_LINKAGES	30, 33
LIBGRADES	30, 32, 38
Libraries	34
Libraries, linking with	35, 36, 38, 158
line numbers	75
lines (mdb command)	78
LINKAGE	31
list (mdb command)	67
list_cmd (mdb command)	77
list_context_lines (mdb command)	76
list_path (mdb command)	77
LLDS backend grade options	120

M

Mac OS X, Using frameworks	160
MAIN_TARGET	31
make --- see Mmake	28
max_io_actions (mdb command)	78
maxdepth (mdb command)	58
MC	31
MC_BUILD_FILES	34
MC_MAKE_FLAGS	34
MCFLAGS	31
mdb	39
Mdb debugging grade options	121
mdbre	47
mdprof	105, 106, 111
Measuring performance	105
Memory attribution	110
Memory profiling	105, 106, 110, 122
Memory retention	110
'Mercury' subdirectory	126, 162
Mercury.options	33
MERCURY_COLOR_SCHEME	167
MERCURY_COMPILER	167
MERCURY_DEBUGGER_INIT	47, 167
MERCURY_DEFAULT_GRADE	167, 179
MERCURY_ENABLE_COLOR	167
MERCURY_LINKAGE	31
MERCURY_MAIN_MODULES	34
MERCURY_MKINIT	167

MERCURY_OPTIONS	25, 168
MERCURY_STDLIB_DIR	167
mgnuc	25
MGNUCC	31
MGNUCCFLAGS	31
Microsoft Management Console	1
mindepth (mdb command)	58
ml	25
ML	31, 37
MLDS backend grade options	121
MLFLAGS	31, 37
MLLIBS	32, 37
MLOBJS	32, 37
MLPICOBJS	37
mm_stacks (mdb command)	86
mmake	28
Mmake variables	30
mmc	1
mmc_options (mdb command)	74
Mode inference	124
modules (mdb command)	73
mprof	105, 106, 107, 110, 113
Mprof profiling grade options	121
MSVC_FLAGS	34

N

neg_enter (trace event)	43
neg_fail (trace event)	43
neg_success (trace event)	43
next (mdb command)	56
No clauses, procedures with	124
NO_COLOR	167
Non-transitive intermodule optimization	151
nondet_stack (mdb command)	86
none (compilation grade)	119
Now-unused former options kept for compatibility	166

O

Object files, linking with	158
Operation selection options for developers only	165
Optimization	105
Optimization levels	140
Optimization options	140
Optimizations during code generation	145
Optimizations specific to high level code	147
Optimizations specific to low level code	147
Optional feature grade options	122
Options about halting for warnings	136
Options controlling searches for files	161
Options controlling the library installation process	162
Options files	33
Options for compiling C code	155
Options for compiling C# code	157

Options for compiling Java code 156
 Options for controlling `mmc -make` 154
 Options for developers only 165
 Options for linking 157
 Options for linking C or C# code 158
 Options for linking just C code 158
 Options for linking just C# code 161
 Options for linking just Java code 161
 Options for modifying the command line 114
 Options for target language compilation 155
 Options intended for internal use by the compiler
 only 166
 Options specifying properties of the environment
 164
 Options specifying the intended semantics 124
 Options that ask for informational files 137
 Options that ask for modified output 153
 Options that control color in diagnostics 126
 Options that control diagnostics 125
 Options that control inference 123
 Options that control warnings 136
 Options that give the compiler its overall task
 114
 Options that record autoconfigured parameters
 165
 Options that request information 137
 Ordering diagnostics 174
 Other program analyses 153
 Overall control of optimizations 140

P

Parallel execution profiling 113
 Parallel performance 105
 Parallel runtime profiling 105
`pass_trace_counts` (mdb command) 78
`path` 44
`paths` in terms 50
 PIC (position independent code) 36
 Pneg stack size 171
`pneg_stack` (mdb command) 86
`pop_list_dir` (mdb command) 77
 Position independent code 36
 Preparing code for mdb debugging 139
 Preparing code for mdprof profiling 140
`print` (mdb command) 60
`print level` 49
`print_optionals` (mdb command) 87
`printlevel` (mdb command) 74
 procedure specification (in mdb) 51
`procedures` (mdb command) 73
 Procedures with no clauses 124
 Profiler feedback 122
 Profiling 105, 106, 121, 122
 Profiling and shared libraries 113
 Profiling grade options 121
 Profiling interrupts 107
 Profiling memory allocation 105, 110

Program analyses 152
`push_list_dir` (mdb command) 77

Q

`query` (mdb command) 54
`quit` (mdb command) 82

R

`RANLIB` 37
`RANLIBFLAGS` 37
 Recompiling 28
`redo` (trace event) 42
`reg` (compilation grade) 119
`register` (mdb command) 73
`retry` (mdb command) 59
`return` (mdb command) 57

S

`save` (mdb command) 82
`scope` (mdb command) 75
`scroll` (mdb command) 74
 Search path 162, 167
 Search path for libraries 157, 158, 160
 shallow tracing 45
 Shared libraries 36
 Shared libraries and profiling 113
 Shared objects 36
`show-ghc-events` 113
`size` (mdb command) 78
`source` (mdb command) 82
 Source-to-source optimizations 141
 Specialization of higher-order calls 143
 spy points 48
`stack` (mdb command) 63
 Stack overflow 26
 Stack size 26
`stack_default_limit` (mdb command) 75
`stack_regs` (mdb command) 86
 Static libraries 36
`stats` (mdb command) 87
`step` (mdb command) 55
 strict debugger commands 49
 Stubs 124
 Subdirectories 126, 162
`subgoal` (mdb command) 86
`switch` (trace event) 43

T

`table` (mdb command) 87
`table_io` (mdb command) 73
 Tags 123
 Target options 119
 term navigation 50

The termination analyser based on linear
 inequality constraints 152
 then (trace event) 43
 Threads, Debugging 171
 threadscope 105, 113
 Threadscope profiling 105
 ThreadScope profiling 113
 Time profiling 105, 106, 107, 121
 trace events 42
 trace level 45
 Tracing 39, 42
 track (mdb command) 59
 Trail size 170
 trail_details (mdb command) 90
 Transitive intermodule optimization 151
 trust (mdb command) 81
 trusted (mdb command) 81
 Type inference 123, 124
 type_ctor (mdb command) 88

U

unalias (mdb command) 79
 unhide_events (mdb command) 87
 untrust (mdb command) 82
 up (mdb command) 63
 user (mdb command) 57
 user defined event attributes (in mdb) 51
 user defined events (in mdb) 50
 user_event_context (mdb command) 76
 Using a different C compiler 178

V

var_details (mdb command) 85
 Variables, environment 167
 Variables, Mmake 30
 vars (mdb command) 60
 Verbose diagnostics 174
 Verbosity 174
 Verbosity options 124
 view (mdb command) 65

W

Warning options 126
 Warnings about missing contiguity 135
 Warnings about missing files 131
 Warnings about missing order 135
 Warnings about possible goal incorrectness 129
 Warnings about possible incorrectness 126
 Warnings about possible inst incorrectness 127
 Warnings about possible module incorrectness
 126
 Warnings about possible performance issues... 131
 Warnings about possible pragma incorrectness
 129
 Warnings about possible predicate incorrectness
 128
 Warnings about programming style 132
 Warnings about style issues with goals 133
 Warnings about style issues with modules 132
 Warnings about style issues with predicates... 132
 web_browser_cmd (mdb command) 78
 width (mdb command) 78