11 11 TEX LATEX

The Glasgow Haskell Compiler User's Guide, Version 4.04

The GHC Team, Email: glasgow-haskell-{bugs,users}-request@haskell.org

Contents

1 Introduction to GHC **6**

1.1 The (batch) compilation system components . . . 6
1.2 What really happens when I "compile" a Haskell program? . . . 7
1.3 Meta-information: Web sites, mailing lists, etc. . . . 7
1.4 Release notes for version 4.04—7/99 . . . 8

2 Using GHC **10**

2.1 Overall command-line structure . . . 10
2.2 Meaningful file suffixes . . . 10
2.3 Help and verbosity options . . . 11
2.4 Running the right phases in the right order . . . 11
2.5 Re-directing the compilation output(s) . . . 11
2.5.1 Saving GHC's standard error output . . . 12
2.5.2 Redirecting temporary files . . . 12
2.6 Warnings and sanity-checking . . . 12
2.7 Separate compilation . . . 14
2.7.1 Interface files . . . 14
2.7.2 Finding interface files . . . 15
2.7.3 Other options related to interface files . . . 15
2.7.4 The recompilation checker . . . 16
2.7.5 Using `make` . . . 16
2.7.6 How to compile mutually recursive modules . . . 18
2.8 Optimisation (code improvement) . . . 19
2.8.1 `-O*`: convenient "packages" of optimisation flags. . . . 19
2.8.2 `-f*`: platform-independent flags . . . 20
2.8.3 `-m*`: platform-specific flags . . . 22
2.8.4 Code improvement by the C compiler. . . . 22
2.9 Options related to a particular phase . . . 23
2.9.1 The C pre-processor . . . 23
2.9.2 Options affecting the C compiler (if applicable) . . . 24
2.9.3 Linking and consistency-checking . . . 24
2.10 Using Concurrent Haskell . . . 25
2.11 Using Parallel Haskell . . . 25

2.13.3 Dumping out compiler intermediate structures . . . 33
2.13.4 Checking for consistency . . . 36
2.13.5 How to read Core syntax (from some `-ddump-*` flags) . . . 36
2.13.6 Command line options in source files . . . 38

3 Profiling **38**

3.1 How to profile a Haskell program . . . 38
3.2 Compiling programs for profiling . . . 39
3.3 How to control your profiled program at runtime . . . 40
3.4 What's in a profiling report? . . . 42
3.5 Producing graphical heap profiles . . . 43
3.5.1 `hp2ps`–heap profile to PostScript . . . 43
3.5.2 `stat2resid`—residency info from GC stats . . . 44
3.6 Using "ticky-ticky" profiling (for implementors) . . . 45

4 Advice on: sooner, faster, smaller, stingier **47**

4.1 Sooner: producing a program more quickly . . . 47
4.2 Faster: producing a program that runs quicker . . . 48
4.3 Smaller: producing a program that is smaller . . . 51
4.4 Stingier: producing a program that gobbles less heap space . . . 52

5 GHC Language Features **52**

5.1 Unboxed types . . . 53
5.2 Primitive state-transformer monad . . . 53
5.3 Primitive arrays, mutable and otherwise . . . 53
5.4 Calling C directly from Haskell . . . 54
5.4.1 `_ccall_` and `_casm_`: an introduction . . . 54
5.4.2 Literal-literals . . . 55
5.4.3 Using function headers . . . 56
5.4.4 Subverting automatic unboxing with "stable pointers" . . . 56
5.4.5 Foreign objects: pointing outside the Haskell heap . . . 57
5.4.6 Avoiding monads . . . 58
5.4.7 C-calling "gotchas" checklist . . . 59
5.5 Multi-parameter type classes . . . 61
5.5.1 Types . . . 61

5.5.2 Class declarations . . . 62
5.5.3 Instance declarations . . . 63
5.6 Explicit universal quantification . . . 65
5.6.1 Universally-quantified data type fields . . . 66
5.6.2 Construction . . . 66
5.6.3 Pattern matching . . . 67
5.6.4 The partial-application restriction . . . 67
5.6.5 Type signatures . . . 68
5.7 Existentially quantified data constructors . . . 69
5.7.1 Why existential? . . . 70
5.7.2 Type classes . . . 70
5.7.3 Restrictions . . . 70
5.8 Assertions . . . 71
5.9 Scoped Type Variables . . . 72
5.9.1 Scope and implicit quantification . . . 72
5.9.2 Polymorphism . . . 73
5.9.3 Result type signatures . . . 74
5.9.4 Pattern signatures on other constructs . . . 74
5.9.5 Existentials . . . 75
5.10 Pragmas . . . 76
5.10.1 INLINE pragma . . . 76
5.10.2 NOINLINE pragma . . . 76
5.10.3 SPECIALIZE pragma . . . 76
5.10.4 SPECIALIZE instance pragma . . . 77
5.10.5 LINE pragma . . . 77
5.10.6 RULES pragma . . . 78
5.11 Rewrite rules . . . 78
5.11.1 Syntax . . . 78
5.11.2 Semantics . . . 79
5.11.3 Controlling what's going on . . . 80
5.12 Concurrent and Parallel Haskell . . . 80
5.12.1 Language features specific to Concurrent Haskell . . . 80
5.12.2 Features specific to Parallel Haskell . . . 83
5.13 Haskell 98 vs. Glasgow Haskell: language non-compliance . . . 84

5.13.1 Expressions and patterns . . . 84
5.13.2 Declarations and bindings . . . 84
5.13.3 Module system and interface files . . . 84
5.13.4 Numbers, basic types, and built-in classes . . . 84
5.13.5 In Prelude support . . . 85

6 The GHC prelude and libraries 85

6.1 Prelude extensions . . . 85
6.2 GHC/Hugs Extension Libraries . . . 86
6.3 GHC-only Extension Libraries . . . 86
6.3.1 MutableArray . . . 86
6.3.2 The ByteArray interface . . . 88
6.3.3 The `CCall` module . . . 89
6.3.4 The `GlaExts` interface . . . 89
6.4 The module `PrelGHC`: really primitive stuff . . . 89
6.4.1 Unboxed Tuples . . . 90
6.4.2 Character and numeric types . . . 91
6.4.3 Comparison operations . . . 91
6.4.4 Primitive-character operations . . . 91
6.4.5 Primitive-`Int` operations . . . 91
6.4.6 Primitive-`Double` and `Float` operations . . . 91
6.4.7 Operations on/for `Integers` (interface to GMP) . . . 92
6.4.8 Words and addresses . . . 93
6.4.9 Arrays . . . 94
6.4.10 Reading and writing . . . 94
6.4.11 The state type . . . 95
6.4.12 State of the world . . . 96
6.4.13 Mutable arrays . . . 96
6.4.14 Stable pointers . . . 97
6.4.15 Foreign objects . . . 97
6.4.16 Synchronizing variables (M-vars) . . . 98
6.5 The Posix library . . . 98
6.5.1 Posix data types . . . 98
6.5.2 Posix Process Primitives . . . 102

6.5.3 Posix Process Environment . . . 106
6.5.4 Posix operations on files and directories . . . 109
6.5.5 Posix Input and Output Primitives . . . 114
6.5.6 Posix, Device- and Class-Specific Functions . . . 115
6.5.7 Posix System Databases . . . 116
6.5.8 POSIX Errors . . . 117
6.6 Miscellaneous libraries . . . 118
6.6.1 The `Bag` type . . . 118
6.6.2 The `FiniteMap` type . . . 119
6.6.3 The `ListSetOps` type . . . 121
6.6.4 The `Maybes` type . . . 121
6.6.5 The `Memo` library . . . 122
6.6.6 The `PackedString` type . . . 122
6.6.7 The `Pretty` type . . . 124
6.6.8 The `Set` type . . . 124
6.6.9 The `BitSet` interface . . . 125
6.6.10 The `Util` type . . . 125
6.7 Interfaces to C libraries . . . 127
6.7.1 The `Readline` interface . . . 127
6.7.2 The `Regex` and `MatchPS` interfaces . . . 127
6.7.3 The `RegexString` interface . . . 130
6.7.4 Network-interface toolkit—`Socket` and `SocketPrim` . . . 130

7 What to do when something goes wrong 131
7.1 When the compiler "does the wrong thing" . . . 132
7.2 When your program "does the wrong thing" . . . 133
7.3 How to report a bug in the GHC system . . . 134
7.4 Hard-core debugging of GHC-compiled programs . . . 134

8 Other Haskell utility programs 135
8.1 Makefile dependencies in Haskell: using `mkdependHS` . . . 135
8.2 Emacs 'TAGS' for Haskell: `hstags` . . . 137
8.3 "Yacc for Haskell": `happy` . . . 138
8.4 Pretty-printing Haskell: `pphs` . . . 138

9 Building and using Win32 DLLs 138

9.1 Linking with DLLs 138

9.2 Not linking with DLLs 139

9.3 Creating a DLL 139

1 Introduction to GHC

This is a guide to using the Glasgow Haskell compilation (GHC) system. It is a batch compiler for the Haskell 1.4 language, with support for various Glasgow-only extensions.

Many people will use GHC very simply: compile some modules—`ghc -c -O Foo.hs Bar.hs`; and link them— `ghc -o wiggle -O Foo.o Bar.o`.

But if you need to do something more complicated, GHC can do that, too:

```
ghc -c -O -fno-foldr-build -dcore-lint -fvia-C -ddump-simpl Foo.lhs
```

Stay tuned—all will be revealed!

In this document, we assume that GHC has been installed at your site as `ghc`. The rest of this section provide some tutorial information on batch-style compilation; if you're familiar with these concepts already, then feel free to skip to the next section.

1.1 The (batch) compilation system components

The Glorious Haskell Compilation System, as with most UNIX (batch) compilation systems, has several interacting parts:

1. A *driver* `ghc`—which you usually think of as "the compiler"—is a program that merely invokes/glues-together the other pieces of the system (listed below), passing the right options to each, slurping in the right libraries, etc.

2. A *literate pre-processor* `unlit` that extracts Haskell code from a literate script; used if you believe in that sort of thing.

3. The *Haskellised C pre-processor* `hscpp`, only needed by people requiring conditional compilation, probably for large systems. The "Haskellised" part just means that `#line` directives in the output have been converted into proper Haskell {`-# LINE ... -`} pragmas.

   You must give an explicit `-cpp` option for the C pre-processor to be invoked.

4. The *Haskell compiler* `hsc`, which—in normal use—takes its input from the C pre-processor and produces assembly-language output (sometimes: ANSI C output).

5. The *ANSI C Haskell high-level assembler :-)*

   compiles `hsc`'s C output into assembly language for a particular target architecture. In fact, the only C compiler we currently support is `gcc`, because we make use of certain extensions to the C language only supported by gcc. Version 2.x is a must; we recommend version 2.7.2.1 for stability (we've heard both good and bad reports of later versions).

6. The *assembler*—a standard UNIX one, probably `as`.
7. The *linker*—a standard UNIX one, probably `ld`.
8. A *runtime system*, including (most notably) a storage manager; the linker links in the code for this.
9. The *Haskell standard prelude*, a large library of standard functions, is linked in as well.
10. Parts of other *installed libraries* that you have at your site may be linked in also.

1.2 What really happens when I "compile" a Haskell program?

You invoke the Glasgow Haskell compilation system through the driver program `ghc`. For example, if you had typed a literate "Hello, world!" program into `hello.lhs`, and you then invoked:

```
ghc hello.lhs
```

the following would happen:

1. The file `hello.lhs` is run through the literate-program code extractor `unlit`, feeding its output to
2. The Haskell compiler proper `hsc`, which produces input for
3. The assembler (or that ubiquitous "high-level assembler," a C compiler), which produces an object file and passes it to
4. The linker, which links your code with the appropriate libraries (including the standard prelude), producing an executable program in the default output file named `a.out`.

You have considerable control over the compilation process. You feed command-line arguments (call them "options," for short) to the driver, `ghc`; the "types" of the input files (as encoded in their names' suffixes) also matter.

Here's hoping this is enough background so that you can read the rest of this guide!

1.3 Meta-information: Web sites, mailing lists, etc.

On the World-Wide Web, there are several URLs of likely interest:

- *Haskell home page* `<http://haskell.org/>`
- *GHC home page* `<http://research.microsoft.com/users/t-simonm/ghc/>`
- *Glasgow FP group page* `<http://www.dcs.gla.ac.uk/fp/>`
- *comp.lang.functional FAQ* `<http://www.cs.nott.ac.uk/Department/Staff/mpj/faq.html>`

We run two mailing lists about Glasgow Haskell. We encourage you to join, as you feel is appropriate.

glasgow-haskell-users:

This list is for GHC users to chat among themselves. Subscribe by sending mail to *majordomo@haskell.org*, with a message body (not header) like this:

```
subscribe glasgow-haskell-users MyName <m.y.self@@bigbucks.com>
```

(The last bit is your all-important e-mail address, of course.)

To communicate with your fellow users, send mail to *glasgow-haskell-users@haskell.org* <mailto:glasgow-haskell-users@haskell.org>.

To contact the list administrator, send mail to *glasgow-haskell-users-request@haskell.org*. An archive of the list is available on the Web: *glasgow-haskell-users mailing list archive* <http://www.dcs.gla.ac.uk/mail-www/glasgow-haskell-users>.

glasgow-haskell-bugs:

Send bug reports for GHC to this address! The sad and lonely people who subscribe to this list will muse upon what's wrong and what you might do about it.

Subscribe via *majordomo@haskell.org* with:

```
subscribe glasgow-haskell-bugs My Name <m.y.self@@hackers.r.us>
```

Again, you may contact the list administrator at *glasgow-haskell-bugs-request@haskell.org*. And, yes, an archive of the list is available on the Web at: : *glasgow-haskell-bugs mailing list archive* <http://www.dcs.gla.ac.uk/mail-www/glasgow-haskell-bugs>

There is also the general Haskell mailing list. Subscribe by sending email to *majordomo@dcs.gla.ac.uk*, with the usual message body:

```
subscribe haskell My Name <m.y.self@@fp.rules.ok.org>
```

Some Haskell-related discussion takes place in the Usenet newsgroup `comp.lang.functional`. (But note: news is basically dead at Glasgow. That's one reason Glaswegians aren't too active in c.f.l.)

The main anonymous-FTP site for Glasgow Haskell is *ftp://ftp.dcs.gla.ac.uk/pub/haskell/glasgow*. "Important" bits are mirrored at other Haskell archive sites (and we have their stuff, too).

1.4 Release notes for version 4.04—7/99

- `Weak` library changes:
  - Finalise is now spelt finalize in all function names.
  - `finalize` now runs the finalizer immediately, and waits for it to complete before returning.
  - `mkWeak` now takes a `Maybe (IO ())` for the finalizer, and `mkWeakNoFinalizer` is removed.
  - A weak pointer whose key is itself a weak pointer will now do the right thing.
- Changed representation of `Integer` type to speed up computations on small integers. The performance of `Integer` is now only slightly slower than `Int` for values between `minBound::Int` and `maxBound::Int`.
- On Win32 platforms, added support for using (and producing) dynamic link libraries (DLLs) containing ghc-compiled code.
- Added `-funbox-strict-fields` for unboxing/unpacking strict constructor fields.

- GHC now has a license! Check out `fptools/ghc/LICENSE/`.
- Added CPR analysis, which reduces allocation by unboxing function results (thanks to Kevin Glynn <`keving@cs.mu.OZ.AU`>).
- UsageSP analysis added, but not yet functional.
- Added a simple common sub-expression analysis pass.
- Implemented update-in-place for certain very restricted cases.
- Replaced GHC's old and worn yacc/lex parser with a new one based on Happy. Fixed several Haskell 98 non-conformance issues in the process.
- Added `Concurrent.yield :: IO ()`.
- Added RULES pragms - transformation rules for Haskell source. This is used for list fusion: now intermediate lists between map, foldr, list comprehensions are removed automatically.
- Unregisterised/unmangled builds now work.
- Much performance tuning: GHC 4.04 produces faster code than all previous compilers.
- GHC now defines `__HASKELL98__` when compiling files with `-cpp`.
- `hppa1.1-hp-hpux` port reinstated.
- New operations for unsafely thawing byte arrays: `unsafeThaw{Byte,ST,IO}Array`.
- `mkdependHS` now lives in GHC's lib directory, not the binary directory. It isn't intended to be run standalone, only via `ghc -M`.
- Asynchronous exceptions are now supported (see Section 5.12.1 (The Concurrent Library)). New operations:
  - `Exception.killThread` now raises an exception (`AsyncException.KilledThread`) in the killed thread.
  - `Exception.raiseInThread` allows a thread to raise an exception in another thread.
  - `Concurrent.myThreadId` returns the `ThreadId` of the calling thread.
  - Stack overflow results in `AsyncException.StackOverflow` being raised in the offending thread.
- Assertion failures now raise an `AssertionFailed` exception.
- Added simple high-level interface to the Regex library, see Section 6.7.3 (The RegexString Library).
- `forkIO` now handles any uncaught exceptions cleanly.
- `integer2Int#` isn't sign preserving any longer, but modulo `(maxBound::Int + 1) * 2`.
- `-ddump-rdr` is now called `-ddump-parsed`.
- Signal handling with the `Posix` library now works.
- Many, many bugs fixed.

2 Using GHC

GHC is a command-line compiler: in order to compile a Haskell program, GHC must be invoked on the source file(s) by typing a command to the shell. The steps involved in compiling a program can be automated using the `make` tool (this is especially useful if the program consists of multiple source files which depend on each other). This section describes how to use GHC from the command-line.

2.1 Overall command-line structure

An invocation of GHC takes the following form:

```
ghc [argument...]
```

Command-line arguments are either options or file names.

Command-line options begin with `-`. They may *not* be grouped: `-vO` is different from `-v -O`. Options need not precede filenames: e.g., `ghc *.o -o foo`. All options are processed and then applied to all files; you cannot, for example, invoke `ghc -c -O1 Foo.hs -O2 Bar.hs` to apply different optimisation levels to the files `Foo.hs` and `Bar.hs`. For conflicting options, e.g., `-c -S`, we reserve the right to do anything we want. (Usually, the last one applies.)

2.2 Meaningful file suffixes

File names with "meaningful" suffixes (e.g., `.lhs` or `.o`) cause the "right thing" to happen to those files.

`.lhs`:
: A "literate Haskell" module.

`.hs`:
: A not-so-literate Haskell module.

`.hi`:
: A Haskell interface file, probably compiler-generated.

`.hc`:
: Intermediate C file produced by the Haskell compiler.

`.c`:
: A C file not produced by the Haskell compiler.

`.s`:
: An assembly-language source file, usually produced by the compiler.

`.o`:
: An object file, produced by an assembler.

Files with other suffixes (or without suffixes) are passed straight to the linker.

2.3 Help and verbosity options

A good option to start with is the `-help` (or `-?`) option. GHC spews a long message to standard output and then exits.

The `-v` option makes GHC *verbose*: it reports its version number and shows (on stderr) exactly how it invokes each phase of the compilation system. Moreover, it passes the `-v` flag to most phases; each reports its version number (and possibly some other information).

Please, oh please, use the `-v` option when reporting bugs! Knowing that you ran the right bits in the right order is always the first thing we want to verify.

If you're just interested in the compiler version number, the `-version` option prints out a one-line string containing the requested info.

2.4 Running the right phases in the right order

The basic task of the `ghc` driver is to run each input file through the right phases (compiling, linking, etc.).

The first phase to run is determined by the input-file suffix, and the last phase is determined by a flag. If no relevant flag is present, then go all the way through linking. This table summarises:

Phase of the compilation system	Suffix saying "start here"	Flag saying "stop after"	(suffix of) output file
literate pre-processor	.lhs	-	-
C pre-processor (opt.)	-	-	-
Haskell compiler	.hs	-C, -S	.hc, .s
C compiler (opt.)	.hc or .c	-S	.s
assembler	.s	-c	.o
linker	other	-	a.out

Thus, a common invocation would be: `ghc -c Foo.hs`

Note: What the Haskell compiler proper produces depends on whether a native-code generator is used (producing assembly language) or not (producing C).

The option `-cpp` must be given for the C pre-processor phase to be run, that is, the pre-processor will be run over your Haskell source file before continuing.

The option `-E` runs just the pre-processing passes of the compiler, outputting the result on stdout before stopping. If used in conjunction with -cpp, the output is the code blocks of the original (literal) source after having put it through the grinder that is the C pre-processor. Sans `-cpp`, the output is the de-litted version of the original source.

The option `-optcpp-E` runs just the pre-processing stage of the C-compiling phase, sending the result to stdout. (For debugging or obfuscation contests, usually.)

2.5 Re-directing the compilation output(s)

GHC's compiled output normally goes into a `.hc`, `.o`, etc., file, depending on the last-run compilation phase. The option `-o foo` re-directs the output of that last-run phase to file `foo`.

Note: this "feature" can be counterintuitive: `ghc -C -o foo.o foo.hs` will put the intermediate C code in the file `foo.o`, name notwithstanding!

EXOTICA: But the `-o` option isn't of much use if you have *several* input files... Non-interface output files are normally put in the same directory as their corresponding input file came from. You may specify that they be put in another directory using the `-odir <dir>` (the "Oh, dear" option). For example:

```
% ghc -c parse/Foo.hs parse/Bar.hs gurgle/Bumble.hs -odir `arch`
```

The output files, `Foo.o`, `Bar.o`, and `Bumble.o` would be put into a subdirectory named after the architecture of the executing machine (`sun4`, `mips`, etc). The directory must already exist; it won't be created.

Note that the `-odir` option does *not* affect where the interface files are put. In the above example, they would still be put in `parse/Foo.hi`, `parse/Bar.hi`, and `gurgle/Bumble.hi`.

MORE EXOTICA: The `-osuf <suffix>` will change the `.o` file suffix for object files to whatever you specify. (We use this in compiling the prelude.). Similarly, the `-hisuf <suffix>` will change the `.hi` file suffix for non-system interface files (see Section 2.7.3 (Other options related to interface files)).

The `-hisuf`/`-osuf` game is useful if you want to compile a program with both GHC and HBC (say) in the same directory. Let HBC use the standard `.hi`/`.o` suffixes; add `-hisuf g_hi -osuf g_o` to your `make` rule for GHC compiling...

FURTHER EXOTICA: If you are doing a normal `.hs`-to-`.o` compilation but would like to hang onto the intermediate `.hc` C file, just throw in a `-keep-hc-file-too` option. If you would like to look at the assembler output, toss in a `-keep-s-file-too`, too.

2.5.1 Saving GHC's standard error output

Sometimes, you may cause GHC to be rather chatty on standard error; with `-v`, for example. You can instruct GHC to *append* this output to a particular log file with a `-odump <blah>` option.

2.5.2 Redirecting temporary files

If you have trouble because of running out of space in `/tmp` (or wherever your installation thinks temporary files should go), you may use the `-tmpdir <dir>` option to specify an alternate directory. For example, `-tmpdir .` says to put temporary files in the current working directory.

Alternatively, use your `TMPDIR` environment variable. Set it to the name of the directory where temporary files should be put. GCC and other programs will honour the `TMPDIR` variable as well.

Even better idea: Set the `TMPDIR` variable when building GHC, and never worry about `TMPDIR` again. (see the build documentation).

2.6 Warnings and sanity-checking

GHC has a number of options that select which types of non-fatal error messages, otherwise known as warnings, can be generated during compilation. By default, you get a standard set of warnings which are generally likely to indicate bugs in your program. These are: `-fwarn-overlpapping-patterns`, `-fwarn-duplicate-exports`, and `-fwarn-missing-methods`. The following flags are simple ways to select standard "packages" of warnings:

`-Wnot`:

Turns off all warnings, including the standard ones.

`-w`:

Synonym for `-Wnot`.

`-W`:

Provides the standard warnings plus `-fwarn-incomplete-patterns`, `-fwarn-unused-imports` and `-fwarn-unused-binds`.

`-Wall`:

Turns on all warning options.

The full set of warning options is described below. To turn off any warning, simply give the corresponding `-fno-warn-...` option on the command line.

`-fwarn-name-shadowing`:

This option causes a warning to be emitted whenever an inner-scope value has the same name as an outer-scope value, i.e. the inner value shadows the outer one. This can catch typographical errors that turn into hard-to-find bugs, e.g., in the inadvertent cyclic definition `let x = ... x ... in`.

Consequently, this option does *not* allow cyclic recursive definitions.

`-fwarn-overlapping-patterns`:

By default, the compiler will warn you if a set of patterns are overlapping, i.e.,

```
f :: String -> Int
f []     = 0
f (_:xs) = 1
f "2"    = 2
```

where the last pattern match in `f` won't ever be reached, as the second pattern overlaps it. More often than not, redundant patterns is a programmer mistake/error, so this option is enabled by default.

`-fwarn-incomplete-patterns`:

Similarly for incomplete patterns, the function `g` below will fail when applied to non-empty lists, so the compiler will emit a warning about this when `-fwarn-incomplete-patterns` is enabled.

```
g [] = 2
```

This option isn't enabled be default because it can be a bit noisy, and it doesn't always indicate a bug in the program. However, it's generally considered good practice to cover all the cases in your functions.

`-fwarn-missing-methods`:

This option is on by default, and warns you whenever an instance declaration is missing one or more methods, and the corresponding class declaration has no default declaration for them.

`-fwarn-unused-imports`:

Report any objects that are explicitly imported but never used.

`-fwarn-unused-binds`:

Report any function definitions (and local bindings) which are unused. For top-level functions, the warning is only given if the binding is not exported.

`-fwarn-unused-matches`:

Report all unused variables which arise from pattern matches, including patterns consisting of a single variable. For instance `f x y = []` would report `x` and `y` as unused. To eliminate the warning, all unused variables can be replaced with wildcards.

`-fwarn-duplicate-exports`:

Have the compiler warn about duplicate entries in export lists. This is useful information if you maintain large export lists, and want to avoid the continued export of a definition after you've deleted (one) mention of it in the export list.

This option is on by default.

`-fwarn-type-defaults`:

Have the compiler warn/inform you where in your source the Haskell defaulting mechanism for numeric types kicks in. This is useful information when converting code from a context that assumed one default into one with another, e.g., the 'default default' for Haskell 1.4 caused the otherwise unconstrained value `1` to be given the type `Int`, whereas Haskell 98 defaults it to `Integer`. This may lead to differences in performance and behaviour, hence the usefulness of being non-silent about this.

This warning is off by default.

`-fwarn-missing-signatures`:

If you would like GHC to check that every top-level function/value has a type signature, use the `-fwarn-missing-signatures` option. This option is off by default.

If you're feeling really paranoid, the `-dcore-lint` option is a good choice. It turns on heavyweight intra-pass sanity-checking within GHC. (It checks GHC's sanity, not yours.)

2.7 Separate compilation

This section describes how GHC supports separate compilation.

2.7.1 Interface files

When GHC compiles a source file `F` which contains a module `A`, say, it generates an object `F.o`, *and* a companion *interface file* `A.hi`. The interface file is not intended for human consumption, as you'll see if you take a look at one. It's merely there to help the compiler compile other modules in the same program.

NOTE: Having the name of the interface file follow the module name and not the file name, means that working with tools such as `make(1)` become harder. `make` implicitly assumes that any output files produced by processing a translation unit will have file names that can be derived from the file name of the translation unit. For instance, pattern rules becomes unusable. For this reason, we recommend you stick to using the same file name as the module name.

The interface file for `A` contains information needed by the compiler when it compiles any module `B` that imports `A`, whether directly or indirectly. When compiling `B`, GHC will read `A.hi` to find the details that it needs to know about things defined in `A`.

Furthermore, when compiling module `C` which imports `B`, GHC may decide that it needs to know something about `A` — for example, `B` might export a function that involves a type defined in `A`. In this case, GHC will go and read `A.hi` even though `C` does not explicitly import `A` at all.

The interface file may contain all sorts of things that aren't explicitly exported from `A` by the programmer. For example, even though a data type is exported abstractly, `A.hi` will contain the full data type definition. For small function definitions, `A.hi` will contain the complete definition of the function. For bigger functions, `A.hi` will contain strictness information about the function. And so on. GHC puts much more information into `.hi` files when optimisation is turned on with the `-O` flag. Without `-O` it puts in just the minimum; with `-O` it lobs in a whole pile of stuff.

`A.hi` should really be thought of as a compiler-readable version of `A.o`. If you use a `.hi` file that wasn't generated by the same compilation run that generates the `.o` file the compiler may assume all sorts of incorrect things about `A`, resulting in core dumps and other unpleasant happenings.

2.7.2 Finding interface files

In your program, you import a module `Foo` by saying `import Foo`. GHC goes looking for an interface file, `Foo.hi`. It has a builtin list of directories (notably including `.`) where it looks.

`-i<dirs>`

This flag prepends a colon-separated list of `dirs` to the "import directories" list.

`-i`

resets the "import directories" list back to nothing.

`-fno-implicit-prelude`

GHC normally imports `Prelude.hi` files for you. If you'd rather it didn't, then give it a `-fno-implicit-prelude` option. You are unlikely to get very far without a Prelude, but, hey, it's a free country.

`-syslib <lib>`

If you are using a system-supplied non-Prelude library (e.g., the POSIX library), just use a `-syslib posix` option (for example). The right interface files should then be available. Section 6 (The GHC Prelude and Libraries) lists the libraries available by this mechanism.

`-I<dir>`

Once a Haskell module has been compiled to C (`.hc` file), you may wish to specify where GHC tells the C compiler to look for `.h` files. (Or, if you are using the `-cpp` option, where it tells the C pre-processor to look...) For this purpose, use a `-I` option in the usual C-ish way.

2.7.3 Other options related to interface files

The interface output may be directed to another file `bar2/Wurble.iface` with the option `-ohi bar2/Wurble.iface` (not recommended).

To avoid generating an interface file at all, use a `-nohi` option.

The compiler does not overwrite an existing `.hi` interface file if the new one is byte-for-byte the same as the old one; this is friendly to `make`. When an interface does change, it is often enlightening to be informed. The `-hi-diffs` option will make `ghc` run `diff` on the old and new `.hi` files. You can also record the difference in the interface file itself, the `-keep-hi-diffs` option takes care of that.

The `.hi` files from GHC contain "usage" information which changes often and uninterestingly. If you really want to see these changes reported, you need to use the `-hi-diffs-with-usages` option.

Interface files are normally jammed full of compiler-produced *pragmas*, which record arities, strictness info, etc. If you think these pragmas are messing you up (or you are doing some kind of weird experiment), you can tell GHC to ignore them with the `-fignore-interface-pragmas` option.

When compiling without optimisations on, the compiler is extra-careful about not slurping in data constructors and instance declarations that it will not need. If you believe it is getting it wrong and not importing stuff which you think it should, this optimisation can be turned off with `-fno-prune-tydecls` and `-fno-prune-instdecls`.

See also Section 2.9.3 (Linking and consistency-checking), which describes how the linker finds standard Haskell libraries.

2.7.4 The recompilation checker

In the olden days, GHC compared the newly-generated `.hi` file with the previous version; if they were identical, it left the old one alone and didn't change its modification date. In consequence, importers of a module with an unchanged output `.hi` file were not recompiled.

This doesn't work any more. In our earlier example, module `C` does not import module `A` directly, yet changes to `A.hi` should force a recompilation of `C`. And some changes to `A` (changing the definition of a function that appears in an inlining of a function exported by `B`, say) may conceivably not change `B.hi` one jot. So now...

GHC keeps a version number on each interface file, and on each type signature within the interface file. It also keeps in every interface file a list of the version numbers of everything it used when it last compiled the file. If the source file's modification date is earlier than the `.o` file's date (i.e. the source hasn't changed since the file was last compiled), and you give GHC the `-recomp` flag, then GHC will be clever. It compares the version numbers on the things it needs this time with the version numbers on the things it needed last time (gleaned from the interface file of the module being compiled); if they are all the same it stops compiling rather early in the process saying "Compilation IS NOT required". What a beautiful sight!

It's still an experimental feature (that's why `-recomp` is off by default), so tell us if you think it doesn't work.

Patrick Sansom has a workshop paper about how all this is done. Ask him (email: *sansom@dcs.gla.ac.uk*) if you want a copy.

2.7.5 Using `make`

It is reasonably straightforward to set up a `Makefile` to use with GHC, assuming you name your source files the same as your modules. Thus:

```
HC      = ghc
```

```
HC_OPTS = -cpp $(EXTRA_HC_OPTS)

SRCS = Main.lhs Foo.lhs Bar.lhs
OBJS = Main.o   Foo.o   Bar.o

.SUFFIXES : .o .hi .lhs .hc .s

cool_pgm : $(OBJS)
        rm $@
        $(HC) -o $@ $(HC_OPTS) $(OBJS)

# Standard suffix rules
.o.hi:
        @:

.lhs.o:
        $(HC) -c $< $(HC_OPTS)

.hs.o:
        $(HC) -c $< $(HC_OPTS)

# Inter-module dependencies
Foo.o Foo.hc Foo.s    : Baz.hi          # Foo imports Baz
Main.o Main.hc Main.s : Foo.hi Baz.hi   # Main imports Foo and Baz
```

(Sophisticated `make` variants may achieve some of the above more elegantly. Notably, `gmake`'s pattern rules let you write the more comprehensible:

```
%.o : %.lhs
        $(HC) -c $< $(HC_OPTS)
```

What we've shown should work with any `make`.)

Note the cheesy `.o.hi` rule: It records the dependency of the interface (`.hi`) file on the source. The rule says a `.hi` file can be made from a `.o` file by doing... nothing. Which is true.

Note the inter-module dependencies at the end of the Makefile, which take the form

```
Foo.o Foo.hc Foo.s    : Baz.hi          # Foo imports Baz
```

They tell `make` that if any of `Foo.o`, `Foo.hc` or `Foo.s` have an earlier modification date than `Baz.hi`, then the out-of-date file must be brought up to date. To bring it up to date, `make` looks for a rule to do so; one of the preceding suffix rules does the job nicely.

Putting inter-dependencies of the form `Foo.o : Bar.hi` into your `Makefile` by hand is rather error-prone. Don't worry—never fear, `mkdependHS` is here! (and is distributed as part of GHC) Add the following to your `Makefile`:

```
depend :
        mkdependHS -- $(HC_OPTS) -- $(SRCS)
```

Now, before you start compiling, and any time you change the `imports` in your program, do `make depend` before you do `make cool_pgm`. `mkdependHS` will append the needed dependencies to your `Makefile`. `mkdependHS` is fully describe in Section 8.1 (Makefile dependencies in Haskell: using mkdependHS).

A few caveats about this simple scheme:

- You may need to compile some modules explicitly to create their interfaces in the first place (e.g., `make Bar.o` to create `Bar.hi`).
- You may have to type `make` more than once for the dependencies to have full effect. However, a `make` run that does nothing *does* mean "everything's up-to-date."
- This scheme will work with mutually-recursive modules but, again, it may take multiple iterations to "settle."

2.7.6 How to compile mutually recursive modules

Currently, the compiler does not have proper support for dealing with mutually recursive modules:

```
module A where

import B

newtype TA = MkTA Int

f :: TB -> TA
f (MkTB x) = MkTA x
--------
module B where

import A

data TB = MkTB !Int

g :: TA -> TB
g (MkTA x) = MkTB x
```

When compiling either module A and B, the compiler will try (in vain) to look for the interface file of the other. So, to get mutually recursive modules off the ground, you need to hand write an interface file for A or B, so as to break the loop. These hand-written interface files are called `hi-boot` files, and are placed in a file called `<module>.hi-boot`. To import from an `hi-boot` file instead of the standard `.hi` file, use the following syntax in the importing module:

```
import {-# SOURCE #-} A
```

The hand-written interface need only contain the bare minimum of information needed to get the boot-strapping process started. For example, it doesn't need to contain declarations for *everything* that module `A` exports, only the things required by the module that imports `A` recursively.

For the example at hand, the boot interface file for A would look like the following:

```
__interface A 1 404 where
__export A TA{MkTA} ;
1 newtype TA = MkTA PrelBase.Int ;
```

The syntax is essentially the same as a normal `.hi` file (unfortunately), but you can usually tailor an existing `.hi` file to make a `.hi-boot` file.

Notice that we only put the declaration for the newtype `TA` in the `hi-boot` file, not the signature for `f`, since `f` isn't used by `B`.

The number "1" after "__interface A" gives the version number of module A; it is incremented whenever anything in A's interface file changes. The "404" is the version number of the interface file *syntax*; we change it when we change the syntax of interface files so that you get a better error message when you try to read an old-format file with a new-format compiler.

The number "1" at the beginning of a declaration is the *version number* of that declaration: for the purposes of `.hi-boot` files these can all be set to 1. All names must be fully qualified with the *original* module that an object comes from: for example, the reference to `Int` in the interface for `A` comes from `PrelBase`, which is a module internal to GHC's prelude. It's a pain, but that's the way it is.

If you want an hi-boot file to export a data type, but you don't want to give its constructors (because the constructors aren't used by the SOURCE-importing module), you can write simply:

```
__interface A 1 404 where
__export A TA;
1 data TA
```

(You must write all the type parameters, but leave out the '=' and everything that follows it.)

Note: This is all a temporary solution, a version of the compiler that handles mutually recursive properly without the manual construction of interface files, is (allegedly) in the works.

2.8 Optimisation (code improvement)

The `-O*` options specify convenient "packages" of optimisation flags; the `-f*` options described later on specify *individual* optimisations to be turned on/off; the `-m*` options specify *machine-specific* optimisations to be turned on/off.

2.8.1 `-O*`: convenient "packages" of optimisation flags.

There are *many* options that affect the quality of code produced by GHC. Most people only have a general goal, something like "Compile quickly" or "Make my program run like greased lightning." The following "packages" of optimisations (or lack thereof) should suffice.

Once you choose a `-O*` "package," stick with it—don't chop and change. Modules' interfaces *will* change with a shift to a new `-O*` option, and you may have to recompile a large chunk of all importing modules before your program can again be run safely (see Section 2.7.4 (The recompilation checker)).

No `-O*`-type option specified:

This is taken to mean: "Please compile quickly; I'm not over-bothered about compiled-code quality." So, for example: `ghc -c Foo.hs`

`-O` **or** `-O1`**:**

Means: "Generate good-quality code without taking too long about it." Thus, for example: `ghc -c -O Main.lhs`

`-O2`**:**

Means: "Apply every non-dangerous optimisation, even if it means significantly longer compile times."

The avoided "dangerous" optimisations are those that can make runtime or space *worse* if you're unlucky. They are normally turned on or off individually.

At the moment, `-O2` is *unlikely* to produce better code than `-O`.

`-O2-for-C`**:**

Says to run GCC with `-O2`, which may be worth a few percent in execution speed. Don't forget `-fvia-C`, lest you use the native-code generator and bypass GCC altogether!

`-Onot`**:**

This option will make GHC "forget" any `-O`ish options it has seen so far. Sometimes useful; for example: `make all EXTRA_HC_OPTS=-Onot`.

`-Ofile <file>`**:**

For those who need *absolute* control over *exactly* what options are used (e.g., compiler writers, sometimes :-), a list of options can be put in a file and then slurped in with `-Ofile`.

In that file, comments are of the `#`-to-end-of-line variety; blank lines and most whitespace is ignored.

Please ask if you are baffled and would like an example of `-Ofile`!

At Glasgow, we don't use a `-O*` flag for day-to-day work. We use `-O` to get respectable speed; e.g., when we want to measure something. When we want to go for broke, we tend to use `-O -fvia-C -O2-for-C` (and we go for lots of coffee breaks).

The easiest way to see what `-O` (etc.) "really mean" is to run with `-v`, then stand back in amazement. Alternatively, just look at the `HsC_minus<blah>` lists in the `ghc` driver script.

2.8.2 `-f*`: platform-independent flags

Flags can be turned *off* individually. (NB: I hope you have a good reason for doing this....) To turn off the `-ffoo` flag, just use the `-fno-foo` flag. So, for example, you can say `-O2 -fno-strictness`, which will then drop out any running of the strictness analyser.

The options you are most likely to want to turn off are: `-fno-strictness` (strictness analyser [because it is sometimes slow]), `-fno-specialise` (automatic specialisation of overloaded functions [because it makes your code bigger]) [US spelling also accepted], and `-fno-update-analysis` (update analyser, because it sometimes takes a *long* time). This one is only enabled with -O2 anyway.

Should you wish to turn individual flags *on*, you are advised to use the `-Ofile` option, described above. Because the order in which optimisation passes are run is sometimes crucial, it's quite hard to do with command-line options.

Here are some "dangerous" optimisations you *might* want to try:

`-fvia-C`:

Compile via C, and don't use the native-code generator. (There are many cases when GHC does this on its own.) You might pick up a little bit of speed by compiling via C. If you use `_ccall_gc_`s or `_casm_`s, you probably *have to* use `-fvia-C`.

The lower-case incantation, `-fvia-c`, is synonymous.

Compiling via C will probably be slower (in compilation time) than using GHC's native code generator.

`-funfolding-interface-threshold<n>`:

(Default: 30) By raising or lowering this number, you can raise or lower the amount of pragmatic junk that gets spewed into interface files. (An unfolding has a "size" that reflects the cost in terms of "code bloat" of expanding that unfolding in another module. A bigger function would be assigned a bigger cost.)

`-funfolding-creation-threshold<n>`:

(Default: 30) This option is similar to `-funfolding-interface-threshold`, except that it governs unfoldings within a single module. Increasing this figure is more likely to result in longer compile times than faster code. The next option is more useful:

`-funfolding-use-threshold<n>`:

(Default: 8) This is the magic cut-off figure for unfolding: below this size, a function definition will be unfolded at the call-site, any bigger and it won't. The size computed for a function depends on two things: the actual size of the expression minus any discounts that apply (see `-funfolding-con-discount`).

`-funfolding-con-discount<n>`:

(Default: 2) If the compiler decides that it can eliminate some computation by performing an unfolding, then this is a discount factor that it applies to the funciton size before deciding whether to unfold it or not.

OK, folks, these magic numbers '30', '8', and '2' are mildly arbitrary; they are of the "seem to be OK" variety. The '8' is the more critical one; it's what determines how eager GHC is about expanding unfoldings.

`-funbox-strict-fields`:

This option causes all constructor fields which are marked strict (i.e. "!") to be unboxed or unpacked if possible. For example:

```
data T = T !Float !Float
```

will create a constructor `T` containing two unboxed floats if the `-funbox-strict-fields` flag is given. This may not always be an optimisation: if the `T` constructor is scrutinised and the floats passed to a non-strict function for example, they will have to be reboxed (this is done automatically by the compiler).

This option should only be used in conjunction with `-O`, in order to expose unfoldings to the compiler so the reboxing can be removed as often as possible. For example:

```
f :: T -> Float
f (T f1 f2) = f1 + f2
```

The compiler will avoid reboxing `f1` and `f2` by inlining `+` on floats, but only when `-O` is on.

Any single-constructor data is eligible for unpacking; for example

```
data T = T !(Int,Int)
```

will store the two `Int`s directly in the `T` constructor, by flattening the pair. Multi-level unpacking is also supported:

```
data T = T !S
data S = S !Int !Int
```

will store two unboxed `Int#`s directly in the `T` constructor.

`-fsemi-tagging`:

This option (which *does not work* with the native-code generator) tells the compiler to add extra code to test for already-evaluated values. You win if you have lots of such values during a run of your program, you lose otherwise. (And you pay in extra code space.)

We have not played with `-fsemi-tagging` enough to recommend it. (For all we know, it doesn't even work anymore... Sigh.)

2.8.3 `-m*`: platform-specific flags

Some flags only make sense for particular target platforms.

`-mv8`:

(SPARC machines) Means to pass the like-named option to GCC; it says to use the Version 8 SPARC instructions, notably integer multiply and divide. The similiar `-m*` GCC options for SPARC also work, actually.

`-mlong-calls`:

(HPPA machines) Means to pass the like-named option to GCC. Required for Very Big modules, maybe. (Probably means you're in trouble...)

`-monly-[32]-regs`:

(iX86 machines) GHC tries to "steal" four registers from GCC, for performance reasons; it almost always works. However, when GCC is compiling some modules with four stolen registers, it will crash, probably saying:

```
Foo.hc:533: fixed or forbidden register was spilled.
This may be due to a compiler bug or to impossible asm
statements or clauses.
```

Just give some registers back with `-monly-N-regs`. Try '3' first, then '2'. If '2' doesn't work, please report the bug to us.

2.8.4 Code improvement by the C compiler.

The C compiler (GCC) is run with `-O` turned on. (It has to be, actually).

If you want to run GCC with `-O2`—which may be worth a few percent in execution speed—you can give a `-O2-for-C` option.

2.9 Options related to a particular phase

2.9.1 The C pre-processor

The C pre-processor `cpp` is run over your Haskell code only if the `-cpp` option is given. Unless you are building a large system with significant doses of conditional compilation, you really shouldn't need it.

`-D<foo>`:

Define macro `<foo>` in the usual way. NB: does *not* affect `-D` macros passed to the C compiler when compiling via C! For those, use the `-optc-Dfoo` hack... (see Section 2.13.2 (Forcing options to a particular phase.)).

`-U<foo>`:

Undefine macro `<foo>` in the usual way.

`-I<dir>`:

Specify a directory in which to look for `#include` files, in the usual C way.

The `ghc` driver pre-defines several macros:

`__HASKELL98__`:

If defined, this means that GHC supports the language defined by the Haskell 98 report.

NB. This macro is *only* set when pre-processing Haskell source (ie. `.hs` or `.lhs` files).

`__HASKELL1__`:

If defined to n, that means GHC supports the Haskell language defined in the Haskell report version *1.n*. Currently 5.

NB. As with `__HASKELL98__`, this macro is *only* set when pre-processing Haskell source (ie. `.hs` or `.lhs` files).

`__GLASGOW_HASKELL__`:

For version n of the GHC system, this will be `#defined` to *100n*. So, for version 4.00, it is 400.

With any luck, `__GLASGOW_HASKELL__` will be undefined in all other implementations that support C-style pre-processing.

(For reference: the comparable symbols for other systems are: `__HUGS__` for Hugs and `__HBC__` for Chalmers.)

NB. This macro is set when pre-processing both Haskell source and C source, including the C source generated from a Haskell module (ie. `.hs`, `.lhs`, `.c` and `.hc` files).

`__CONCURRENT_HASKELL__`:

This symbol is defined when pre-processing Haskell (input) and pre-processing C (GHC output). Since GHC from verion 4.00 now supports concurrent haskell by default, this symbol is always defined.

`__PARALLEL_HASKELL__`:

Only defined when `-parallel` is in use! This symbol is defined when pre-processing Haskell (input) and pre-processing C (GHC output).

Options other than the above can be forced through to the C pre-processor with the `-opt` flags (see Section 2.13.2 (Forcing options to a particular phase.)).

A small word of warning: `-cpp` is not friendly to "string gaps".. In other words, strings such as the following:

```
strmod = "\
\ p \
\ "
```

don't work with `-cpp`; `/usr/bin/cpp` elides the backslash-newline pairs.

However, it appears that if you add a space at the end of the line, then `cpp` (at least GNU `cpp` and possibly other `cpp`s) leaves the backslash-space pairs alone and the string gap works as expected.

2.9.2 Options affecting the C compiler (if applicable)

At the moment, quite a few common C-compiler options are passed on quietly to the C compilation of Haskell-compiler-generated C files. THIS MAY CHANGE. Meanwhile, options so sent are:

`-ansi`	do ANSI C (not K&R)
`-pedantic`	be so
`-dgcc-lint`	(hack) short for "make GCC very paranoid"

If you are compiling with lots of `ccalls`, etc., you may need to tell the C compiler about some `#include` files. There is no real pretty way to do this, but you can use this hack from the command-line:

```
% ghc -c '-#include <X/Xlib.h>' Xstuff.lhs
```

2.9.3 Linking and consistency-checking

GHC has to link your code with various libraries, possibly including: user-supplied, GHC-supplied, and system-supplied (`-lm` math library, for example).

`-l<FOO>`:

Link in a library named `lib<FOO>.a` which resides somewhere on the library directories path.

Because of the sad state of most UNIX linkers, the order of such options does matter. Thus: `ghc -lbar *.o` is almost certainly wrong, because it will search `libbar.a` *before* it has collected unresolved symbols from the `*.o` files. `ghc *.o -lbar` is probably better.

The linker will of course be informed about some GHC-supplied libraries automatically; these are:

-l equivalent	**description**
`-lHSrts,-lHSclib`	basic runtime libraries
`-lHS`	standard Prelude library
`-lHS_cbits`	C support code for standard Prelude library
`-lgmp`	GNU multi-precision library (for Integers)

`-syslib <name>`:

If you are using a Haskell "system library" (e.g., the POSIX library), just use the `-syslib posix` option, and the correct code should be linked in.

`-L<dir>`:

Where to find user-supplied libraries... Prepend the directory `<dir>` to the library directories path.

`-static`:

Tell the linker to avoid shared libraries.

`-no-link-chk` **and** `-link-chk`:

By default, immediately after linking an executable, GHC verifies that the pieces that went into it were compiled with compatible flags; a "consistency check". (This is to avoid mysterious failures caused by non-meshing of incompatibly-compiled programs; e.g., if one `.o` file was compiled for a parallel machine and the others weren't.) You may turn off this check with `-no-link-chk`. You can turn it (back) on with `-link-chk` (the default).

`-no-hs-main`:

In the event you want to include ghc-compiled code as part of another (non-Haskell) program, the RTS will not be supplying its definition of `main()` at link-time, you will have to. To signal that to the driver script when linking, use `-no-hs-main`.

Notice that since the command-line passed to the linker is rather involved, you probably want to use the ghc driver script to do the final link of your 'mixed-language' application. This is not a requirement though, just try linking once with `-v` on to see what options the driver passes through to the linker.

2.10 Using Concurrent Haskell

GHC (as of version 4.00) supports Concurrent Haskell by default, without requiring a special option or libraries compiled in a certain way. To get access to the support libraries for Concurrent Haskell (ie. `Concurrent` and friends), use the `-syslib concurrent` option.

Three RTS options are provided for modifying the behaviour of the threaded runtime system. See the descriptions of `-C[<us>]`, `-q`, and `-t<num>` in Section 2.11.4 (RTS options for Concurrent/Parallel Haskell).

Concurrent Haskell is described in more detail in Section 5.12 (Concurrent and Parallel Haskell).

2.11 Using Parallel Haskell

[You won't be able to execute parallel Haskell programs unless PVM3 (Parallel Virtual Machine, version 3) is installed at your site.]

To compile a Haskell program for parallel execution under PVM, use the `-parallel` option, both when compiling *and linking*. You will probably want to `import Parallel` into your Haskell modules.

To run your parallel program, once PVM is going, just invoke it "as normal". The main extra RTS option is `-N<n>`, to say how many PVM "processors" your program to run on. (For more details of all relevant RTS options, please see Section 2.11.4 (RTS options for Concurrent/Parallel Haskell).)

In truth, running Parallel Haskell programs and getting information out of them (e.g., parallelism profiles) is a battle with the vagaries of PVM, detailed in the following sections.

2.11.1 Dummy's guide to using PVM

Before you can run a parallel program under PVM, you must set the required environment variables (PVM's idea, not ours); something like, probably in your `.cshrc` or equivalent:

```
setenv PVM_ROOT /wherever/you/put/it
setenv PVM_ARCH `$PVM_ROOT/lib/pvmgetarch`
setenv PVM_DPATH $PVM_ROOT/lib/pvmd
```

Creating and/or controlling your "parallel machine" is a purely-PVM business; nothing specific to Parallel Haskell.

You use the `pvm` command to start PVM on your machine. You can then do various things to control/monitor your "parallel machine;" the most useful being:

\begin{tabular}{ll} `Control-D` & exit `pvm`, leaving it running \\ `halt` & kill off this "parallel machine" \& exit \\ `add <host>` & add `<host>` as a processor \\ `delete <host>` & delete `<host>` \\ `reset` & kill what's going, but leave PVM up \\ `conf` & list the current configuration \\ `ps` & report processes' status \\ `pstat <pid>` & status of a particular process \\ \end{tabular}

The PVM documentation can tell you much, much more about `pvm`!

2.11.2 Parallelism profiles

With Parallel Haskell programs, we usually don't care about the results—only with "how parallel" it was! We want pretty pictures.

Parallelism profiles (`a la `hbcpp`) can be generated with the `-q` RTS option. The per-processor profiling info is dumped into files named `<full-path><program>.gr`. These are then munged into a PostScript picture, which you can then display. For example, to run your program `a.out` on 8 processors, then view the parallelism profile, do:

```
% ./a.out +RTS -N8 -q
% grs2gr *.???.gr > temp.gr     # combine the 8 .gr files into one
% gr2ps -O temp.gr              # cvt to .ps; output in temp.ps
% ghostview -seascape temp.ps   # look at it!
```

The scripts for processing the parallelism profiles are distributed in `ghc/utils/parallel/`.

2.11.3 Other useful info about running parallel programs

The "garbage-collection statistics" RTS options can be useful for seeing what parallel programs are doing. If you do either `+RTS -Sstderr` or `+RTS -sstderr`, then you'll get mutator, garbage-collection, etc., times on standard error. The standard error of all PE's other than the 'main thread' appears in `/tmp/pvml.nnn`, courtesy of PVM.

Whether doing `+RTS -Sstderr` or not, a handy way to watch what's happening overall is: `tail -f /tmp/pvml.nnn`.

2.11.4 RTS options for Concurrent/Parallel Haskell

Besides the usual runtime system (RTS) options (Section 2.12 (Running a compiled program)), there are a few options particularly for concurrent/parallel execution.

`-N<N>`:

(PARALLEL ONLY) Use <N> PVM processors to run this program; the default is 2.

`-C[<us>]`:

Sets the context switch interval to <us> microseconds. A context switch will occur at the next heap allocation after the timer expires. With `-C0` or `-C`, context switches will occur as often as possible (at every heap allocation). By default, context switches occur every 10 milliseconds. Note that many interval timers are only capable of 10 millisecond granularity, so the default setting may be the finest granularity possible, short of a context switch at every heap allocation.

[NOTE: this option currently has no effect (version 4.00). Context switches happen when the current heap block is full, i.e. every 4k of allocation].

`-q[v]`:

(PARALLEL ONLY) Produce a quasi-parallel profile of thread activity, in the file <program>`.qp`. In the style of `hbcpp`, this profile records the movement of threads between the green (runnable) and red (blocked) queues. If you specify the verbose suboption (`-qv`), the green queue is split into green (for the currently running thread only) and amber (for other runnable threads). We do not recommend that you use the verbose suboption if you are planning to use the `hbcpp` profiling tools or if you are context switching at every heap check (with `-C`).

`-t<num>`:

(PARALLEL ONLY) Limit the number of concurrent threads per processor to <num>. The default is 32. Each thread requires slightly over 1K *words* in the heap for thread state and stack objects. (For 32-bit machines, this translates to 4K bytes, and for 64-bit machines, 8K bytes.)

`-d`:

(PARALLEL ONLY) Turn on debugging. It pops up one xterm (or GDB, or something...) per PVM processor. We use the standard `debugger` script that comes with PVM3, but we sometimes meddle with the `debugger2` script. We include ours in the GHC distribution, in `ghc/utils/pvm/`.

`-e<num>`:

(PARALLEL ONLY) Limit the number of pending sparks per processor to <num>. The default is 100. A larger number may be appropriate if your program generates large amounts of parallelism initially.

`-Q<num>`:

(PARALLEL ONLY) Set the size of packets transmitted between processors to <num>. The default is 1024 words. A larger number may be appropriate if your machine has a high communication cost relative to computation speed.

2.12 Running a compiled program

To make an executable program, the GHC system compiles your code and then links it with a non-trivial runtime system (RTS), which handles storage management, profiling, etc.

You have some control over the behaviour of the RTS, by giving special command-line arguments to your program.

When your Haskell program starts up, its RTS extracts command-line arguments bracketed between `+RTS` and `-RTS` as its own. For example:

```
% ./a.out -f +RTS -p -S -RTS -h foo bar
```

The RTS will snaffle `-p -S` for itself, and the remaining arguments `-f -h foo bar` will be handed to your program if/when it calls `System.getArgs`.

No `-RTS` option is required if the runtime-system options extend to the end of the command line, as in this example:

```
% hls -ltr /usr/etc +RTS -A5m
```

If you absolutely positively want all the rest of the options in a command line to go to the program (and not the RTS), use a `-RTS`.

As always, for RTS options that take <`size`>s: If the last character of `size` is a K or k, multiply by 1000; if an M or m, by 1,000,000; if a G or G, by 1,000,000,000. (And any wraparound in the counters is *your* fault!)

Giving a `+RTS -f` option will print out the RTS options actually available in your program (which vary, depending on how you compiled).

NOTE: to send RTS options to the compiler itself, you need to prefix the option with `-optCrts`, eg. to increase the maximum heap size for a compilation to 128M, you would add `-optCrts-M128m` to the command line. The compiler understands some options directly without needing `-optCrts`: these are `-H` and `-K`.

2.12.1 RTS options to control the garbage-collector

There are several options to give you precise control over garbage collection. Hopefully, you won't need any of these in normal operation, but there are several things that can be tweaked for maximum performance.

`-A<size>`:

[Default: 256k] Set the allocation area size used by the garbage collector. The allocation area (actually generation 0 step 0) is fixed and is never resized (unless you use `-H`, below).

Increasing the allocation area size may or may not give better performance (a bigger allocation area means worse cache behaviour but fewer garbage collections and less promotion).

With only 1 generation (`-G1`) the `-A` option specifies the minimum allocation area, since the actual size of the allocation area will be resized according to the amount of data in the heap (see `-F`, below).

`-F<factor>`:

[Default: 2] This option controls the amount of memory reserved for the older generations (and in the case of a two space collector the size of the allocation area) as a factor of the amount of live data. For example, if there was 2M of live data in the oldest generation when we last collected it, then by default we'll wait until it grows to 4M before collecting it again.

The default seems to work well here. If you have plenty of memory, it is usually better to use `-H<size>` than to increase `-F<factor>`.

The `-F` setting will be automatically reduced by the garbage collector when the maximum heap size (the `-M<size>` setting) is approaching.

`-G<generations>`:

[Default: 2] Set the number of generations used by the garbage collector. The default of 2 seems to be good, but the garbage collector can support any number of generations. Anything larger than about 4 is probably not a good idea unless your program runs for a *long* time, because the oldest generation will never get collected.

Specifying 1 generation with `+RTS -G1` gives you a simple 2-space collector, as you would expect. In a 2-space collector, the `-A` option (see above) specifies the *minimum* allocation area size, since the allocation area will grow with the amount of live data in the heap. In a multi-generational collector the allocation area is a fixed size (unless you use the `-H` option, see below).

`-H<size>`:

[Default: 0] This option provides a "suggested heap size" for the garbage collector. The garbage collector will use about this much memory until the program residency grows and the heap size needs to be expanded to retain reasonable performance.

By default, the heap will start small, and grow and shrink as necessary. This can be bad for performance, so if you have plenty of memory it's worthwhile supplying a big `-H<size>`. For improving GC performance, using `-H<size>` is usually a better bet than `-A<size>`.

`-k<size>`:

[Default: 1k] Set the initial stack size for new threads. Thread stacks (including the main thread's stack) live on the heap, and grow as required. The default value is good for concurrent applications with lots of small threads; if your program doesn't fit this model then increasing this option may help performance.

The main thread is normally started with a slightly larger heap to cut down on unnecessary stack growth while the program is starting up.

`-K<size>`:

[Default: 1M] Set the maximum stack size for an individual thread to `<size>` bytes. This option is there purely to stop the program eating up all the available memory in the machine if it gets into an infinite loop.

`-m<n>`:

Minimum % `<n>` of heap which must be available for allocation. The default is 3%.

`-M<size>`:

[Default: 64M] Set the maximum heap size to `<size>` bytes. The heap normally grows and shrinks according to the memory requirements of the program. The only reason for having this option is to

stop the heap growing without bound and filling up all the available swap space, which at the least will result in the program being summarily killed by the operating system.

`-s<file>` **or** `-S<file>`**:**

Write modest (`-s`) or verbose (`-S`) garbage-collector statistics into file `<file>`. The default `<file>` is `<program>@.stat`. The `<file>` `stderr` is treated specially, with the output really being sent to `stderr`.

This option is useful for watching how the storage manager adjusts the heap size based on the current amount of live data.

2.12.2 RTS options for profiling and Concurrent/Parallel Haskell

The RTS options related to profiling are described in Section 3.3 (How to control your profiled program at runtime); and those for concurrent/parallel stuff, in Section 2.11.4 (RTS options for Concurrent/Parallel Haskell).

2.12.3 RTS options for hackers, debuggers, and over-interested souls

These RTS options might be used (a) to avoid a GHC bug, (b) to see "what's really happening", or (c) because you feel like it. Not recommended for everyday use!

`-B`**:**

Sound the bell at the start of each (major) garbage collection.

Oddly enough, people really do use this option! Our pal in Durham (England), Paul Callaghan, writes: "Some people here use it for a variety of purposes—honestly!—e.g., confirmation that the code/machine is doing something, infinite loop detection, gauging cost of recently added code. Certain people can even tell what stage [the program] is in by the beep pattern. But the major use is for annoying others in the same office..."

`-r<file>`**:**

Produce "ticky-ticky" statistics at the end of the program run. The `<file>` business works just like on the `-S` RTS option (above).

"Ticky-ticky" statistics are counts of various program actions (updates, enters, etc.) The program must have been compiled using `-ticky` (a.k.a. "ticky-ticky profiling"), and, for it to be really useful, linked with suitable system libraries. Not a trivial undertaking: consult the installation guide on how to set things up for easy "ticky-ticky" profiling. For more information, see Section 3.6 (Using "ticky-ticky" profiling (for implementors)).

`-D<num>`**:**

An RTS debugging flag; varying quantities of output depending on which bits are set in `<num>`. Only works if the RTS was compiled with the `DEBUG` option.

`-Z`**:**

Turn *off* "update-frame squeezing" at garbage-collection time. (There's no particularly good reason to turn it off, except to ensure the accuracy of certain data collected regarding thunk entry counts.)

2.12.4 "Hooks" to change RTS behaviour

GHC lets you exercise rudimentary control over the RTS settings for any given program, by compiling in a "hook" that is called by the run-time system. The RTS contains stub definitions for all these hooks, but by writing your own version and linking it on the GHC command line, you can override the defaults.

The function `defaultsHook` lets you change various RTS options. The commonest use for this is to give your program a default heap and/or stack size that is greater than the default. For example, to set `-H8m -K1m`:

```
#include "Rts.h"
#include "RtsFlags.h"
void defaultsHook (void) {
   RTSflags.GcFlags.stksSize =  1000002 / sizeof(W_);
   RTSflags.GcFlags.heapSize =  8000002 / sizeof(W_);
}
```

Don't use powers of two for heap/stack sizes: these are more likely to interact badly with direct-mapped caches. The full set of flags is defined in `ghc/rts/RtsFlags.h` the the GHC source tree.

You can also change the messages printed when the runtime system "blows up," e.g., on stack overflow. The hooks for these are as follows:

`void ErrorHdrHook (FILE *):`

What's printed out before the message from `error`.

`void OutOfHeapHook (unsigned long, unsigned long):`

The heap-overflow message.

`void StackOverflowHook (long int):`

The stack-overflow message.

`void MallocFailHook (long int):`

The message printed if `malloc` fails.

`void PatErrorHdrHook (FILE *):`

The message printed if a pattern-match fails (the failures that were not handled by the Haskell programmer).

`void PreTraceHook (FILE *):`

What's printed out before a `trace` message.

`void PostTraceHook (FILE *):`

What's printed out after a `trace` message.

For example, here is the "hooks" code used by GHC itself:

```
#include <stdio.h>
#define W_ unsigned long int
#define I_ long int

void
ErrorHdrHook (FILE *where)
{
    fprintf(where, "\n"); /* no "Fail: " */
}

void
OutOfHeapHook (W_ request_size, W_ heap_size) /* both sizes in bytes */
{
    fprintf(stderr, "GHC's heap exhausted;\nwhile trying to
        allocate %lu bytes in a %lu-byte heap;\nuse the `-H<size>'
        option to increase the total heap size.\n",
        request_size,
        heap_size);
}

void
StackOverflowHook (I_ stack_size)    /* in bytes */
{
    fprintf(stderr, "GHC stack-space overflow: current size
        %ld bytes.\nUse the `-K<size>' option to increase it.\n",
        stack_size);
}

void
PatErrorHdrHook (FILE *where)
{
    fprintf(where, "\n*** Pattern-matching error within GHC!\n\n
        This is a compiler bug; please report it to
        glasgow-haskell-bugs@haskell.org.\n\nFail: ");
}

void
PreTraceHook (FILE *where)
{
    fprintf(where, "\n"); /* not "Trace On" */
}

void
PostTraceHook (FILE *where)
{
    fprintf(where, "\n"); /* not "Trace Off" */
}
```

2.13 Debugging the compiler

HACKER TERRITORY. HACKER TERRITORY. (You were warned.)

2.13.1 Replacing the program for one or more phases.

You may specify that a different program be used for one of the phases of the compilation system, in place of whatever the driver `ghc` has wired into it. For example, you might want to try a different assembler. The `-pgm<phase-code><program-name>` option to `ghc` will cause it to use `<program-name>` for phase `<phase-code>`, where the codes to indicate the phases are:

code	phase
L	literate pre-processor
P	C pre-processor (if -cpp only)
C	Haskell compiler
c	C compiler
a	assembler
l	linker
dep	Makefile dependency generator

2.13.2 Forcing options to a particular phase.

The preceding sections describe driver options that are mostly applicable to one particular phase. You may also *force* a specific option `<option>` to be passed to a particular phase `<phase-code>` by feeding the driver the option `-opt<phase-code><option>`. The codes to indicate the phases are the same as in the previous section.

So, for example, to force an `-Ewurble` option to the assembler, you would tell the driver `-opta-Ewurble` (the dash before the E is required).

Besides getting options to the Haskell compiler with `-optC<blah>`, you can get options through to its runtime system with `-optCrts<blah>`.

So, for example: when I want to use my normal driver but with my profiled compiler binary, I use this script:

```
#! /bin/sh
exec /local/grasp_tmp3/simonpj/ghc-BUILDS/working-alpha/ghc/driver/ghc \
     -pgmC/local/grasp_tmp3/simonpj/ghc-BUILDS/working-hsc-prof/hsc \
     -optCrts-i0.5 \
     -optCrts-PT \
     "$@"
```

2.13.3 Dumping out compiler intermediate structures

`-noC`:

Don't bother generating C output *or* an interface file. Usually used in conjunction with one or more of the `-ddump-*` options; for example: `ghc -noC -ddump-simpl Foo.hs`

`-hi:`

Do generate an interface file. This would normally be used in conjunction with `-noC`, which turns off interface generation; thus: `-noC -hi`.

`-dshow-passes:`

Prints a message to stderr as each pass starts. Gives a warm but undoubtedly misleading feeling that GHC is telling you what's happening.

`-ddump-<pass>:`

Make a debugging dump after pass `<pass>` (may be common enough to need a short form...). You can get all of these at once (*lots* of output) by using `-ddump-all`, or most of them with `-ddump-most`. Some of the most useful ones are:

`-ddump-parsed:`
oarser output

`-ddump-rn:`
renamer output

`-ddump-tc:`
typechecker output

`-ddump-deriv:`
derived instances

`-ddump-ds:`
desugarer output

`-ddump-spec:`
output of specialisation pass

`-ddump-rules:`
dumps all rewrite rules (including those generated by the specialisation pass)

`-ddump-simpl:`
simplifer output (Core-to-Core passes)

`-ddump-usagesp:`
UsageSP inference pre-inf and output

`-ddump-cpranal:`
CPR analyser output

`-ddump-stranal:`
strictness analyser output

`-ddump-workwrap:`
worker/wrapper split output

`-ddump-occur-anal:`
'occurrence analysis' output

`-ddump-stg:`
output of STG-to-STG passes

`-ddump-absC`:

*un*flattened Abstract C

`-ddump-flatC`:

flattened Abstract C

`-ddump-realC`:

same as what goes to the C compiler

`-ddump-asm`:

assembly language from the native-code generator

`-dverbose-simpl` **and** `-dverbose-stg`:

Show the output of the intermediate Core-to-Core and STG-to-STG passes, respectively. (*Lots* of output!) So: when we're really desperate:

```
% ghc -noC -O -ddump-simpl -dverbose-simpl -dcore-lint Foo.hs
```

`-ddump-simpl-iterations`:

Show the output of each *iteration* of the simplifier (each run of the simplifier has a maximum number of iterations, normally 4). Used when even `-dverbose-simpl` doesn't cut it.

`-dppr-{user,debug}`:

Debugging output is in one of several "styles." Take the printing of types, for example. In the "user" style, the compiler's internal ideas about types are presented in Haskell source-level syntax, insofar as possible. In the "debug" style (which is the default for debugging output), the types are printed in with explicit foralls, and variables have their unique-id attached (so you can check for things that look the same but aren't).

`-ddump-simpl-stats`:

Dump statistics about how many of each kind of transformation too place. If you add `-dppr-debug` you get more detailed information.

`-ddump-raw-asm`:

Dump out the assembly-language stuff, before the "mangler" gets it.

`-ddump-rn-trace`:

Make the renamer be *real* chatty about what it is upto.

`-dshow-rn-stats`:

Print out summary of what kind of information the renamer had to bring in.

`-dshow-unused-imports`:

Have the renamer report what imports does not contribute.

2.13.4 Checking for consistency

`-dcore-lint`:

Turn on heavyweight intra-pass sanity-checking within GHC, at Core level. (It checks GHC's sanity, not yours.)

`-dstg-lint`:

Ditto for STG level.

`-dusagesp-lint`:

Turn on checks around UsageSP inference (`-fusagesp`). This verifies various simple properties of the results of the inference, and also warns if any identifier with a used-once annotation before the inference has a used-many annotation afterwards; this could indicate a non-worksafe transformation is being applied.

2.13.5 How to read Core syntax (from some `-ddump-*` flags)

Let's do this by commenting an example. It's from doing `-ddump-ds` on this code:

```
skip2 m = m : skip2 (m+2)
```

Before we jump in, a word about names of things. Within GHC, variables, type constructors, etc., are identified by their "Uniques." These are of the form 'letter' plus 'number' (both loosely interpreted). The 'letter' gives some idea of where the Unique came from; e.g., `_` means "built-in type variable"; `t` means "from the typechecker"; `s` means "from the simplifier"; and so on. The 'number' is printed fairly compactly in a 'base-62' format, which everyone hates except me (WDP).

Remember, everything has a "Unique" and it is usually printed out when debugging, in some form or another. So here we go...

```
Desugared:
Main.skip2{-r1L6-} :: _forall_ a$_4 =>{{Num a$_4}} -> a$_4 -> [a$_4]

--# 'r1L6' is the Unique for Main.skip2;
--# '_4' is the Unique for the type-variable (template) 'a'
--# '{{Num a$_4}}' is a dictionary argument

_NI_

--# '_NI_' means "no (pragmatic) information" yet; it will later
--# evolve into the GHC_PRAGMA info that goes into interface files.

Main.skip2{-r1L6-} =
    /\ _4 -> \ d.Num.t4Gt ->
        let {
          {- CoRec -}
          +.t4Hg :: _4 -> _4 -> _4
          _NI_
          +.t4Hg = (+{-r3JH-} _4) d.Num.t4Gt
```

```
          fromInt.t4GS :: Int{-2i-} -> _4
          _NI_
          fromInt.t4GS = (fromInt{-r3JX-} _4) d.Num.t4Gt

--# The '+' class method (Unique: r3JH) selects the addition code
--# from a 'Num' dictionary (now an explicit lamba'd argument).
--# Because Core is 2nd-order lambda-calculus, type applications
--# and lambdas (/\) are explicit.  So '+' is first applied to a
--# type ('_4'), then to a dictionary, yielding the actual addition
--# function that we will use subsequently...

--# We play the exact same game with the (non-standard) class method
--# 'fromInt'.  Unsurprisingly, the type 'Int' is wired into the
--# compiler.

          lit.t4Hb :: _4
          _NI_
          lit.t4Hb =
              let {
                ds.d4Qz :: Int{-2i-}
                _NI_
                ds.d4Qz = I#! 2#
              } in  fromInt.t4GS ds.d4Qz

--# 'I# 2#' is just the literal Int '2'; it reflects the fact that
--# GHC defines 'data Int = I# Int#', where Int# is the primitive
--# unboxed type.  (see relevant info about unboxed types elsewhere...)

--# The '!' after 'I#' indicates that this is a *saturated*
--# application of the 'I#' data constructor (i.e., not partially
--# applied).

          skip2.t3Ja :: _4 -> [_4]
          _NI_
          skip2.t3Ja =
              \ m.r1H4 ->
                  let { ds.d4QQ :: [_4]
                        _NI_
                        ds.d4QQ =
                    let {
                      ds.d4QY :: _4
                      _NI_
                      ds.d4QY = +.t4Hg m.r1H4 lit.t4Hb
                    } in  skip2.t3Ja ds.d4QY
                  } in
                  :! _4 m.r1H4 ds.d4QQ

          {- end CoRec -}
        } in  skip2.t3Ja
```

("It's just a simple functional language" is an unregisterised trademark of Peyton Jones Enterprises, plc.)

2.13.6 Command line options in source files

Sometimes it is useful to make the connection between a source file and the command-line options it requires quite tight. For instance, if a (Glasgow) Haskell source file uses `casms`, the C back-end often needs to be told about which header files to include. Rather than maintaining the list of files the source depends on in a `Makefile` (using the `-#include` command-line option), it is possible to do this directly in the source file using the `OPTIONS` pragma :

```
{-# OPTIONS -#include "foo.h" #-}
module X where

...
```

`OPTIONS` pragmas are only looked for at the top of your source files, upto the first (non-literate,non-empty) line not containing `OPTIONS`. Multiple `OPTIONS` pragmas are recognised. Note that your command shell does not get to the source file options, they are just included literally in the array of command-line arguments the compiler driver maintains internally, so you'll be desperately disappointed if you try to glob etc. inside `OPTIONS`.

NOTE: the contents of OPTIONS are prepended to the command-line options, so you *do* have the ability to override OPTIONS settings via the command line.

It is not recommended to move all the contents of your Makefiles into your source files, but in some circumstances, the `OPTIONS` pragma is the Right Thing. (If you use `-keep-hc-file-too` and have OPTION flags in your module, the OPTIONS will get put into the generated .hc file).

3 Profiling

Glasgow Haskell comes with a time and space profiling system. Its purpose is to help you improve your understanding of your program's execution behaviour, so you can improve it.

Any comments, suggestions and/or improvements you have are welcome. Recommended "profiling tricks" would be especially cool!

3.1 How to profile a Haskell program

The GHC approach to profiling is very simple: annotate the expressions you consider "interesting" with *cost centre* labels (strings); so, for example, you might have:

```
f x y
  = let
        output1 = _scc_ "Pass1" ( pass1 x )
        output2 = _scc_ "Pass2" ( pass2 output1 y )
        output3 = _scc_ "Pass3" ( pass3 (output2 `zip` [1 .. ]) )
    in concat output3
```

The costs of the evaluating the expressions bound to `output1`, `output2` and `output3` will be attributed to the "cost centres" `Pass1`, `Pass2` and `Pass3`, respectively.

The costs of evaluating other expressions, e.g., `concat output4`, will be inherited by the scope which referenced the function `f`.

You can put in cost-centres via `_scc_` constructs by hand, as in the example above. Perfectly cool. That's probably what you *would* do if your program divided into obvious "passes" or "phases", or whatever.

If your program is large or you have no clue what might be gobbling all the time, you can get GHC to mark all functions with `_scc_` constructs, automagically. Add an `-auto` compilation flag to the usual `-prof` option.

Once you start homing in on the Guilty Suspects, you may well switch from automagically-inserted cost-centres to a few well-chosen ones of your own.

To use profiling, you must *compile* and *run* with special options. (We usually forget the "run" magic!—Do as we say, not as we do...) Details follow.

If you're serious about this profiling game, you should probably read one or more of the Sansom/Peyton Jones papers about the GHC profiling system. Just visit the *Glasgow FP group web page* <`http://www.dcs.gla.ac.uk/fp/`>...

3.2 Compiling programs for profiling

To make use of the cost centre profiling system *all* modules must be compiled and linked with the `-prof` option. Any `_scc_` constructs you've put in your source will spring to life.

Without a `-prof` option, your `_scc_`s are ignored; so you can compiled `_scc_`-laden code without changing it.

There are a few other profiling-related compilation options. Use them *in addition to* `-prof`. These do not have to be used consistently for all modules in a program.

`-auto`**:**

GHC will automatically add `_scc_` constructs for all top-level, exported functions.

`-auto-all`**:**

All top-level functions, exported or not, will be automatically `_scc_`'d.

`-caf-all`**:**

The costs of all CAFs in a module are usually attributed to one "big" CAF cost-centre. With this option, all CAFs get their own cost-centre. An "if all else fails" option...

`-ignore-scc`**:**

Ignore any `_scc_` constructs, so a module which already has `_scc_`s can be compiled for profiling with the annotations ignored.

`-G<group>`**:**

Specifies the <`group`> to be attached to all the cost-centres declared in the module. If no group is specified it defaults to the module name.

In addition to the `-prof` option your system might be setup to enable you to compile and link with the `-prof-details` option instead. This enables additional detailed counts to be reported with the `-P` RTS option.

3.3 How to control your profiled program at runtime

It isn't enough to compile your program for profiling with `-prof`!

When you *run* your profiled program, you must tell the runtime system (RTS) what you want to profile (e.g., time and/or space), and how you wish the collected data to be reported. You also may wish to set the sampling interval used in time profiling.

Executive summary: `./a.out +RTS -pT` produces a time profile in `a.out.prof`; `./a.out +RTS -hC` produces space-profiling info which can be mangled by `hp2ps` and viewed with `ghostview` (or equivalent).

Profiling runtime flags are passed to your program between the usual `+RTS` and `-RTS` options.

`-p<sort>` or `-P<sort>`:

The `-p?` option produces a standard *time profile* report. It is written into the file `<program>@.prof`.

The `-P?` option produces a more detailed report containing the actual time and allocation data as well. (Not used much.)

The `<sort>` indicates how the cost centres are to be sorted in the report. Valid `<sort>` options are:

`T`:

by time, largest first (the default);

`A`:

by bytes allocated, largest first;

`C`:

alphabetically by group, module and cost centre.

`-i<secs>`:

Set the profiling (sampling) interval to `<secs>` seconds (the default is 1 second). Fractions are allowed: for example `-i0.2` will get 5 samples per second.

`-h<break-down>`:

Produce a detailed *space profile* of the heap occupied by live closures. The profile is written to the file `<program>@.hp` from which a PostScript graph can be produced using `hp2ps` (see Section 3.5.1 (hp2ps - heap profile to PostScript)).

The heap space profile may be broken down by different criteria:

`-hC`:

cost centre which produced the closure (the default).

`-hM`:

cost centre module which produced the closure.

`-hG`:

cost centre group which produced the closure.

`-hD`:

closure description — a string describing the closure.

`-hY`:

closure type — a string describing the closure's type.

By default all live closures in the heap are profiled, but particular closures of interest can be selected (see below).

Heap (space) profiling uses hash tables. If these tables should fill the run will abort. The `-z<tbl><size>` option is used to increase the size of the relevant hash table (`C`, `M`, `G`, `D` or `Y`, defined as for `<break-down>` above). The actual size used is the next largest power of 2.

The heap profile can be restricted to particular closures of interest. The closures of interest can selected by the attached cost centre (module:label, module and group), closure category (description, type, and kind) using the following options:

`-c{<mod>:<lab>,<mod>:<lab>...}`:

RTS option (profiling)} Selects individual cost centre(s).

`-m{<mod>,<mod>...}`:

RTS option (profiling)} Selects all cost centres from the module(s) specified.

`-g{<grp>,<grp>...}`:

RTS option (profiling)} Selects all cost centres from the groups(s) specified.

`-d{<des>,<des>...}`:

RTS option (profiling)} Selects closures which have one of the specified descriptions.

`-y{<typ>,<typ>...}`:

RTS option (profiling)} Selects closures which have one of the specified type descriptions.

`-k{<knd>,<knd>...}`:

RTS option (profiling)} Selects closures which are of one of the specified closure kinds. Valid closure kinds are `CON` (constructor), `FN` (manifest function), `PAP` (partial application), `BH` (black hole) and `THK` (thunk).

The space occupied by a closure will be reported in the heap profile if the closure satisfies the following logical expression:

([-c] or [-m] or [-g]) and ([-d] or [-y] or [-k])

where a particular option is true if the closure (or its attached cost centre) is selected by the option (or the option is not specified).

3.4 What's in a profiling report?

When you run your profiled program with the `-p` RTS option , you get the following information about your "cost centres":

`COST CENTRE`:

The cost-centre's name.

`MODULE`:

The module associated with the cost-centre; important mostly if you have identically-named cost-centres in different modules.

`scc`:

How many times this cost-centre was entered; think of it as "I got to the `_scc_` construct this many times..."

`%time`:

What part of the time was spent in this cost-centre (see also "ticks," below).

`%alloc`:

What part of the memory allocation was done in this cost-centre (see also "bytes," below).

`inner`:

How many times this cost-centre "passed control" to an inner cost-centre; for example, `scc=4` plus `subscc=8` means "This `_scc_` was entered four times, but went out to other `_scc_s` eight times."

`cafs`:

How many CAFs this cost centre evaluated.

`dicts`:

How many dictionaries this cost centre evaluated.

In addition you can use the `-P` RTS option to get the following additional information:

`ticks`:

The raw number of time "ticks" which were attributed to this cost-centre; from this, we get the `%time` figure mentioned above.

`bytes`:

Number of bytes allocated in the heap while in this cost-centre; again, this is the raw number from which we get the `%alloc` figure mentioned above.

Finally if you built your program with `-prof-details` the `-P` RTS option will also produce the following information:

`closures`:

How many heap objects were allocated; these objects may be of varying size. If you divide the number of bytes (mentioned below) by this number of "closures", then you will get the average object size. (Not too interesting, but still...)

`thunks`:

How many times we entered (evaluated) a thunk—an unevaluated object in the heap—while we were in this cost-centre.

`funcs`:

How many times we entered (evaluated) a function while we we in this cost-centre. (In Haskell, functions are first-class values and may be passed as arguments, returned as results, evaluated, and generally manipulated just like data values)

`PAPs`:

How many times we entered (evaluated) a partial application (PAP), i.e., a function applied to fewer arguments than it needs. For example, `Int` addition applied to one argument would be a PAP. A PAP is really just a particular form for a function.

3.5 Producing graphical heap profiles

Utility programs which produce graphical profiles.

3.5.1 `hp2ps`–heap profile to PostScript

Usage:

```
hp2ps [flags] [<file>[.stat]]
```

The program `hp2ps` converts a heap profile as produced by the `-h<break-down>` runtime option into a PostScript graph of the heap profile. By convention, the file to be processed by `hp2ps` has a `.hp` extension. The PostScript output is written to `<file>@.ps`. If `<file>` is omitted entirely, then the program behaves as a filter.

`hp2ps` is distributed in `ghc/utils/hp2ps` in a GHC source distribution. It was originally developed by Dave Wakeling as part of the HBC/LML heap profiler.

The flags are:

`-d`

In order to make graphs more readable, `hp2ps` sorts the shaded bands for each identifier. The default sort ordering is for the bands with the largest area to be stacked on top of the smaller ones. The `-d` option causes rougher bands (those representing series of values with the largest standard deviations) to be stacked on top of smoother ones.

`-b`

Normally, `hp2ps` puts the title of the graph in a small box at the top of the page. However, if the JOB string is too long to fit in a small box (more than 35 characters), then `hp2ps` will choose to use a big box instead. The `-b` option forces `hp2ps` to use a big box.

`-e<float>[in|mm|pt]`

Generate encapsulated PostScript suitable for inclusion in LaTeX documents. Usually, the PostScript graph is drawn in landscape mode in an area 9 inches wide by 6 inches high, and `hp2ps` arranges for

this area to be approximately centred on a sheet of a4 paper. This format is convenient of studying the graph in detail, but it is unsuitable for inclusion in LaTeX documents. The `-e` option causes the graph to be drawn in portrait mode, with float specifying the width in inches, millimetres or points (the default). The resulting PostScript file conforms to the Encapsulated PostScript (EPS) convention, and it can be included in a LaTeX document using Rokicki's dvi-to-PostScript converter `dvips`.

`-g`

Create output suitable for the `gs` PostScript previewer (or similar). In this case the graph is printed in portrait mode without scaling. The output is unsuitable for a laser printer.

`-l`

Normally a profile is limited to 20 bands with additional identifiers being grouped into an `OTHER` band. The `-l` flag removes this 20 band and limit, producing as many bands as necessary. No key is produced as it won't fit!. It is useful for creation time profiles with many bands.

`-m<int>`

Normally a profile is limited to 20 bands with additional identifiers being grouped into an `OTHER` band. The `-m` flag specifies an alternative band limit (the maximum is 20).

`-m0` requests the band limit to be removed. As many bands as necessary are produced. However no key is produced as it won't fit! It is useful for displaying creation time profiles with many bands.

`-p`

Use previous parameters. By default, the PostScript graph is automatically scaled both horizontally and vertically so that it fills the page. However, when preparing a series of graphs for use in a presentation, it is often useful to draw a new graph using the same scale, shading and ordering as a previous one. The `-p` flag causes the graph to be drawn using the parameters determined by a previous run of `hp2ps` on `file`. These are extracted from `file@.aux`.

`-s`

Use a small box for the title.

`-t<float>`

Normally trace elements which sum to a total of less than 1% of the profile are removed from the profile. The `-t` option allows this percentage to be modified (maximum 5%).

`-t0` requests no trace elements to be removed from the profile, ensuring that all the data will be displayed.

`-?`

Print out usage information.

3.5.2 `stat2resid`—residency info from GC stats

Usage:

```
stat2resid [<file>[.stat] [<outfile>]]
```

The program `stat2resid` converts a detailed garbage collection statistics file produced by the `-S` runtime option into a PostScript heap residency graph. The garbage collection statistics file can be produced without compiling your program for profiling.

By convention, the file to be processed by `stat2resid` has a `.stat` extension. If the `<outfile>` is not specified the PostScript will be written to `<file>@.resid.ps`. If `<file>` is omitted entirely, then the program behaves as a filter.

The plot can not be produced from the statistics file for a generational collector, though a suitable stats file can be produced using the `-F2s` runtime option when the program has been compiled for generational garbage collection (the default).

`stat2resid` is distributed in `ghc/utils/stat2resid` in a GHC source distribution.

3.6 Using "ticky-ticky" profiling (for implementors)

(ToDo: document properly.)

It is possible to compile Glasgow Haskell programs so that they will count lots and lots of interesting things, e.g., number of updates, number of data constructors entered, etc., etc. We call this "ticky-ticky" profiling, because that's the sound a Sun4 makes when it is running up all those counters (*slowly*).

Ticky-ticky profiling is mainly intended for implementors; it is quite separate from the main "cost-centre" profiling system, intended for all users everywhere.

To be able to use ticky-ticky profiling, you will need to have built appropriate libraries and things when you made the system. See "Customising what libraries to build," in the installation guide.

To get your compiled program to spit out the ticky-ticky numbers, use a `-r` RTS option. See Section 2.12 (Running a compiled program).

Compiling your program with the `-ticky` switch yields an executable that performs these counts. Here is a sample ticky-ticky statistics file, generated by the invocation `foo +RTS -rfoo.ticky`.

```
foo +RTS -rfoo.ticky


ALLOCATIONS: 3964631 (11330900 words total: 3999476 admin, 6098829 goods, 1232595 slop)
                                total words:        2     3     4     5    6+
  69647 (  1.8%) function values                 50.0  50.0   0.0   0.0   0.0
2382937 ( 60.1%) thunks                           0.0  83.9  16.1   0.0   0.0
1477218 ( 37.3%) data values                     66.8  33.2   0.0   0.0   0.0
      0 (  0.0%) big tuples
      2 (  0.0%) black holes                      0.0 100.0   0.0   0.0   0.0
      0 (  0.0%) prim things
  34825 (  0.9%) partial applications             0.0   0.0   0.0 100.0   0.0
      2 (  0.0%) thread state objects             0.0   0.0   0.0   0.0 100.0

Total storage-manager allocations: 3647137 (11882004 words)
        [551104 words lost to speculative heap-checks]

STACK USAGE:
```

```
ENTERS: 9400092  of which 2005772 (21.3%) direct to the entry code
                  [the rest indirected via Node's info ptr]
1860318 ( 19.8%) thunks
3733184 ( 39.7%) data values
3149544 ( 33.5%) function values
                  [of which 1999880 (63.5%) bypassed arg-satisfaction chk]
 348140 (  3.7%) partial applications
 308906 (  3.3%) normal indirections
      0 (  0.0%) permanent indirections

RETURNS: 5870443
2137257 ( 36.4%) from entering a new constructor
                  [the rest from entering an existing constructor]
2349219 ( 40.0%) vectored [the rest unvectored]

RET_NEW:         2137257:  32.5% 46.2% 21.3%  0.0%  0.0%  0.0%  0.0%  0.0%  0.0%
RET_OLD:         3733184:   2.8% 67.9% 29.3%  0.0%  0.0%  0.0%  0.0%  0.0%  0.0%
RET_UNBOXED_TUP:       2:   0.0%  0.0%100.0%  0.0%  0.0%  0.0%  0.0%  0.0%  0.0%

RET_VEC_RETURN : 2349219:   0.0%  0.0%100.0%  0.0%  0.0%  0.0%  0.0%  0.0%  0.0%

UPDATE FRAMES: 2241725 (0 omitted from thunks)
SEQ FRAMES:    1
CATCH FRAMES:  1
UPDATES: 2241725
      0 (  0.0%) data values
  34827 (  1.6%) partial applications
                  [2 in place, 34825 allocated new space]
2206898 ( 98.4%) updates to existing heap objects (46 by squeezing)
UPD_CON_IN_NEW:         0:       0      0      0      0      0      0      0      0      0
UPD_PAP_IN_NEW:     34825:       0      0      0  34825      0      0      0      0      0

NEW GEN UPDATES: 2274700 ( 99.9%)

OLD GEN UPDATES: 1852 (  0.1%)

Total bytes copied during GC: 190096

************************************************
3647137 ALLOC_HEAP_ctr
11882004 ALLOC_HEAP_tot
  69647 ALLOC_FUN_ctr
  69647 ALLOC_FUN_adm
  69644 ALLOC_FUN_gds
  34819 ALLOC_FUN_slp
  34831 ALLOC_FUN_hst_0
  34816 ALLOC_FUN_hst_1
      0 ALLOC_FUN_hst_2
      0 ALLOC_FUN_hst_3
      0 ALLOC_FUN_hst_4
2382937 ALLOC_UP_THK_ctr
```

```
      0 ALLOC_SE_THK_ctr
 308906 ENT_IND_ctr
      0 E!NT_PERM_IND_ctr requires +RTS -Z
[... lots more info omitted ...]
      0 GC_SEL_ABANDONED_ctr
      0 GC_SEL_MINOR_ctr
      0 GC_SEL_MAJOR_ctr
      0 GC_FAILED_PROMOTION_ctr
  47524 GC_WORDS_COPIED_ctr
```

The formatting of the information above the row of asterisks is subject to change, but hopefully provides a useful human-readable summary. Below the asterisks *all counters* maintained by the ticky-ticky system are dumped, in a format intended to be machine-readable: zero or more spaces, an integer, a space, the counter name, and a newline.

In fact, not *all* counters are necessarily dumped; compile- or run-time flags can render certain counters invalid. In this case, either the counter will simply not appear, or it will appear with a modified counter name, possibly along with an explanation for the omission (notice `ENT_PERM_IND_ctr` appears with an inserted `!` above). Software analysing this output should always check that it has the counters it expects. Also, beware: some of the counters can have *large* values!

4 Advice on: sooner, faster, smaller, stingier

Please advise us of other "helpful hints" that should go here!

4.1 Sooner: producing a program more quickly

Don't use `-O` or (especially) `-O2`:

By using them, you are telling GHC that you are willing to suffer longer compilation times for better-quality code.

GHC is surprisingly zippy for normal compilations without `-O`!

Use more memory:

Within reason, more memory for heap space means less garbage collection for GHC, which means less compilation time. If you use the `-Rgc-stats` option, you'll get a garbage-collector report. (Again, you can use the cheap-and-nasty `-optCrts-Sstderr` option to send the GC stats straight to standard error.)

If it says you're using more than 20% of total time in garbage collecting, then more memory would help.

If the heap size is approaching the maximum (64M by default), and you have lots of memory, try increasing the maximum with the `-M<size>` option, e.g.: `ghc -c -O -M16m Foo.hs`.

Increasing the default allocation area size used by the compiler's RTS might also help: use the `-A<size>` option.

If GHC persists in being a bad memory citizen, please report it as a bug.

Don't use too much memory!

As soon as GHC plus its "fellow citizens" (other processes on your machine) start using more than the *real memory* on your machine, and the machine starts "thrashing," *the party is over*. Compile times will be worse than terrible! Use something like the csh-builtin `time` command to get a report on how many page faults you're getting.

If you don't know what virtual memory, thrashing, and page faults are, or you don't know the memory configuration of your machine, *don't* try to be clever about memory use: you'll just make your life a misery (and for other people, too, probably).

Try to use local disks when linking:

Because Haskell objects and libraries tend to be large, it can take many real seconds to slurp the bits to/from a remote filesystem.

It would be quite sensible to *compile* on a fast machine using remotely-mounted disks; then *link* on a slow machine that had your disks directly mounted.

Don't derive/use `Read` unnecessarily:

It's ugly and slow.

GHC compiles some program constructs slowly:

Deeply-nested list comprehensions seem to be one such; in the past, very large constant tables were bad, too.

We'd rather you reported such behaviour as a bug, so that we can try to correct it.

The parts of the compiler that seem most prone to wandering off for a long time are the abstract interpreters (strictness and update analysers). You can turn these off individually with `-fno-strictness` and `-fno-update-analysis`.

To figure out which part of the compiler is badly behaved, the `-dshow-passes` option is your friend.

If your module has big wads of constant data, GHC may produce a huge basic block that will cause the native-code generator's register allocator to founder. Bring on `-fvia-C` (not that GCC will be that quick about it, either).

Avoid the consistency-check on linking:

Use `-no-link-chk`; saves effort. This is probably safe in a I-only-compile-things-one-way setup.

Explicit `import` declarations:

Instead of saying `import Foo`, say `import Foo (...stuff I want...)`.

Truthfully, the reduction on compilation time will be very small. However, judicious use of `import` declarations can make a program easier to understand, so it may be a good idea anyway.

4.2 Faster: producing a program that runs quicker

The key tool to use in making your Haskell program run faster are GHC's profiling facilities, described separately in Section 3 (Profiling). There is *no substitute* for finding where your program's time/space is *really* going, as opposed to where you imagine it is going.

Another point to bear in mind: By far the best way to improve a program's performance *dramatically* is to use better algorithms. Once profiling has thrown the spotlight on the guilty time-consumer(s), it may be better to re-think your program than to try all the tweaks listed below.

Another extremely efficient way to make your program snappy is to use library code that has been Seriously Tuned By Someone Else. You *might* be able to write a better quicksort than the one in the HBC library, but it will take you much longer than typing `import QSort`. (Incidentally, it doesn't hurt if the Someone Else is Lennart Augustsson.)

Please report any overly-slow GHC-compiled programs. The current definition of "overly-slow" is "the HBC-compiled version ran faster"...

Optimise, using `-O` or `-O2`:

This is the most basic way to make your program go faster. Compilation time will be slower, especially with `-O2`.

At present, `-O2` is nearly indistinguishable from `-O`.

Compile via C and crank up GCC:

Even with `-O`, GHC tries to use a native-code generator, if available. But the native code-generator is designed to be quick, not mind-bogglingly clever. Better to let GCC have a go, as it tries much harder on register allocation, etc.

So, when we want very fast code, we use: `-O -fvia-C -O2-for-C`.

Overloaded functions are not your friend:

Haskell's overloading (using type classes) is elegant, neat, etc., etc., but it is death to performance if left to linger in an inner loop. How can you squash it?

Give explicit type signatures:

Signatures are the basic trick; putting them on exported, top-level functions is good software-engineering practice, anyway. (Tip: using `-fwarn-missing-signatures` can help enforce good signature-practice).

The automatic specialisation of overloaded functions (with `-O`) should take care of overloaded local and/or unexported functions.

Use `SPECIALIZE` pragmas:

Specialize the overloading on key functions in your program. See Sections 5.10.3 (SPECIALIZE pragmas) and 5.10.4 (SPECIALIZE instance pragmas)

"But how do I know where overloading is creeping in?":

A low-tech way: grep (search) your interface files for overloaded type signatures; e.g.,:

```
% egrep '^[a-z].*::.*=>' *.hi
```

Strict functions are your dear friends:

and, among other things, lazy pattern-matching is your enemy.

(If you don't know what a "strict function" is, please consult a functional-programming textbook. A sentence or two of explanation here probably would not do much good.)

Consider these two code fragments:

```
f (Wibble x y) =  ... # strict

f arg = let { (Wibble x y) = arg } in ... # lazy
```

The former will result in far better code.

A less contrived example shows the use of `cases` instead of `lets` to get stricter code (a good thing):

```
f (Wibble x y)  # beautiful but slow
  = let
        (a1, b1, c1) = unpackFoo x
        (a2, b2, c2) = unpackFoo y
    in ...

f (Wibble x y)  # ugly, and proud of it
  = case (unpackFoo x) of { (a1, b1, c1) ->
    case (unpackFoo y) of { (a2, b2, c2) ->
    ...
    }}
```

GHC loves single-constructor data-types:

It's all the better if a function is strict in a single-constructor type (a type with only one data-constructor; for example, tuples are single-constructor types).

Newtypes are better than datatypes:

If your datatype has a single constructor with a single field, use a `newtype` declaration instead of a `data` declaration. The `newtype` will be optimised away in most cases.

"How do I find out a function's strictness?"

Don't guess—look it up.

Look for your function in the interface file, then for the third field in the pragma; it should say `_S_ <string>`. The `<string>` gives the strictness of the function's arguments. `L` is lazy (bad), `S` and `E` are strict (good), `P` is "primitive" (good), `U(...)` is strict and "unpackable" (very good), and `A` is absent (very good).

For an "unpackable" `U(...)` argument, the info inside tells the strictness of its components. So, if the argument is a pair, and it says `U(AU(LSS))`, that means "the first component of the pair isn't used; the second component is itself unpackable, with three components (lazy in the first, strict in the second \& third)."

If the function isn't exported, just compile with the extra flag `-ddump-simpl`; next to the signature for any binder, it will print the self-same pragmatic information as would be put in an interface file. (Besides, Core syntax is fun to look at!)

Force key functions to be `INLINE`d (esp. monads):

Placing `INLINE` pragmas on certain functions that are used a lot can have a dramatic effect. See Section 5.10.1 (INLINE pragmas).

Explicit `export` list:

If you do not have an explicit export list in a module, GHC must assume that everything in that module will be exported. This has various pessimising effects. For example, if a bit of code is actually

unused (perhaps because of unfolding effects), GHC will not be able to throw it away, because it is exported and some other module may be relying on its existence.

GHC can be quite a bit more aggressive with pieces of code if it knows they are not exported.

Look at the Core syntax!

(The form in which GHC manipulates your code.) Just run your compilation with `-ddump-simpl` (don't forget the `-O`).

If profiling has pointed the finger at particular functions, look at their Core code. `lets` are bad, `cases` are good, dictionaries (`d.<Class>.<Unique>`) [or anything overloading-ish] are bad, nested lambdas are bad, explicit data constructors are good, primitive operations (e.g., `eqInt#`) are good, ...

Use unboxed types (a GHC extension):

When you are *really* desperate for speed, and you want to get right down to the "raw bits." Please see Section 5.1 (Unboxed types) for some information about using unboxed types.

Use `_ccall_s` (a GHC extension) to plug into fast libraries:

This may take real work, but... There exist piles of massively-tuned library code, and the best thing is not to compete with it, but link with it.

Section 5.4 (Calling~C directly from Haskell) says a little about how to use C calls.

Don't use `Floats`:

We don't provide specialisations of Prelude functions for `Float` (but we do for `Double`). If you end up executing overloaded code, you will lose on performance, perhaps badly.

`Floats` (probably 32-bits) are almost always a bad idea, anyway, unless you Really Know What You Are Doing. Use Doubles. There's rarely a speed disadvantage—modern machines will use the same floating-point unit for both. With `Doubles`, you are much less likely to hang yourself with numerical errors.

One time when `Float` might be a good idea is if you have a *lot* of them, say a giant array of `Floats`. They take up half the space in the heap compared to `Doubles`. However, this isn't true on a 64-bit machine.

Use a bigger heap!

If your program's GC stats (`-S` RTS option) indicate that it's doing lots of garbage-collection (say, more than 20% of execution time), more memory might help—with the `-M<size>` or `-A<size>` RTS options (see Section 2.12.1 (RTS options to control the garbage-collector)).

4.3 Smaller: producing a program that is smaller

Decrease the "go-for-it" threshold for unfolding smallish expressions. Give a `-funfolding-use-threshold0` option for the extreme case. ("Only unfoldings with zero cost should proceed.") Warning: except in certain specialiised cases (like Happy parsers) this is likely to actually *increase* the size of your program, because unfolding generally enables extra simplifying optimisations to be performed.

Avoid `Read`.

Use `strip` on your executables.

4.4 Stingier: producing a program that gobbles less heap space

"I think I have a space leak..." Re-run your program with `+RTS -Sstderr`, and remove all doubt! (You'll see the heap usage get bigger and bigger...) [Hmmm... this might be even easier with the `-F2s` RTS option; so... `./a.out +RTS -Sstderr -F2s`...]

Once again, the profiling facilities (Section 3 (Profiling)) are the basic tool for demystifying the space behaviour of your program.

Strict functions are good for space usage, as they are for time, as discussed in the previous section. Strict functions get right down to business, rather than filling up the heap with closures (the system's notes to itself about how to evaluate something, should it eventually be required).

5 GHC Language Features

As with all known Haskell systems, GHC implements some extensions to the language. To use them, you'll need to give a `-fglasgow-exts` option.

Virtually all of the Glasgow extensions serve to give you access to the underlying facilities with which we implement Haskell. Thus, you can get at the Raw Iron, if you are willing to write some non-standard code at a more primitive level. You need not be "stuck" on performance because of the implementation costs of Haskell's "high-level" features—you can always code "under" them. In an extreme case, you can write all your time-critical code in C, and then just glue it together with Haskell!

Executive summary of our extensions:

Unboxed types and primitive operations:

You can get right down to the raw machine types and operations; included in this are "primitive arrays" (direct access to Big Wads of Bytes). Please see Section 5.1 (Unboxed types) and following.

Multi-parameter type classes:

GHC's type system supports extended type classes with multiple parameters. Please see Section 5.5 (Mult-parameter type classes).

Local universal quantification:

GHC's type system supports explicit universal quantification in constructor fields and function arguments. This is useful for things like defining `runST` from the state-thread world. See Section 5.6 (Local universal quantification).

Extistentially quantification in data types:

Some or all of the type variables in a datatype declaration may be *existentially quantified*. More details in Section 5.7 (Existential Quantification).

Scoped type variables:

Scoped type variables enable the programmer to supply type signatures for some nested declarations, where this would not be legal in Haskell 98. Details in Section 5.9 (Scoped Type Variables).

Calling out to C:

Just what it sounds like. We provide *lots* of rope that you can dangle around your neck. Please see Section 5.4 (Calling~C directly from Haskell).

Pragmas

Pragmas are special instructions to the compiler placed in the source file. The pragmas GHC supports are described in Section 5.10 (Pragmas).

Rewrite rules:

The programmer can specify rewrite rules as part of the source program (in a pragma). GHC applies these rewrite rules wherever it can. Details in Section 5.11 (Rewrite Rules).

Before you get too carried away working at the lowest level (e.g., sloshing `MutableByteArray#`s around your program), you may wish to check if there are system libraries that provide a "Haskellised veneer" over the features you want. See Section 6 (GHC Prelude and libraries).

5.1 Unboxed types

These types correspond to the "raw machine" types you would use in C: `Int#` (long int), `Double#` (double), `Addr#` (void *), etc. The *primitive operations* (PrimOps) on these types are what you might expect; e.g., `(+#)` is addition on `Int#`s, and is the machine-addition that we all know and love—usually one instruction.

There are some restrictions on the use of unboxed types, the main one being that you can't pass an unboxed value to a polymorphic function or store one in a polymorphic data type. This rules out things like `[Int#]` (ie. lists of unboxed integers). The reason for this restriction is that polymorphic arguments and constructor fields are assumed to be pointers: if an unboxed integer is stored in one of these, the garbage collector would attempt to follow it, leading to unpredictable space leaks. Or a `seq` operation on the polymorphic component may attempt to dereference the pointer, with disastrous results. Even worse, the unboxed value might be larger than a pointer (`Double#` for instance).

Nevertheless, A numerically-intensive program using unboxed types can go a *lot* faster than its "standard" counterpart—we saw a threefold speedup on one example.

Please see Section 6.4 (The module PrelGHC: really primitive stuff) for the details of unboxed types and the operations on them.

5.2 Primitive state-transformer monad

This monad underlies our implementation of arrays, mutable and immutable, and our implementation of I/O, including "C calls".

The `ST` library, which provides access to the `ST` monad, is a GHC/Hugs extension library and is described in the separate *GHC/Hugs Extension Libraries* document.

5.3 Primitive arrays, mutable and otherwise

GHC knows about quite a few flavours of Large Swathes of Bytes.

First, GHC distinguishes between primitive arrays of (boxed) Haskell objects (type `Array# obj`) and primitive arrays of bytes (type `ByteArray#`).

Second, it distinguishes between...

Immutable:

Arrays that do not change (as with "standard" Haskell arrays); you can only read from them. Obviously, they do not need the care and attention of the state-transformer monad.

Mutable:

Arrays that may be changed or "mutated." All the operations on them live within the state-transformer monad and the updates happen *in-place*.

"Static" (in C land):

A C routine may pass an `Addr#` pointer back into Haskell land. There are then primitive operations with which you may merrily grab values over in C land, by indexing off the "static" pointer.

"Stable" pointers:

If, for some reason, you wish to hand a Haskell pointer (i.e., *not* an unboxed value) to a C routine, you first make the pointer "stable," so that the garbage collector won't forget that it exists. That is, GHC provides a safe way to pass Haskell pointers to C.

Please see Section 5.4.4 (Subverting automatic unboxing with "stable pointers") for more details.

"Foreign objects":

A "foreign object" is a safe way to pass an external object (a C-allocated pointer, say) to Haskell and have Haskell do the Right Thing when it no longer references the object. So, for example, C could pass a large bitmap over to Haskell and say "please free this memory when you're done with it."

Please see Section 5.4.5 (Pointing outside the Haskell heap) for more details.

The libraries section gives more details on all these "primitive array" types and the operations on them, Section 6 (The GHC Prelude and Libraries). Some of these extensions are also supported by Hugs, and the supporting libraries are described in the *GHC/Hugs Extension Libraries* document.

5.4 Calling C directly from Haskell

GOOD ADVICE: Because this stuff is not Entirely Stable as far as names and things go, you would be well-advised to keep your C-callery corraled in a few modules, rather than sprinkled all over your code. It will then be quite easy to update later on.

5.4.1 `_ccall_` and `_casm_`: an introduction

The simplest way to use a simple C function

```
double fooC( FILE *in, char c, int i, double d, unsigned int u )
```

is to provide a Haskell wrapper:

```
fooH :: Char -> Int -> Double -> Word -> IO Double
fooH c i d w = _ccall_ fooC (``stdin''::Addr) c i d w
```

The function `fooH` will unbox all of its arguments, call the C function `fooC` and box the corresponding arguments.

One of the annoyances about `_ccall_`s is when the C types don't quite match the Haskell compiler's ideas. For this, the `_casm_` variant may be just the ticket (NB: *no chance* of such code going through a native-code generator):

```
import Addr
import CString

oldGetEnv name
  = _casm_ ``%r = getenv((char *) %0);'' name >>= \ litstring ->
    return (
        if (litstring == nullAddr) then
            Left ("Fail:oldGetEnv:"++name)
        else
            Right (unpackCString litstring)
    )
```

The first literal-literal argument to a `_casm_` is like a `printf` format: `%r` is replaced with the "result," `%0`–`%n-1` are replaced with the 1st–nth arguments. As you can see above, it is an easy way to do simple C casting. Everything said about `_ccall_` goes for `_casm_` as well.

The use of `_casm_` in your code does pose a problem to the compiler when it comes to generating an interface file for a freshly compiled module. Included in an interface file is the unfolding (if any) of a declaration. However, if a declaration's unfolding happens to contain a `_casm_`, its unfolding will *not* be emitted into the interface file even if it qualifies by all the other criteria. The reason why the compiler prevents this from happening is that unfolding `_casm_`s into an interface file unduly constrains how code that import your module have to be compiled. If an imported declaration is unfolded and it contains a `_casm_`, you now have to be using a compiler backend capable of dealing with it (i.e., the C compiler backend). If you are using the C compiler backend, the unfolded `_casm_` may still cause you problems since the C code snippet it contains may mention CPP symbols that were in scope when compiling the original module are not when compiling the importing module.

If you're willing to put up with the drawbacks of doing cross-module inlining of C code (GHC - A Better C Compiler :-), the option `-funfold-casms-in-hi-file` will turn off the default behaviour.

5.4.2 Literal-literals

The literal-literal argument to `_casm_` can be made use of separately from the `_casm_` construct itself. Indeed, we've already used it:

```
fooH :: Char -> Int -> Double -> Word -> IO Double
fooH c i d w = _ccall_ fooC (``stdin''::Addr) c i d w
```

The first argument that's passed to `fooC` is given as a literal-literal, that is, a literal chunk of C code that will be inserted into the generated `.hc` code at the right place.

A literal-literal is restricted to having a type that's an instance of the `CCallable` class, see 5.4.7 (CCallable) for more information.

Notice that literal-literals are by their very nature unfriendly to native code generators, so exercise judgement about whether or not to make use of them in your code.

5.4.3 Using function headers

When generating C (using the `-fvia-C` directive), one can assist the C compiler in detecting type errors by using the `-#include` directive to provide `.h` files containing function headers.

For example,

```
typedef unsigned long *StgForeignObj;
typedef long StgInt;

void          initialiseEFS (StgInt size);
StgInt        terminateEFS (void);
StgForeignObj emptyEFS(void);
StgForeignObj updateEFS (StgForeignObj a, StgInt i, StgInt x);
StgInt        lookupEFS (StgForeignObj a, StgInt i);
```

You can find appropriate definitions for `StgInt`, `StgForeignObj`, etc using `gcc` on your architecture by consulting `ghc/includes/StgTypes.h`. The following table summarises the relationship between Haskell types and C types.

C type name	Haskell Type
StgChar	Char#
StgInt	Int#
StgWord	Word#
StgAddr	Addr#
StgFloat	Float#
StgDouble	Double#
StgArray	Array#
StgByteArray	ByteArray#
StgArray	MutableArray#
StgByteArray	MutableByteArray#
StgStablePtr	StablePtr#
StgForeignObj	ForeignObj#

Note that this approach is only *essential* for returning `floats` (or if `sizeof(int) != sizeof(int *)` on your architecture) but is a Good Thing for anyone who cares about writing solid code. You're crazy not to do it.

5.4.4 Subverting automatic unboxing with "stable pointers"

The arguments of a `_ccall_` are automatically unboxed before the call. There are two reasons why this is usually the Right Thing to do:

- C is a strict language: it would be excessively tedious to pass unevaluated arguments and require the C programmer to force their evaluation before using them.

- Boxed values are stored on the Haskell heap and may be moved within the heap if a garbage collection occurs—that is, pointers to boxed objects are not *stable*.

It is possible to subvert the unboxing process by creating a "stable pointer" to a value and passing the stable pointer instead. For example, to pass/return an integer lazily to C functions `storeC` and `fetchC`, one might write:

```
storeH :: Int -> IO ()
storeH x = makeStablePtr x              >>= \ stable_x ->
           _ccall_ storeC stable_x

fetchH :: IO Int
fetchH x = _ccall_ fetchC               >>= \ stable_x ->
           deRefStablePtr stable_x      >>= \ x ->
           freeStablePtr stable_x       >>
           return x
```

The garbage collector will refrain from throwing a stable pointer away until you explicitly call one of the following from C or Haskell.

```
void freeStablePointer( StgStablePtr stablePtrToToss )
freeStablePtr :: StablePtr a -> IO ()
```

As with the use of `free` in C programs, GREAT CARE SHOULD BE EXERCISED to ensure these functions are called at the right time: too early and you get dangling references (and, if you're lucky, an error message from the runtime system); too late and you get space leaks.

And to force evaluation of the argument within `fooC`, one would call one of the following C functions (according to type of argument).

```
void     performIO  ( StgStablePtr stableIndex /* StablePtr s (IO ()) */ );
StgInt   enterInt   ( StgStablePtr stableIndex /* StablePtr s Int */ );
StgFloat enterFloat ( StgStablePtr stableIndex /* StablePtr s Float */ );
```

Note Bene: `_ccall_GC_` must be used if any of these functions are used.

5.4.5 Foreign objects: pointing outside the Haskell heap

There are two types that `ghc` programs can use to reference (heap-allocated) objects outside the Haskell world: `Addr` and `ForeignObj`.

If you use `Addr`, it is up to you to the programmer to arrange allocation and deallocation of the objects.

If you use `ForeignObj`, `ghc`'s garbage collector will call upon the user-supplied *finaliser* function to free the object when the Haskell world no longer can access the object. (An object is associated with a finaliser function when the abstract Haskell type `ForeignObj` is created). The finaliser function is expressed in C, and is passed as argument the object:

```
void foreignFinaliser ( StgForeignObj fo )
```

when the Haskell world can no longer access the object. Since `ForeignObj`s only get released when a garbage collection occurs, we provide ways of triggering a garbage collection from within C and from within Haskell.

```
void GarbageCollect()
performGC :: IO ()
```

More information on the programmers' interface to `ForeignObj` can be found in the library documentation.

5.4.6 Avoiding monads

The `_ccall_` construct is part of the `IO` monad because 9 out of 10 uses will be to call imperative functions with side effects such as `printf`. Use of the monad ensures that these operations happen in a predictable order in spite of laziness and compiler optimisations.

To avoid having to be in the monad to call a C function, it is possible to use `unsafePerformIO`, which is available from the `IOExts` module. There are three situations where one might like to call a C function from outside the IO world:

- Calling a function with no side-effects:

```
atan2d :: Double -> Double -> Double
atan2d y x = unsafePerformIO (_ccall_ atan2d y x)

sincosd :: Double -> (Double, Double)
sincosd x = unsafePerformIO $ do
        da <- newDoubleArray (0, 1)
        _casm_ ``sincosd( %0, &((double *)%1[0]), &((double *)%1[1]) );'' x da
        s <- readDoubleArray da 0
        c <- readDoubleArray da 1
        return (s, c)
```

- Calling a set of functions which have side-effects but which can be used in a purely functional manner.

  For example, an imperative implementation of a purely functional lookup-table might be accessed using the following functions.

```
empty  :: EFS x
update :: EFS x -> Int -> x -> EFS x
lookup :: EFS a -> Int -> a

empty = unsafePerformIO (_ccall_ emptyEFS)

update a i x = unsafePerformIO $
        makeStablePtr x         >>= \ stable_x ->
        _ccall_ updateEFS a i stable_x

lookup a i = unsafePerformIO $
        _ccall_ lookupEFS a i   >>= \ stable_x ->
        deRefStablePtr stable_x
```

  You will almost always want to use `ForeignObj`s with this.

- Calling a side-effecting function even though the results will be unpredictable. For example the `trace` function is defined by:

```
trace :: String -> a -> a
trace string expr
  = unsafePerformIO (
        ((_ccall_ PreTraceHook sTDERR{-msg-}):: IO ())  >>
        fputs sTDERR string                             >>
        ((_ccall_ PostTraceHook sTDERR{-msg-}):: IO ()) >>
        return expr )
  where
    sTDERR = (``stderr'' :: Addr)
```

  (This kind of use is not highly recommended — it is only really useful in debugging code.)

5.4.7 C-calling "gotchas" checklist

And some advice, too.

- For modules that use `_ccall_`s, etc., compile with `-fvia-C`. You don't have to, but you should.

  Also, use the `-#include "prototypes.h"` flag (hack) to inform the C compiler of the fully-prototyped types of all the C functions you call. (Section 5.4.3 (Using function headers) says more about this...)

  This scheme is the *only* way that you will get *any* typechecking of your `_ccall_`s. (It shouldn't be that way, but...). GHC will pass the flag `-Wimplicit` to gcc so that you'll get warnings if any `_ccall_`ed functions have no prototypes.

- Try to avoid `_ccall_`s to C functions that take `float` arguments or return `float` results. Reason: if you do, you will become entangled in (ANSI?) C's rules for when arguments/results are promoted to `doubles`. It's a nightmare and just not worth it. Use `doubles` if possible.

  If you do use `floats`, check and re-check that the right thing is happening. Perhaps compile with `-keep-hc-file-too` and look at the intermediate C (`.hc` file).

- The compiler uses two non-standard type-classes when type-checking the arguments and results of `_ccall_`: the arguments (respectively result) of `_ccall_` must be instances of the class `CCallable` (respectively `CReturnable`). Both classes may be imported from the module `CCall`, but this should only be necessary if you want to define a new instance. (Neither class defines any methods — their only function is to keep the type-checker happy.)

  The type checker must be able to figure out just which of the C-callable/returnable types is being used. If it can't, you have to add type signatures. For example,

```
f x = _ccall_ foo x
```

  is not good enough, because the compiler can't work out what type `x` is, nor what type the `_ccall_` returns. You have to write, say:

```
f :: Int -> IO Double
f x = _ccall_ foo x
```

  This table summarises the standard instances of these classes.

Type	CCallable	CReturnable	Which is probably...
`Char`	Yes	Yes	`unsigned char`
`Int`	Yes	Yes	`long int`
`Word`	Yes	Yes	`unsigned long int`
`Addr`	Yes	Yes	`void *`
`Float`	Yes	Yes	`float`
`Double`	Yes	Yes	`double`
`()`	No	Yes	`void`
`[Char]`	Yes	No	`char *` (null-terminated)
`Array`	Yes	No	`unsigned long *`
`ByteArray`	Yes	No	`unsigned long *`
`MutableArray`	Yes	No	`unsigned long *`
`MutableByteArray`	Yes	No	`unsigned long *`
`State`	Yes	Yes	nothing!
`StablePtr`	Yes	Yes	`unsigned long *`
`ForeignObjs`	Yes	Yes	see later

Actually, the `Word` type is defined as being the same size as a pointer on the target architecture, which is *probably* `unsigned long int`.

The brave and careful programmer can add their own instances of these classes for the following types:

- A *boxed-primitive* type may be made an instance of both `CCallable` and `CReturnable`.

  A boxed primitive type is any data type with a single unary constructor with a single primitive argument. For example, the following are all boxed primitive types:

  ```
  Int
  Double
  data XDisplay = XDisplay Addr#
  data EFS a = EFS# ForeignObj#

  instance CCallable   (EFS a)
  instance CReturnable (EFS a)
  ```

- Any datatype with a single nullary constructor may be made an instance of `CReturnable`. For example:

  ```
  data MyVoid = MyVoid
  instance CReturnable MyVoid
  ```

- As at version 2.09, `String` (i.e., `[Char]`) is still not a `CReturnable` type.

  Also, the now-builtin type `PackedString` is neither `CCallable` nor `CReturnable`. (But there are functions in the PackedString interface to let you get at the necessary bits...)

- The code-generator will complain if you attempt to use `%r` in a `_casm_` whose result type is `IO ()`; or if you don't use `%r` *precisely* once for any other result type. These messages are supposed to be helpful and catch bugs—please tell us if they wreck your life.

- If you call out to C code which may trigger the Haskell garbage collector or create new threads (examples of this later...), then you must use the `_ccall_GC_` or `_casm_GC_` variant of C-calls. (This does not work with the native code generator - use `\fvia-C`.) This stuff is hairy with a capital H!

5.5 Multi-parameter type classes

This section documents GHC's implementation of multi-paramter type classes. There's lots of background in the paper *Type classes: exploring the design space* <http://www.dcs.gla.ac.uk/~simonpj/multi.ps.gz> (Simon Peyton Jones, Mark Jones, Erik Meijer).

I'd like to thank people who reported shorcomings in the GHC 3.02 implementation. Our default decisions were all conservative ones, and the experience of these heroic pioneers has given useful concrete examples to support several generalisations. (These appear below as design choices not implemented in 3.02.)

I've discussed these notes with Mark Jones, and I believe that Hugs will migrate towards the same design choices as I outline here. Thanks to him, and to many others who have offered very useful feedback.

5.5.1 Types

There are the following restrictions on the form of a qualified type:

```
forall tv1..tvn (c1, ...,cn) => type
```

(Here, I write the "foralls" explicitly, although the Haskell source language omits them; in Haskell 1.4, all the free type variables of an explicit source-language type signature are universally quantified, except for the class type variables in a class declaration. However, in GHC, you can give the foralls if you want. See Section 5.6 (Explicit universal quantification)).

1. **Each universally quantified type variable `tvi` must be mentioned (i.e. appear free) in `type`.**

   The reason for this is that a value with a type that does not obey this restriction could not be used without introducing ambiguity. Here, for example, is an illegal type:

   ```
   forall a. Eq a => Int
   ```

   When a value with this type was used, the constraint `Eq tv` would be introduced where `tv` is a fresh type variable, and (in the dictionary-translation implementation) the value would be applied to a dictionary for `Eq tv`. The difficulty is that we can never know which instance of `Eq` to use because we never get any more information about `tv`.

2. **Every constraint `ci` must mention at least one of the universally quantified type variables `tvi`.**

   For example, this type is OK because `C a b` mentions the universally quantified type variable `b`:

   ```
   forall a. C a b => burble
   ```

   The next type is illegal because the constraint `Eq b` does not mention `a`:

   ```
   forall a. Eq b => burble
   ```

   The reason for this restriction is milder than the other one. The excluded types are never useful or necessary (because the offending context doesn't need to be witnessed at this point; it can be floated out). Furthermore, floating them out increases sharing. Lastly, excluding them is a conservative choice; it leaves a patch of territory free in case we need it later.

These restrictions apply to all types, whether declared in a type signature or inferred.

Unlike Haskell 1.4, constraints in types do **not** have to be of the form *(class type-variables)*. Thus, these type signatures are perfectly OK

```
f :: Eq (m a) => [m a] -> [m a]
g :: Eq [a] => ...
```

This choice recovers principal types, a property that Haskell 1.4 does not have.

5.5.2 Class declarations

1. **Multi-parameter type classes are permitted**. For example:

   ```
   class Collection c a where
     union :: c a -> c a -> c a
     ...etc..
   ```

2. **The class hierarchy must be acyclic**. However, the definition of "acyclic" involves only the superclass relationships. For example, this is OK:

   ```
   class C a where {
     op :: D b => a -> b -> b
   }

   class C a => D a where { ... }
   ```

   Here, `C` is a superclass of `D`, but it's OK for a class operation `op` of `C` to mention `D`. (It would not be OK for `D` to be a superclass of `C`.)

3. **There are no restrictions on the context in a class declaration (which introduces superclasses), except that the class hierarchy must be acyclic**. So these class declarations are OK:

   ```
   class Functor (m k) => FiniteMap m k where
     ...

   class (Monad m, Monad (t m)) => Transform t m where
     lift :: m a -> (t m) a
   ```

4. **In the signature of a class operation, every constraint must mention at least one type variable that is not a class type variable**.

   Thus:

   ```
   class Collection c a where
     mapC :: Collection c b => (a->b) -> c a -> c b
   ```

   is OK because the constraint `(Collection a b)` mentions `b`, even though it also mentions the class variable `a`. On the other hand:

   ```
   class C a where
     op :: Eq a => (a,b) -> (a,b)
   ```

is not OK because the constraint `(Eq a)` mentions on the class type variable `a`, but not `b`. However, any such example is easily fixed by moving the offending context up to the superclass context:

```
class Eq a => C a where
  op ::(a,b) -> (a,b)
```

A yet more relaxed rule would allow the context of a class-op signature to mention only class type variables. However, that conflicts with Rule 1(b) for types above.

5. **The type of each class operation must mention *all* of the class type variables**. For example:

```
class Coll s a where
  empty  :: s
  insert :: s -> a -> s
```

is not OK, because the type of `empty` doesn't mention `a`. This rule is a consequence of Rule 1(a), above, for types, and has the same motivation.

Sometimes, offending class declarations exhibit misunderstandings. For example, `Coll` might be rewritten

```
class Coll s a where
  empty  :: s a
  insert :: s a -> a -> s a
```

which makes the connection between the type of a collection of `a`'s (namely `(s a)`) and the element type `a`. Occasionally this really doesn't work, in which case you can split the class like this:

```
class CollE s where
  empty  :: s

class CollE s => Coll s a where
  insert :: s -> a -> s
```

5.5.3 Instance declarations

1. **Instance declarations may not overlap**. The two instance declarations

```
instance context1 => C type1 where ...
instance context2 => C type2 where ...
```

"overlap" if `type1` and `type2` unify

However, if you give the command line option `-fallow-overlapping-instances` then two overlapping instance declarations are permitted iff

- EITHER `type1` and `type2` do not unify
- OR `type2` is a substitution instance of `type1` (but not identical to `type1`)
- OR vice versa

Notice that these rules

- make it clear which instance decl to use (pick the most specific one that matches)

- do not mention the contexts `context1`, `context2` Reason: you can pick which instance decl "matches" based on the type.

Regrettably, GHC doesn't guarantee to detect overlapping instance declarations if they appear in different modules. GHC can "see" the instance declarations in the transitive closure of all the modules imported by the one being compiled, so it can "see" all instance decls when it is compiling `Main`. However, it currently chooses not to look at ones that can't possibly be of use in the module currently being compiled, in the interests of efficiency. (Perhaps we should change that decision, at least for `Main`.)

2. **There are no restrictions on the type in an instance *head*, except that at least one must not be a type variable**. The instance "head" is the bit after the "=>" in an instance decl. For example, these are OK:

```
instance C Int a where ...

instance D (Int, Int) where ...

instance E [[a]] where ...
```

Note that instance heads **may** contain repeated type variables. For example, this is OK:

```
instance Stateful (ST s) (MutVar s) where ...
```

The "at least one not a type variable" restriction is to ensure that context reduction terminates: each reduction step removes one type constructor. For example, the following would make the type checker loop if it wasn't excluded:

```
instance C a => C a where ...
```

There are two situations in which the rule is a bit of a pain. First, if one allows overlapping instance declarations then it's quite convenient to have a "default instance" declaration that applies if something more specific does not:

```
instance C a where
  op = ... -- Default
```

Second, sometimes you might want to use the following to get the effect of a "class synonym":

```
class (C1 a, C2 a, C3 a) => C a where { }

instance (C1 a, C2 a, C3 a) => C a where { }
```

This allows you to write shorter signatures:

```
f :: C a => ...
```

instead of

```
f :: (C1 a, C2 a, C3 a) => ...
```

I'm on the lookout for a simple rule that preserves decidability while allowing these idioms. The experimental flag `-fallow-undecidable-instances` lifts this restriction, allowing all the types in an instance head to be type variables.

3. **Unlike Haskell 1.4, instance heads may use type synonyms**. As always, using a type synonym is just shorthand for writing the RHS of the type synonym definition. For example:

```
type Point = (Int,Int)
instance C Point   where ...
instance C [Point] where ...
```

   is legal. However, if you added

```
instance C (Int,Int) where ...
```

   as well, then the compiler will complain about the overlapping (actually, identical) instance declarations. As always, type synonyms must be fully applied. You cannot, for example, write:

```
type P a = [[a]]
instance Monad P where ...
```

   This design decision is independent of all the others, and easily reversed, but it makes sense to me.

4. **The types in an instance-declaration *context* must all be type variables**. Thus

```
instance C a b => Eq (a,b) where ...
```

   is OK, but

```
instance C Int b => Foo b where ...
```

   is not OK. Again, the intent here is to make sure that context reduction terminates.

   Voluminous correspondence on the Haskell mailing list has convinced me that it's worth experimenting with a more liberal rule. If you use the flag `-fallow-undecidable-instances` you can use arbitrary types in an instance context. Termination is ensured by having a fixed-depth recursion stack. If you exceed the stack depth you get a sort of backtrace, and the opportunity to increase the stack depth with `-fcontext-stack`*N*.

5.6 Explicit universal quantification

GHC now allows you to write explicitly quantified types. GHC's syntax for this now agrees with Hugs's, namely:

```
forall a b. (Ord a, Eq  b) => a -> b -> a
```

The context is, of course, optional. You can't use `forall` as a type variable any more!

Haskell type signatures are implicitly quantified. The `forall` allows us to say exactly what this means. For example:

```
g :: b -> b
```

means this:

```
g :: forall b. (b -> b)
```

The two are treated identically.

5.6.1 Universally-quantified data type fields

In a `data` or `newtype` declaration one can quantify the types of the constructor arguments. Here are several examples:

```
data T a = T1 (forall b. b -> b -> b) a

data MonadT m = MkMonad { return :: forall a. a -> m a,
                          bind   :: forall a b. m a -> (a -> m b) -> m b
                        }

newtype Swizzle = MkSwizzle (Ord a => [a] -> [a])
```

The constructors now have so-called *rank 2* polymorphic types, in which there is a for-all in the argument types.:

```
T1 :: forall a. (forall b. b -> b -> b) -> a -> T1 a
MkMonad :: forall m. (forall a. a -> m a)
                  -> (forall a b. m a -> (a -> m b) -> m b)
                  -> MonadT m
MkSwizzle :: (Ord a => [a] -> [a]) -> Swizzle
```

Notice that you don't need to use a `forall` if there's an explicit context. For example in the first argument of the constructor `MkSwizzle`, an implicit "`forall a.`" is prefixed to the argument type. The implicit `forall` quantifies all type variables that are not already in scope, and are mentioned in the type quantified over.

As for type signatures, implicit quantification happens for non-overloaded types too. So if you write this:

```
data T a = MkT (Either a b) (b -> b)
```

it's just as if you had written this:

```
data T a = MkT (forall b. Either a b) (forall b. b -> b)
```

That is, since the type variable `b` isn't in scope, it's implicitly universally quantified. (Arguably, it would be better to *require* explicit quantification on constructor arguments where that is what is wanted. Feedback welcomed.)

5.6.2 Construction

You construct values of types `T1, MonadT, Swizzle` by applying the constructor to suitable values, just as usual. For example,

```
(T1 (\xy->x) 3) :: T Int

(MkSwizzle sort)    :: Swizzle
(MkSwizzle reverse) :: Swizzle

(let r x = Just x
```

```
   b m k = case m of
               Just y -> k y
               Nothing -> Nothing
in
MkMonad r b) :: MonadT Maybe
```

The type of the argument can, as usual, be more general than the type required, as `(MkSwizzle reverse)` shows. (`reverse` does not need the `Ord` constraint.)

5.6.3 Pattern matching

When you use pattern matching, the bound variables may now have polymorphic types. For example:

```
f :: T a -> a -> (a, Char)
f (T1 f k) x = (f k x, f 'c' 'd')

g :: (Ord a, Ord b) => Swizzle -> [a] -> (a -> b) -> [b]
g (MkSwizzle s) xs f = s (map f (s xs))

h :: MonadT m -> [m a] -> m [a]
h m [] = return m []
h m (x:xs) = bind m x           $ \y ->
             bind m (h m xs)    $ \ys ->
             return m (y:ys)
```

In the function `h` we use the record selectors `return` and `bind` to extract the polymorphic bind and return functions from the `MonadT` data structure, rather than using pattern matching.

You cannot pattern-match against an argument that is polymorphic. For example:

```
newtype TIM s a = TIM (ST s (Maybe a))

runTIM :: (forall s. TIM s a) -> Maybe a
runTIM (TIM m) = runST m
```

Here the pattern-match fails, because you can't pattern-match against an argument of type `(forall s. TIM s a)`. Instead you must bind the variable and pattern match in the right hand side:

```
runTIM :: (forall s. TIM s a) -> Maybe a
runTIM tm = case tm of { TIM m -> runST m }
```

The `tm` on the right hand side is (invisibly) instantiated, like any polymorphic value at its occurrence site, and now you can pattern-match against it.

5.6.4 The partial-application restriction

There is really only one way in which data structures with polymorphic components might surprise you: you must not partially apply them. For example, this is illegal:

```
map MkSwizzle [sort, reverse]
```

The restriction is this: *every subexpression of the program must have a type that has no for-alls, except that in a function application (f e1 ... en) the partial applications are not subject to this rule.* The restriction makes type inference feasible.

In the illegal example, the sub-expression `MkSwizzle` has the polymorphic type `(Ord b => [b] -> [b]) -> Swizzle` and is not a sub-expression of an enclosing application. On the other hand, this expression is OK:

```
map (T1 (\a b -> a)) [1,2,3]
```

even though it involves a partial application of `T1`, because the sub-expression `T1 (\a b -> a)` has type `Int -> T Int`.

5.6.5 Type signatures

Once you have data constructors with universally-quantified fields, or constants such as `runST` that have rank-2 types, it isn't long before you discover that you need more! Consider:

```
mkTs f x y = [T1 f x, T1 f y]
```

`mkTs` is a fuction that constructs some values of type `T`, using some pieces passed to it. The trouble is that since `f` is a function argument, Haskell assumes that it is monomorphic, so we'll get a type error when applying `T1` to it. This is a rather silly example, but the problem really bites in practice. Lots of people trip over the fact that you can't make "wrappers functions" for `runST` for exactly the same reason. In short, it is impossible to build abstractions around functions with rank-2 types.

The solution is fairly clear. We provide the ability to give a rank-2 type signature for *ordinary* functions (not only data constructors), thus:

```
mkTs :: (forall b. b -> b -> b) -> a -> [T a]
mkTs f x y = [T1 f x, T1 f y]
```

This type signature tells the compiler to attribute `f` with the polymorphic type `(forall b. b -> b -> b)` when type checking the body of `mkTs`, so now the application of `T1` is fine.

There are two restrictions:

- You can only define a rank 2 type, specified by the following grammar:

  ```
  rank2type ::= [forall tyvars .] [context =>] funty
  funty     ::= ([forall tyvars .] [context =>] ty) -> funty
              | ty
  ty        ::= ...current Haskell monotype syntax...
  ```

  Informally, the universal quantification must all be right at the beginning, or at the top level of a function argument.

- There is a restriction on the definition of a function whose type signature is a rank-2 type: the polymorphic arguments must be matched on the left hand side of the "=" sign. You can't define `mkTs` like this:

```
mkTs :: (forall b. b -> b -> b) -> a -> [T a]
mkTs = \ f x y -> [T1 f x, T1 f y]
```

  The same partial-application rule applies to ordinary functions with rank-2 types as applied to data constructors.

5.7 Existentially quantified data constructors

The idea of using existential quantification in data type declarations was suggested by Laufer (I believe, thought doubtless someone will correct me), and implemented in Hope+. It's been in Lennart Augustsson's `hbc` Haskell compiler for several years, and proved very useful. Here's the idea. Consider the declaration:

```
data Foo = forall a. MkFoo a (a -> Bool)
         | Nil
```

The data type `Foo` has two constructors with types:

```
MkFoo :: forall a. a -> (a -> Bool) -> Foo
Nil   :: Foo
```

Notice that the type variable `a` in the type of `MkFoo` does not appear in the data type itself, which is plain `Foo`. For example, the following expression is fine:

```
[MkFoo 3 even, MkFoo 'c' isUpper] :: [Foo]
```

Here, `(MkFoo 3 even)` packages an integer with a function `even` that maps an integer to `Bool`; and `MkFoo 'c' isUpper` packages a character with a compatible function. These two things are each of type `Foo` and can be put in a list.

What can we do with a value of type `Foo`?. In particular, what happens when we pattern-match on `MkFoo`?

```
f (MkFoo val fn) = ???
```

Since all we know about `val` and `fn` is that they are compatible, the only (useful) thing we can do with them is to apply `fn` to `val` to get a boolean. For example:

```
f :: Foo -> Bool
f (MkFoo val fn) = fn val
```

What this allows us to do is to package heterogenous values together with a bunch of functions that manipulate them, and then treat that collection of packages in a uniform manner. You can express quite a bit of object-oriented-like programming this way.

5.7.1 Why existential?

What has this to do with *existential* quantification? Simply that `MkFoo` has the (nearly) isomorphic type

```
MkFoo :: (exists a . (a, a -> Bool)) -> Foo
```

But Haskell programmers can safely think of the ordinary *universally* quantified type given above, thereby avoiding adding a new existential quantification construct.

5.7.2 Type classes

An easy extension (implemented in `hbc`) is to allow arbitrary contexts before the constructor. For example:

```
data Baz = forall a. Eq a => Baz1 a a
         | forall b. Show b => Baz2 b (b -> b)
```

The two constructors have the types you'd expect:

```
Baz1 :: forall a. Eq a => a -> a -> Baz
Baz2 :: forall b. Show b => b -> (b -> b) -> Baz
```

But when pattern matching on `Baz1` the matched values can be compared for equality, and when pattern matching on `Baz2` the first matched value can be converted to a string (as well as applying the function to it). So this program is legal:

```
f :: Baz -> String
f (Baz1 p q) | p == q    = "Yes"
             | otherwise = "No"
f (Baz1 v fn)            = show (fn v)
```

Operationally, in a dictionary-passing implementation, the constructors `Baz1` and `Baz2` must store the dictionaries for `Eq` and `Show` respectively, and extract it on pattern matching.

Notice the way that the syntax fits smoothly with that used for universal quantification earlier.

5.7.3 Restrictions

There are several restrictions on the ways in which existentially-quantified constructors can be use.

- When pattern matching, each pattern match introduces a new, distinct, type for each existential type variable. These types cannot be unified with any other type, nor can they escape from the scope of the pattern match. For example, these fragments are incorrect:

  ```
  f1 (MkFoo a f) = a
  ```

  Here, the type bound by `MkFoo` "escapes", because `a` is the result of `f1`. One way to see why this is wrong is to ask what type `f1` has:

  ```
  f1 :: Foo -> a             -- Weird!
  ```

What is this "a" in the result type? Clearly we don't mean this:

```
f1 :: forall a. Foo -> a   -- Wrong!
```

The original program is just plain wrong. Here's another sort of error

```
f2 (Baz1 a b) (Baz1 p q) = a==q
```

It's ok to say a==b or p==q, but a==q is wrong because it equates the two distinct types arising from the two Baz1 constructors.

- You can't pattern-match on an existentially quantified constructor in a let or where group of bindings. So this is illegal:

  ```
  f3 x = a==b where { Baz1 a b = x }
  ```

  You can only pattern-match on an existentially-quantified constructor in a case expression or in the patterns of a function definition.

  The reason for this restriction is really an implementation one. Type-checking binding groups is already a nightmare without existentials complicating the picture. Also an existential pattern binding at the top level of a module doesn't make sense, because it's not clear how to prevent the existentially-quantified type "escaping". So for now, there's a simple-to-state restriction. We'll see how annoying it is.

- You can't use existential quantification for newtype declarations. So this is illegal:

  ```
  newtype T = forall a. Ord a => MkT a
  ```

  Reason: a value of type T must be represented as a pair of a dictionary for Ord t and a value of type t. That contradicts the idea that newtype should have no concrete representation. You can get just the same efficiency and effect by using data instead of newtype. If there is no overloading involved, then there is more of a case for allowing an existentially-quantified newtype, because the data because the data version does carry an implementation cost, but single-field existentially quantified constructors aren't much use. So the simple restriction (no existential stuff on newtype) stands, unless there are convincing reasons to change it.

5.8 Assertions

If you want to make use of assertions in your standard Haskell code, you could define a function like the following:

```
assert :: Bool -> a -> a
assert False x = error "assertion failed!"
assert _     x = x
```

which works, but gives you back a less than useful error message – an assertion failed, but which and where?

One way out is to define an extended assert function which also takes a descriptive string to include in the error message and perhaps combine this with the use of a pre-processor which inserts the source location where assert was used.

Ghc offers a helping hand here, doing all of this for you. For every use of assert in the user's source:

```
kelvinToC :: Double -> Double
kelvinToC k = assert (k &gt;= 0.0) (k+273.15)
```

Ghc will rewrite this to also include the source location where the assertion was made,

```
assert pred val ==> assertError "Main.hs|15" pred val
```

The rewrite is only performed by the compiler when it spots applications of `Exception.assert`, so you can still define and use your own versions of `assert`, should you so wish. If not, import `Exception` to make use `assert` in your code.

To have the compiler ignore uses of assert, use the compiler option `-fignore-asserts`. That is, expressions of the form `assert pred e` will be rewritten to `e`.

Assertion failures can be caught, see the documentation for the Hugs/GHC Exception library for information of how.

5.9 Scoped Type Variables

A *pattern type signature* can introduce a *scoped type variable*. For example

```
f (xs::[a]) = ys ++ ys
           where
              ys :: [a]
              ys = reverse xs
```

The pattern `(xs::[a])` includes a type signature for `xs`. This brings the type variable `a` into scope; it scopes over all the patterns and right hand sides for this equation for `f`. In particular, it is in scope at the type signature for `y`.

At ordinary type signatures, such as that for `ys`, any type variables mentioned in the type signature *that are not in scope* are implicitly universally quantified. (If there are no type variables in scope, all type variables mentioned in the signature are universally quantified, which is just as in Haskell 98.) In this case, since `a` is in scope, it is not universally quantified, so the type of `ys` is the same as that of `xs`. In Haskell 98 it is not possible to declare a type for `ys`; a major benefit of scoped type variables is that it becomes possible to do so.

Scoped type variables are implemented in both GHC and Hugs. Where the implementations differ from the specification below, those differences are noted.

So much for the basic idea. Here are the details.

5.9.1 Scope and implicit quantification

- All the type variables mentioned in the patterns for a single function definition equation, that are not already in scope, are brought into scope by the patterns. We describe this set as the *type variables bound by the equation*.
- The type variables thus brought into scope may be mentioned in ordinary type signatures or pattern type signatures anywhere within their scope.

- In ordinary type signatures, any type variable mentioned in the signature that is in scope is *not* universally quantified.
- Ordinary type signatures do not bring any new type variables into scope (except in the type signature itself!). So this is illegal:

```
f :: a -> a
f x = x::a
```

  It's illegal because `a` is not in scope in the body of `f`, so the ordinary signature `x::a` is equivalent to `x::forall a.a`; and that is an incorrect typing.
- There is no implicit universal quantification on pattern type signatures, nor may one write an explicit `forall` type in a pattern type signature. The pattern type signature is a monotype.
- The type variables in the head of a `class` or `instance` declaration scope over the methods defined in the `where` part. For example:

```
class C a where
  op :: [a] -> a

  op xs = let ys::[a]
              ys = reverse xs
          in
          head ys
```

  (Not implemented in Hugs yet, Dec 98).

5.9.2 Polymorphism

- Pattern type signatures are completely orthogonal to ordinary, separate type signatures. The two can be used independently or together. There is no scoping associated with the names of the type variables in a separate type signature.

```
f :: [a] -> [a]
f (xs::[b]) = reverse xs
```

- The function must be polymorphic in the type variables bound by all its equations. Operationally, the type variables bound by one equation must not:
  - Be unified with a type (such as `Int`, or `[a]`).
  - Be unified with a type variable free in the environment.
  - Be unified with each other. (They may unify with the type variables bound by another equation for the same function, of course.)

  For example, the following all fail to type check:

```
f (x::a) (y::b) = [x,y]       -- a unifies with b

g (x::a) = x + 1::Int         -- a unifies with Int
```

```
h x = let k (y::a) = [x,y]     -- a is free in the
      in k x                   -- environment

k (x::a) True    = ...         -- a unifies with Int
k (x::Int) False = ...

w :: [b] -> [b]
w (x::a) = x                   -- a unifies with [b]
```

- The pattern-bound type variable may, however, be constrained by the context of the principal type, thus:

  ```
  f (x::a) (y::a) = x+y*2
  ```

  gets the inferred type: `forall a.  Num a => a -> a -> a`.

5.9.3 Result type signatures

- The result type of a function can be given a signature, thus:

  ```
  f (x::a) :: [a] = [x,x,x]
  ```

  The final "`::  [a]`" after all the patterns gives a signature to the result type. Sometimes this is the only way of naming the type variable you want:

  ```
  f :: Int -> [a] -> [a]
  f n :: ([a] -> [a]) = let g (x::a, y::a) = (y,x)
                        in \xs -> map g (reverse xs `zip` xs)
  ```

Result type signatures are not yet implemented in Hugs.

5.9.4 Pattern signatures on other constructs

- A pattern type signature can be on an arbitrary sub-pattern, not just on a variable:

  ```
  f ((x,y)::(a,b)) = (y,x) :: (b,a)
  ```

- Pattern type signatures, including the result part, can be used in lambda abstractions:

  ```
  (\ (x::a, y) :: a -> x)
  ```

  Type variables bound by these patterns must be polymorphic in the sense defined above. For example:

  ```
  f1 (x::c) = f1 x      -- ok
  f2 = \(x::c) -> f2 x  -- not ok
  ```

  Here, `f1` is OK, but `f2` is not, because `c` gets unified with a type variable free in the environment, in this case, the type of `f2`, which is in the environment when the lambda abstraction is checked.

- Pattern type signatures, including the result part, can be used in `case` expressions:

  ```
  case e of { (x::a, y) :: a -> x }
  ```

The pattern-bound type variables must, as usual, be polymorphic in the following sense: each case alternative, considered as a lambda abstraction, must be polymorphic. Thus this is OK:

```
case (True,False) of { (x::a, y) -> x }
```

Even though the context is that of a pair of booleans, the alternative itself is polymorphic. Of course, it is also OK to say:

```
case (True,False) of { (x::Bool, y) -> x }
```

- To avoid ambiguity, the type after the "`::`" in a result pattern signature on a lambda or `case` must be atomic (i.e. a single token or a parenthesised type of some sort). To see why, consider how one would parse this:

```
\ x :: a -> b -> x
```

- Pattern type signatures that bind new type variables may not be used in pattern bindings at all. So this is illegal:

```
f x = let (y, z::a) = x in ...
```

But these are OK, because they do not bind fresh type variables:

```
f1 x            = let (y, z::Int) = x in ...
f2 (x::(Int,a)) = let (y, z::a)   = x in ...
```

However a single variable is considered a degenerate function binding, rather than a degerate pattern binding, so this is permitted, even though it binds a type variable:

```
f :: (b->b) = \(x::b) -> x
```

Such degnerate function bindings do not fall under the monomorphism restriction. Thus:

```
g :: a -> a -> Bool = \x y. x==y
```

Here g has type `forall a. Eq a => a -> a -> Bool`, just as if g had a separate type signature. Lacking a type signature, g would get a monomorphic type.

5.9.5 Existentials

- Pattern type signatures can bind existential type variables. For example:

```
data T = forall a. MkT [a]

f :: T -> T
f (MkT [t::a]) = MkT t3
                 where
                   t3::[a] = [t,t,t]
```

5.10 Pragmas

GHC supports several pragmas, or instructions to the compiler placed in the source code. Pragmas don't affect the meaning of the program, but they might affect the efficiency of the generated code.

5.10.1 INLINE pragma

GHC (with `-O`, as always) tries to inline (or "unfold") functions/values that are "small enough," thus avoiding the call overhead and possibly exposing other more-wonderful optimisations.

You will probably see these unfoldings (in Core syntax) in your interface files.

Normally, if GHC decides a function is "too expensive" to inline, it will not do so, nor will it export that unfolding for other modules to use.

The sledgehammer you can bring to bear is the `INLINE` pragma, used thusly:

```
key_function :: Int -> String -> (Bool, Double)

#ifdef __GLASGOW_HASKELL__
{-# INLINE key_function #-}
#endif
```

(You don't need to do the C pre-processor carry-on unless you're going to stick the code through HBC—it doesn't like `INLINE` pragmas.)

The major effect of an `INLINE` pragma is to declare a function's "cost" to be very low. The normal unfolding machinery will then be very keen to inline it.

An `INLINE` pragma for a function can be put anywhere its type signature could be put.

`INLINE` pragmas are a particularly good idea for the `then`/`return` (or `bind`/`unit`) functions in a monad. For example, in GHC's own `UniqueSupply` monad code, we have:

```
#ifdef __GLASGOW_HASKELL__
{-# INLINE thenUs #-}
{-# INLINE returnUs #-}
#endif
```

5.10.2 NOINLINE pragma

The `NOINLINE` pragma does exactly what you'd expect: it stops the named function from being inlined by the compiler. You shouldn't ever need to do this, unless you're very cautious about code size.

5.10.3 SPECIALIZE pragma

(UK spelling also accepted.) For key overloaded functions, you can create extra versions (NB: more code space) specialised to particular types. Thus, if you have an overloaded function:

```
hammeredLookup :: Ord key => [(key, value)] -> key -> value
```

If it is heavily used on lists with `Widget` keys, you could specialise it as follows:

```
{-# SPECIALIZE hammeredLookup :: [(Widget, value)] -> Widget -> value #-}
```

To get very fancy, you can also specify a named function to use for the specialised value, by adding `= blah`, as in:

```
{-# SPECIALIZE hammeredLookup :: ...as before... = blah #-}
```

It's *Your Responsibility* to make sure that `blah` really behaves as a specialised version of `hammeredLookup`!!!

NOTE: the `=blah` feature isn't implemented in GHC 4.xx.

An example in which the `= blah` form will Win Big:

```
toDouble :: Real a => a -> Double
toDouble = fromRational . toRational

{-# SPECIALIZE toDouble :: Int -> Double = i2d #-}
i2d (I# i) = D# (int2Double# i) -- uses Glasgow prim-op directly
```

The `i2d` function is virtually one machine instruction; the default conversion—via an intermediate `Rational`—is obscenely expensive by comparison.

By using the US spelling, your `SPECIALIZE` pragma will work with HBC, too. Note that HBC doesn't support the `= blah` form.

A `SPECIALIZE` pragma for a function can be put anywhere its type signature could be put.

5.10.4 SPECIALIZE instance pragma

Same idea, except for instance declarations. For example:

```
instance (Eq a) => Eq (Foo a) where { ... usual stuff ... }

{-# SPECIALIZE instance Eq (Foo [(Int, Bar)] #-}
```

Compatible with HBC, by the way.

5.10.5 LINE pragma

This pragma is similar to C's `#line` pragma, and is mainly for use in automatically generated Haskell code. It lets you specify the line number and filename of the original code; for example

```
{-# LINE 42 "Foo.vhs" #-}
```

if you'd generated the current file from something called `Foo.vhs` and this line corresponds to line 42 in the original. GHC will adjust its error messages to refer to the line/file named in the `LINE` pragma.

5.10.6 RULES pragma

The RULES pragma lets you specify rewrite rules. It is described in Section 5.11 (Rewrite Rules).

5.11 Rewrite rules

The programmer can specify rewrite rules as part of the source program (in a pragma). GHC applies these rewrite rules wherever it can.

Here is an example:

```
{-# RULES
      "map/map"       forall f,g,xs. map f (map g) xs = map (f.g) xs
#-}
```

5.11.1 Syntax

From a syntactic point of view:

- Each rule has a name, enclosed in double quotes.
- There may be zero or more rules in a `RULES` pragma.
- Layout applies in a `RULES` pragma. Currently no new indentation level is set, so you must lay out your rules starting in the same column as the enclosing definitions.
- Each variable mentioned in a rule must either be in scope (e.g. `map`), or bound by the `forall` (e.g. `f`, `g`, `xs`). The variables bound by the `forall` are called the *pattern* variables.
- A pattern variable may optionally have a type signature. If its type is polymorphic, it *must* have a type signature. For example, here is the `foldr/build` rule:

  ```
  "fold/build"  forall k,z,g::forall b. (a->b->b) -> b -> b .
                foldr k z (build g) = g k z
  ```

- The left hand side of a rule must consist of a top-level variable applied to arbitrary expressions. For example, this is *not* OK:

  ```
  "wrong1"   forall e1,e2.  case True of { True -> e1; False -> e2 } = e1
  "wrong2"   forall f.      f True = True
  ```

  In `"wrong1"`, the LHS is not an application; in `"wrong1"`, the LHS has a pattern variable in the head.

- A rule does not need to be in the same module as (any of) the variables it mentions, though of course they need to be in scope.
- Rules are automatically exported from a module, just as instance declarations are.

5.11.2 Semantics

From a semantic point of view:

- Rules are only applied if you use the `-O` flag.
- Rules are regarded as left-to-right rewrite rules. When GHC finds an expression that is a substitution instance of the LHS of a rule, it replaces the expression by the (appropriately-substituted) RHS. By "a substitution instance" we mean that the LHS can be made equal to the expression by substituting for the pattern variables.
- The LHS and RHS of a rule are typechecked, and must have the same type.
- GHC makes absolutely no attempt to verify that the LHS and RHS of a rule have the same meaning. That is undecideable in general, and infeasible in most interesting cases. The responsibility is entirely the programmer's!
- GHC makes no attempt to make sure that the rules are confluent or terminating. For example:

  ```
  "loop"        forall x,y.  f x y = f y x
  ```

  This rule will cause the compiler to go into an infinite loop.
- GHC currently uses a very simple, syntactic, matching algorithm for matching a rule LHS with an expression. It seeks a substitution which makes the LHS and expression syntactically equal modulo: alpha conversion, and (I think) eta conversion. But not beta conversion (that's called higher-order matching).
- GHC keeps trying to apply the rules as it optimises the program. For example, consider:

  ```
  let s = map f
      t = map g
  in
  s (t xs)
  ```

  The expression `s (t xs)` does not match the rule `"map/map"`, but GHC will substitute for `s` and `t`, giving an expression which does match. If `s` or `t` was (a) used more than once, and (b) large or a redex, then it would not be substituted, and the rule would not fire.
- In the earlier phases of compilation, GHC inlines *nothing that appears on the LHS of a rule*, because once you have substituted for something you can't match against it (given the simple minded matching). So if you write the rule

  ```
  "map/map"       forall f,g.  map f . map g = map (f.g)
  ```

  this *won't* match the expression `map f (map g xs)`. It will only match something written with explicit use of ".". Well, not quite. It *will* match the expression

  ```
  wibble f g xs
  ```

  where `wibble` is defined:

  ```
  wibble f g = map f . map g
  ```

  because `wibble` will be inlined (it's small).

  Later on in compilation, GHC starts inlining even things on the LHS of rules, but still leaves the rules enabled. This inlining policy is controlled by the per-simplification-pass flag `-finline-phasen`.

5.11.3 Controlling what's going on

- Use `-fddump-rules` to see what transformation rules GHC is using.
- Use `-fddump-simpl-stats` to see what rules are being fired.
- The defintion of (say) `build` in `PrelBase.lhs` looks llike this:

```
build   :: forall a. (forall b. (a -> b -> b) -> b -> b) -> [a]
{-# INLINE build #-}
build g = g (:) []
```

  Notice the `INLINE`! That prevents `(:)` from being inlined when compiling `PrelBase`, so that an importing module will "see" the `(:)`, and can match it on the LHS of a rule. `INLINE` prevents any inlining happening in the RHS of the `INLINE` thing. I regret the delicacy of this.

- In `ghc/lib/std/PrelBase.lhs` look at the rules for `map` to see how to write rules that will do fusion and yet give an efficient program even if fusion doesn't happen. More rules in `PrelList.lhs`.

5.12 Concurrent and Parallel Haskell

Concurrent and Parallel Haskell are Glasgow extensions to Haskell which let you structure your program as a group of independent 'threads'.

Concurrent and Parallel Haskell have very different purposes.

Concurrent Haskell is for applications which have an inherent structure of interacting, concurrent tasks (i.e. 'threads'). Threads in such programs may be *required*. For example, if a concurrent thread has been spawned to handle a mouse click, it isn't optional—the user wants something done!

A Concurrent Haskell program implies multiple 'threads' running within a single Unix process on a single processor.

You will find at least one paper about Concurrent Haskell hanging off of *Simon Peyton Jones's Web page* `<http://www.dcs.gla.ac.uk/~simonpj/>`.

Parallel Haskell is about *speed*—spawning threads onto multiple processors so that your program will run faster. The 'threads' are always *advisory*—if the runtime system thinks it can get the job done more quickly by sequential execution, then fine.

A Parallel Haskell program implies multiple processes running on multiple processors, under a PVM (Parallel Virtual Machine) framework.

Parallel Haskell is still relatively new; it is more about "research fun" than about "speed." That will change.

Again, check Simon's Web page for publications about Parallel Haskell (including "GUM", the key bits of the runtime system).

Some details about Concurrent and Parallel Haskell follow.

5.12.1 Language features specific to Concurrent Haskell

The `Concurrent` interface (recommended) GHC provides a `Concurrent` module, a common interface to a collection of useful concurrency abstractions, including those mentioned in the "concurrent paper".

Just add the flag `-syslib concurrent` to your GHC command line and put `import Concurrent` into your modules, and away you go. To create a "required thread":

```
forkIO :: IO () -> IO ThreadId
```

where `ThreadId` is an abstract type representing a handle to the newly created thread. Threads may also be killed:

```
killThread :: ThreadId -> IO ()
```

this terminates the given thread. Any work already done by the thread isn't lost: the computation is suspended until required by another thread. The memory used by the thread will be garbage collected if it isn't referenced from anywhere else.

More generally, an arbitrary exception may be raised in any thread for which we have a `ThreadId`, with `raiseInThread`:

```
raiseInThread :: ThreadId -> Exception -> IO ()
```

Actually `killThread` just raises the `ThreadKilled` exception in the target thread, the normal action of which is to just terminate the thread. The target thread will stop whatever it was doing (even if it was blocked on an `MVar` or other computation) and handle the exception.

The `ThreadId` for the current thread can be obtained with `myThreadId`:

```
myThreadId :: IO ThreadId
```

NOTE: if you have a `ThreadId`, you essentially have a pointer to the thread itself. This means the thread itself can't be garbage collected until you drop the `ThreadId`. This misfeature will hopefully be corrected at a later date.

The `Concurrent` interface also provides access to "M-Vars", which are *synchronising variables*.

`MVars` are rendezvous points, mostly for concurrent threads. They begin either empty or full, and any attempt to read an empty `MVar` blocks. When an `MVar` is written, a single blocked thread may be freed. Reading an `MVar` toggles its state from full back to empty. Therefore, any value written to an `MVar` may only be read once. Multiple reads and writes are allowed, but there must be at least one read between any two writes. Interface:

```
newEmptyMVar :: IO (MVar a)
newMVar      :: a -> IO (MVar a)
takeMVar     :: MVar a -> IO a
putMVar      :: MVar a -> a -> IO ()
readMVar     :: MVar a -> IO a
swapMVar     :: MVar a -> a -> IO a
```

A *channel variable* (`CVar`) is a one-element channel, as described in the paper:

```
data CVar a
newCVar :: IO (CVar a)
putCVar :: CVar a -> a -> IO ()
getCVar :: CVar a -> IO a
```

A `Channel` is an unbounded channel:

```
data Chan a
newChan         :: IO (Chan a)
putChan         :: Chan a -> a -> IO ()
getChan         :: Chan a -> IO a
dupChan         :: Chan a -> IO (Chan a)
unGetChan       :: Chan a -> a -> IO ()
getChanContents :: Chan a -> IO [a]
```

General and quantity semaphores:

```
data QSem
newQSem     :: Int   -> IO QSem
waitQSem    :: QSem  -> IO ()
signalQSem  :: QSem  -> IO ()

data QSemN
newQSemN    :: Int   -> IO QSemN
signalQSemN :: QSemN -> Int -> IO ()
waitQSemN   :: QSemN -> Int -> IO ()
```

Merging streams—binary and n-ary:

```
mergeIO  :: [a]   -> [a] -> IO [a]
nmergeIO :: [[a]] -> IO [a]
```

A *Sample variable* (`SampleVar`) is slightly different from a normal `MVar`:

- Reading an empty `SampleVar` causes the reader to block (same as `takeMVar` on empty `MVar`).
- Reading a filled `SampleVar` empties it and returns value. (same as `takeMVar`)
- Writing to an empty `SampleVar` fills it with a value, and potentially, wakes up a blocked reader (same as for `putMVar` on empty `MVar`).
- Writing to a filled `SampleVar` overwrites the current value. (different from `putMVar` on full `MVar`.)

```
type SampleVar a = MVar (Int, MVar a)

emptySampleVar :: SampleVar a -> IO ()
newSampleVar   :: IO (SampleVar a)
readSample     :: SampleVar a -> IO a
writeSample    :: SampleVar a -> a -> IO ()
```

Finally, there are operations to delay a concurrent thread, and to make one wait:

```
threadDelay     :: Int -> IO () -- delay rescheduling for N microseconds
threadWaitRead  :: Int -> IO () -- wait for input on specified file descriptor
threadWaitWrite :: Int -> IO () -- (read and write, respectively).
```

5.12.2 Features specific to Parallel Haskell

The `Parallel` interface (recommended) GHC provides two functions for controlling parallel execution, through the `Parallel` interface:

```
interface Parallel where
infixr 0 `par`
infixr 1 `seq`

par :: a -> b -> b
seq :: a -> b -> b
```

The expression `(x `par` y)` *sparks* the evaluation of `x` (to weak head normal form) and returns `y`. Sparks are queued for execution in FIFO order, but are not executed immediately. At the next heap allocation, the currently executing thread will yield control to the scheduler, and the scheduler will start a new thread (until reaching the active thread limit) for each spark which has not already been evaluated to WHNF.

The expression `(x `seq` y)` evaluates `x` to weak head normal form and then returns `y`. The `seq` primitive can be used to force evaluation of an expression beyond WHNF, or to impose a desired execution sequence for the evaluation of an expression.

For example, consider the following parallel version of our old nemesis, `nfib`:

```
import Parallel

nfib :: Int -> Int
nfib n | n <= 1 = 1
       | otherwise = par n1 (seq n2 (n1 + n2 + 1))
                     where n1 = nfib (n-1)
                           n2 = nfib (n-2)
```

For values of `n` greater than 1, we use `par` to spark a thread to evaluate `nfib (n-1)`, and then we use `seq` to force the parent thread to evaluate `nfib (n-2)` before going on to add together these two subexpressions. In this divide-and-conquer approach, we only spark a new thread for one branch of the computation (leaving the parent to evaluate the other branch). Also, we must use `seq` to ensure that the parent will evaluate `n2` *before* `n1` in the expression `(n1 + n2 + 1)`. It is not sufficient to reorder the expression as `(n2 + n1 + 1)`, because the compiler may not generate code to evaluate the addends from left to right.

Underlying functions and primitives The functions `par` and `seq` are wired into GHC, and unfold into uses of the `par#` and `seq#` primitives, respectively. If you'd like to see this with your very own eyes, just run GHC with the `-ddump-simpl` option. (Anything for a good time...)

Scheduling policy for concurrent/parallel threads Runnable threads are scheduled in round-robin fashion. Context switches are signalled by the generation of new sparks or by the expiry of a virtual timer (the timer interval is configurable with the `-C[<num>]` RTS option). However, a context switch doesn't really happen until the current heap block is full. You can't get any faster context switching than this.

When a context switch occurs, pending sparks which have not already been reduced to weak head normal form are turned into new threads. However, there is a limit to the number of active threads (runnable or

blocked) which are allowed at any given time. This limit can be adjusted with the `-t<num>` RTS option (the default is 32). Once the thread limit is reached, any remaining sparks are deferred until some of the currently active threads are completed.

5.13 Haskell 98 vs. Glasgow Haskell: language non-compliance

This section lists Glasgow Haskell infelicities in its implementation of Haskell 98. See also the "when things go wrong" section (Section 7 (What to do when something goes wrong)) for information about crashes, space leaks, and other undesirable phenomena.

The limitations here are listed in Haskell-Report order (roughly).

5.13.1 Expressions and patterns

Very long `String` constants:

May not go through. If you add a "string gap" every few thousand characters, then the strings can be as long as you like.

Bear in mind that string gaps and the `-cpp` option don't mix very well (see Section 2.9.1 (The C pre-processor)).

Very long literal lists:

These may tickle a "yacc stack overflow" error in the parser. (It depends on the Yacc used to build your parser.)

Single quotes in module names:

It might work, but it's just begging for trouble.

5.13.2 Declarations and bindings

Derived instances for records:

Hmmm.

5.13.3 Module system and interface files

Namespace pollution

Several modules internal to GHC are visible in the standard namespace. All of these modules begin with `Prel`, so the rule is: don't use any modules beginning with `Prel` in your program, or you will be comprehensively screwed.

5.13.4 Numbers, basic types, and built-in classes

Unchecked arithmetic:

Arguably *not* an infelicity, but... Bear in mind that operations on `Int`, `Float`, and `Double` numbers are *unchecked* for overflow, underflow, and other sad occurrences. (note, however that some architectures trap floating-point overflow and loss-of-precision and report a floating-point exception).

Use `Integer`, `Rational`, etc., numeric types if this stuff keeps you awake at night.

Multiply-defined array elements—not checked:

This code fragment *should* elicit a fatal error, but it does not:

```
main = print (array (1,1) [ 1:=2, 1:=3 ])
```

5.13.5 In Prelude support

Arbitrary-sized tuples:

Plain old tuples of arbitrary size *do* work.

HOWEVER: standard instances for tuples (`Eq`, `Ord`, `Bounded`, `Ix` `Read`, and `Show`) are available *only* up to 5-tuples.

These limitations are easily subvertible, so please ask if you get stuck on them.

Unicode character set:

Haskell 98 embraces the Unicode character set, but GHC doesn't handle it. Yet.

6 The GHC prelude and libraries

This document describes GHC's prelude and libraries. The basic story is that of the Haskell 1.4 Report and Libraries document (which we do not reproduce here), but this document describes in addition:

- GHC's additional non-standard libraries and types, such as state transformers, packed strings, foreign objects, stable pointers, and so on.
- GHC's primitive types and operations. The standard Haskell functions are implemented on top of these, and it is sometimes useful to use them directly.
- The organisation of these libraries into directories.
- Short description of programmer interface to the non-standard libraries provided in addition to the standard prelude.

A number of the libraries that provide access to GHC's language extensions are shared by Hugs, and are described in the *GHC/Hugs Extension Libraries* document.

6.1 Prelude extensions

GHC's prelude contains the following non-standard extensions:

`fromInt` method in class `Num`:

It's there. Converts from an `Int` to the type.

`toInt` method in class `Integral`:

Converts from Integral type to an `Int`.

GHC also internally uses a number of modules that begin with the string `Prel`: for this reason, we don't recommend that you use any module names beginning with `Prel` in your own programs. The `Prel` modules are always available: in fact, you can get access to several extensions this way (for some you might need to give the `-fglasgow-exts` flag).

6.2 GHC/Hugs Extension Libraries

The extension libraries provided by both GHC and Hugs are described in the *GHC/Hugs Extension Library Document*

6.3 GHC-only Extension Libraries

If you rely on the implicit `import Prelude` that GHC normally does for you, and if you don't use any weird flags (notably `-fglasgow-exts`), and if you don't import the Glasgow extensions interface, `GlaExts`, then GHC should work *exactly* as the Haskell report says (modulo a few minor issues, see Section 5.13 (Language Non-compliance)).

If you turn on `-fglasgow-exts`, a new world opesn up to you and the compiler will recognise and parse unboxed values properly, and provide access to the various interfaces libraries described here (and piles of other goodies.)

6.3.1 MutableArray

The `MutableArray` interface provide operations for reading and writing values to mutable arrays. There's two kinds of mutable arrays, the mutatable version of Haskell `Array`s and *mutable byte arrays*, chunks of memory containing values of some basic type.

Mutable arrays The mutable array section of the API provides the following operations:

```
-- mutable arrays:
newArray      :: Ix ix -> (ix,ix) -> elt -> ST s (MutableArray s ix elt)
boundsOfArray :: Ix ix => MutableArray s ix elt -> (ix, ix)
readArray     :: Ix ix => MutableArray s ix elt -> ix -> ST s elt
writeArray    :: Ix ix => MutableArray s ix elt -> ix -> elt -> ST s ()
freezeArray   :: Ix ix => MutableArray s ix elt -> ST s (Array ix elt)
thawArray     :: Ix ix => Array ix elt -> ST s (MutableArray s ix elt)

unsafeFreezeArray   :: Ix ix => MutableArray s ix elt -> ST s (Array ix elt)
unsafeThawArray     :: Ix ix => Array ix elt -> ST s (MutableArray s ix elt)
```

Remarks:

- The `freezeArray` action converts a mutable array into an immutable one by copying, whereas `unsafeFreezeArray` returns an immutable array that is effectively just the type cast version of the mutable array. Should you write to the mutable array after it has been (unsafely) frozen, you'll side-effect the immutable array in the process. Please don't :-)

- The operation `thawArray` goes the other way, converting an immutable `Array` into a mutable one. This is done by copying. The operation `unsafeThawArray` is also provided, which places the same kind of proof obligation on the programmer as `unsafeFreezeArray` does.

Mutable byte arrays

```
-- creators:
newCharArray      :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newAddrArray      :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newIntArray       :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newWordArray      :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newFloatArray     :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newDoubleArray    :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)
newStablePtrArray :: Ix ix => (ix,ix) -> ST s (MutableByteArray s ix)

boundsOfMutableByteArray
                  :: Ix ix => MutableByteArray s ix -> (ix, ix)

readCharArray      :: Ix ix => MutableByteArray s ix -> ix -> ST s Char
readIntArray       :: Ix ix => MutableByteArray s ix -> ix -> ST s Int
readAddrArray      :: Ix ix => MutableByteArray s ix -> ix -> ST s Addr
readFloatArray     :: Ix ix => MutableByteArray s ix -> ix -> ST s Float
readDoubleArray    :: Ix ix => MutableByteArray s ix -> ix -> ST s Double
readStablePtrArray :: Ix ix => MutableByteArray s ix -> ix -> ST s (StablePtr a)
readWord8Array     :: Ix ix => MutableByteArray s ix -> ix -> ST s Word8
readWord16Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Word16
readWord32Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Word32
readWord64Array    :: Ix ix => MutableByteArray s ix -> ix -> ST s Word64
readInt8Array      :: Ix ix => MutableByteArray s ix -> ix -> ST s Int8
readInt16Array     :: Ix ix => MutableByteArray s ix -> ix -> ST s Int16
readInt32Array     :: Ix ix => MutableByteArray s ix -> ix -> ST s Int32
readInt64Array     :: Ix ix => MutableByteArray s ix -> ix -> ST s Int64

writeCharArray        :: Ix ix => MutableByteArray s ix -> ix -> Char -> ST s ()
writeIntArray         :: Ix ix => MutableByteArray s ix -> ix -> Int  -> ST s ()
writeAddrArray        :: Ix ix => MutableByteArray s ix -> ix -> Addr -> ST s ()
writeFloatArray       :: Ix ix => MutableByteArray s ix -> ix -> Float -> ST s ()
writeDoubleArray      :: Ix ix => MutableByteArray s ix -> ix -> Double -> ST s ()
writeStablePtrArray   :: Ix ix => MutableByteArray s ix -> ix -> StablePtr a -> ST s ()
writeWord8Array       :: Ix ix => MutableByteArray s ix -> ix -> Word8  -> ST s ()
writeWord16Array      :: Ix ix => MutableByteArray s ix -> ix -> Word16 -> ST s ()
writeWord32Array      :: Ix ix => MutableByteArray s ix -> ix -> Word32 -> ST s ()
writeWord64Array      :: Ix ix => MutableByteArray s ix -> ix -> Word64 -> ST s ()
writeInt8Array        :: Ix ix => MutableByteArray s ix -> ix -> Int8  -> ST s ()
writeInt16Array       :: Ix ix => MutableByteArray s ix -> ix -> Int16 -> ST s ()
writeInt32Array       :: Ix ix => MutableByteArray s ix -> ix -> Int32 -> ST s ()
writeInt64Array       :: Ix ix => MutableByteArray s ix -> ix -> Int64 -> ST s ()

freezeCharArray       :: Ix ix => MutableByteArray s ix -> ST s (ByteArray ix)
freezeIntArray        :: Ix ix => MutableByteArray s ix -> ST s (ByteArray ix)
freezeAddrArray       :: Ix ix => MutableByteArray s ix -> ST s (ByteArray ix)
```

```
freezeFloatArray      :: Ix ix => MutableByteArray s ix -> ST s (ByteArray ix)
freezeDoubleArray     :: Ix ix => MutableByteArray s ix -> ST s (ByteArray ix)
freezeStablePtrArray  :: Ix ix => MutableByteArray s ix -> ST s (ByteArray ix)

unsafeFreezeByteArray  :: Ix ix => MutableByteArray s ix -> ST s (ByteArray ix)

sizeofMutableByteArray :: Ix ix => MutableByteArray s ix -> Int

thawByteArray       :: Ix ix => ByteArray ixt -> ST s (MutableByteArray s ix)
unsafeThawByteArray :: Ix ix => ByteArray ixt -> ST s (MutableByteArray s ix)
```

Remarks:

- A Mutable byte array is created by specifying its size in units of some basic type. For example,

```
mkPair :: ST s (MutableByteArray s Int)
mkPair = newIntArray (0,1)
```

  creates a mutable array capable of storing two `Int`s. Notice that the range size *is not in bytes*, but in units of the basic type.

- A mutable byte array is not parameterised over the kind of values it contains. A consequence of this is that it is possible to have byte arrays containing a mix of basic types, or even read a value from the array at a different type from which it was written, e.g.,

```
isLitteEndian :: IO Bool
isLitteEndian = stToIO $ do
   x <- newIntArray (0,1)
   writeIntArray x 1
   v <- readCharArray x 0
   return (v == chr 1)
```

  It's left as an exercise for the reader to determine whether having byte arrays not be parameterised over the type of values they contain is a bug or a feature..

- As for mutable arrays, operations for turning mutable byte arrays into immutable byte arrays are also provided by the `freeze*` class of actions. There's also the non-copying `unsafeFreezeByteArray`.

- Operations for going the other way, where an immutable byte array is 'thawed' are also provided. `thawByteArray` does this by copying, whereas `unsafeThawByteArray` does not

- The operation `sizeofMutableByteArray` returns the size of the array, *in bytes.*

6.3.2 The ByteArray interface

`ByteArray`s are chunks of immutable Haskell heap:

```
data ByteArray ix -- abstract
                  -- instance of: Eq, CCallable.

newByteArray       :: Ix ix => (ix,ix) -> ST s (ByteArray ix)

indexCharArray     :: Ix ix => ByteArray ix -> ix -> Char
indexIntArray      :: Ix ix => ByteArray ix -> ix -> Int
```

```
indexAddrArray     :: Ix ix => ByteArray ix -> ix -> Addr
indexFloatArray    :: Ix ix => ByteArray ix -> ix -> Float
indexDoubleArray   :: Ix ix => ByteArray ix -> ix -> Double

sizeofByteArray    :: Ix ix => ByteArray ix -> Int
```

Remarks:

- The operation `newByteArray` creates a byte array of length equal to the range of its indices *in bytes*.
- `sizeofByteArray` returns the size of the byte array, *in bytes*.
- Equality on byte arrays is value equality, not pointer equality (as is the case for its mutable variant.) Two byte arrays are equal if they're of the same length and they're pairwise equal.

6.3.3 The `CCall` module

The `CCall` module defines the classes `CCallable` and `CReturnable`, along with instances for the primitive types (`Int`, `Int#`, `Float`, `Float#` etc.) GHC knows to import this module if you use `_ccall_`, but if you need to define your own instances of these classes, you will need to import `CCall` explicitly.

More information on how to use `_ccall_` can be found in Section 5.4 (Calling~C directly from Haskell).

6.3.4 The `GlaExts` interface

The `GlaExts` interface provides access to extensions that only GHC implements. These currently are: unboxed types, including the representations of the primitive types (Int, Float, etc.), and the GHC primitive operations (+#, ==#, etc.).

This module used to provide access to all the Glasgow extensions, but these have since been moved into separate libraries for compatibility with Hugs (version 2.09: in fact, you can still get at this stuff via `GlaExts` for compatibility, but this facility will likely be removed in the future).

```
-- the representation of some basic types:
data Char    = C# Char#
data Int     = I# Int#
data Addr    = A# Addr#
data Word    = W# Word#
data Float   = F# Float#
data Double  = D# Double#
data Integer = J# Int# Int# ByteArray#

module GHC  -- all primops and primitive types.
```

6.4 The module `PrelGHC`: really primitive stuff

This module defines all the types which are primitive in Glasgow Haskell, and the operations provided for them.

A primitive type is one which cannot be defined in Haskell, and which is therefore built into the language and compiler. Primitive types are always unlifted; that is, a value of primitive type cannot be bottom. We use the convention that primitive types, values, and operations have a `#` suffix.

Primitive values are often represented by a simple bit-pattern, such as `Int#`, `Float#`, `Double#`. But this is not necessarily the case: a primitive value might be represented by a pointer to a heap-allocated object. Examples include `Array#`, the type of primitive arrays. A primitive array is heap-allocated because it is too big a value to fit in a register, and would be too expensive to copy around; in a sense, it is accidental that it is represented by a pointer. If a pointer represents a primitive value, then it really does point to that value: no unevaluated thunks, no indirections...nothing can be at the other end of the pointer than the primitive value.

6.4.1 Unboxed Tuples

Unboxed tuples aren't really exported by `PrelGHC`, they're available by default with `-fglasgow-exts`. An unboxed tuple looks like this:

```
(# e_1, ..., e_n #)
```

where `e_1..e_n` are expressions of any type (primitive or non-primitive). The type of an unboxed tuple looks the same.

Unboxed tuples are used for functions that need to return multiple values, but they avoid the heap allocation normally associated with using fully-fledged tuples. When an unboxed tuple is returned, the components are put directly into registers or on the stack; the unboxed tuple itself does not have a composite representation. Many of the primitive operations listed in this section return unboxed tuples.

There are some pretty stringent restrictions on the use of unboxed tuples:

- Unboxed tuple types are subject to the same restrictions as other unboxed types; i.e. they may not be stored in polymorphic data structures or passed to polymorphic functions.
- Unboxed tuples may only be constructed as the direct result of a function, and may only be deconstructed with a `case` expression. eg. the following are valid:

  ```
  f x y = (# x+1, y-1 #)
  g x = case f x x of { (# a, b #) -> a + b }
  ```

  but the following are invalid:

  ```
  f x y = g (# x, y #)
  g (# x, y #) = x + y
  ```

- No variable can have an unboxed tuple type. This is illegal:

  ```
  f :: (# Int, Int #) -> (# Int, Int #)
  f x = x
  ```

  because `x` has an unboxed tuple type.

Note: we may relax some of these restrictions in the future.

The `IO` and `ST` monads use unboxed tuples to avoid unnecessary allocation during sequences of operations.

6.4.2 Character and numeric types

There are the following obvious primitive types:

```
type Char#
type Int#       -- see also Word# and Addr#, later
type Float#
type Double#
```

If you really want to know their exact equivalents in C, see `ghc/includes/StgTypes.h` in the GHC source tree.

Literals for these types may be written as follows:

```
1#              an Int#
1.2#            a Float#
1.34##          a Double#
'a'#            a Char#; for weird characters, use '\o<octal>'#
"a"#            an Addr# (a `char *')
```

6.4.3 Comparison operations

```
{>,>=,==,/=,<,<=}# :: Int# -> Int# -> Bool

{gt,ge,eq,ne,lt,le}Char# :: Char# -> Char# -> Bool
    -- ditto for Word# and Addr#
```

6.4.4 Primitive-character operations

```
ord# :: Char# -> Int#
chr# :: Int# -> Char#
```

6.4.5 Primitive-`Int` operations

```
{+,-,*,quotInt,remInt}# :: Int# -> Int# -> Int#
negateInt# :: Int# -> Int#

iShiftL#, iShiftRA#, iShiftRL# :: Int# -> Int# -> Int#
        -- shift left, right arithmetic, right logical
```

Note: No error/overflow checking!

6.4.6 Primitive-`Double` and `Float` operations

```
{+,-,*,/}##         :: Double# -> Double# -> Double#
{<,<=,==,/=,>=,>}## :: Double# -> Double# -> Bool
```

```
negateDouble#       :: Double# -> Double#
double2Int#         :: Double# -> Int#
int2Double#         :: Int#    -> Double#

{plus,minux,times,divide}Float# :: Float# -> Float# -> Float#
{gt,ge,eq,ne,lt,le}Float# :: Float# -> Float# -> Bool
negateFloat#        :: Float# -> Float#
float2Int#          :: Float# -> Int#
int2Float#          :: Int#   -> Float#
```

And a full complement of trigonometric functions:

```
expDouble#      :: Double# -> Double#
logDouble#      :: Double# -> Double#
sqrtDouble#     :: Double# -> Double#
sinDouble#      :: Double# -> Double#
cosDouble#      :: Double# -> Double#
tanDouble#      :: Double# -> Double#
asinDouble#     :: Double# -> Double#
acosDouble#     :: Double# -> Double#
atanDouble#     :: Double# -> Double#
sinhDouble#     :: Double# -> Double#
coshDouble#     :: Double# -> Double#
tanhDouble#     :: Double# -> Double#
powerDouble#    :: Double# -> Double# -> Double#
```

similarly for `Float#`.

There are two coercion functions for `Float#`/`Double#`:

```
float2Double#   :: Float# -> Double#
double2Float#   :: Double# -> Float#
```

The primitive versions of `encodeDouble`/`decodeDouble`:

```
encodeDouble#   :: Int# -> Int# -> ByteArray#   -- Integer mantissa
                -> Int#                         -- Int exponent
                -> Double#

decodeDouble#   :: Double# -> PrelNum.ReturnIntAndGMP
```

(And the same for `Float#`s.)

6.4.7 Operations on/for `Integers` (interface to GMP)

We implement `Integers` (arbitrary-precision integers) using the GNU multiple-precision (GMP) package (version 2.0.2).

The data type for `Integer` must mirror that for `MP_INT` in `gmp.h` (see `gmp.info` in `ghc/includes/runtime/gmp`). It comes out as:

```
data Integer = J# Int# Int# ByteArray#
```

So, `Integer` is really just a "pairing" type for a particular collection of primitive types.

The primitive ops to support `Integers` use the "pieces" of the representation, and are as follows:

```
negateInteger#  :: Int# -> Int# -> ByteArray# -> Integer

{plus,minus,times}Integer# :: Int# -> Int# -> ByteArray#
                           -> Int# -> Int# -> ByteArray#
                           -> Integer

cmpInteger# :: Int# -> Int# -> ByteArray#
            -> Int# -> Int# -> ByteArray#
            -> Int# -- -1 for <; 0 for ==; +1 for >

divModInteger#, quotRemInteger#
        :: Int# -> Int# -> ByteArray#
        -> Int# -> Int# -> ByteArray#
        -> PrelNum.Return2GMPs

integer2Int# :: Int# -> Int# -> ByteArray# -> Int#

int2Integer#  :: Int#  -> Integer -- NB: no error-checking on these two!
word2Integer# :: Word# -> Integer

addr2Integer# :: Addr# -> Integer
        -- the Addr# is taken to be a 'char *' string
        -- to be converted into an Integer.
```

6.4.8 Words and addresses

A `Word#` is used for bit-twiddling operations. It is the same size as an `Int#`, but has no sign nor any arithmetic operations.

```
type Word#      -- Same size/etc as Int# but *unsigned*
type Addr#      -- A pointer from outside the "Haskell world" (from C, probably);
                -- described under "arrays"
```

`Word#`s and `Addr#`s have the usual comparison operations. Other unboxed-`Word` ops (bit-twiddling and coercions):

```
and#, or#, xor# :: Word# -> Word# -> Word#
        -- standard bit ops.

quotWord#, remWord# :: Word# -> Word# -> Word#
        -- word (i.e. unsigned) versions are different from int
        -- versions, so we have to provide these explicitly.
```

```
not# :: Word# -> Word#

shiftL#, shiftRA#, shiftRL# :: Word# -> Int# -> Word#
        -- shift left, right arithmetic, right logical

int2Word#       :: Int#  -> Word# -- just a cast, really
word2Int#       :: Word# -> Int#
```

Unboxed-`Addr` ops (C casts, really):

```
int2Addr#       :: Int#  -> Addr#
addr2Int#       :: Addr# -> Int#
```

The casts between `Int#`, `Word#` and `Addr#` correspond to null operations at the machine level, but are required to keep the Haskell type checker happy.

Operations for indexing off of C pointers (`Addr#`s) to snatch values are listed under "arrays".

6.4.9 Arrays

The type `Array# elt` is the type of primitive, unpointed arrays of values of type `elt`.

```
type Array# elt
```

`Array#` is more primitive than a Haskell array — indeed, the Haskell `Array` interface is implemented using `Array#` — in that an `Array#` is indexed only by `Int#`s, starting at zero. It is also more primitive by virtue of being unboxed. That doesn't mean that it isn't a heap-allocated object - of course, it is. Rather, being unboxed means that it is represented by a pointer to the array itself, and not to a thunk which will evaluate to the array (or to bottom). The components of an `Array#` are themselves boxed.

The type `ByteArray#` is similar to `Array#`, except that it contains just a string of (non-pointer) bytes.

```
type ByteArray#
```

Arrays of these types are useful when a Haskell program wishes to construct a value to pass to a C procedure. It is also possible to use them to build (say) arrays of unboxed characters for internal use in a Haskell program. Given these uses, `ByteArray#` is deliberately a bit vague about the type of its components. Operations are provided to extract values of type `Char#`, `Int#`, `Float#`, `Double#`, and `Addr#` from arbitrary offsets within a `ByteArray#`. (For type `Foo#`, the ith offset gets you the ith `Foo#`, not the `Foo#` at byte-position i. Mumble.) (If you want a `Word#`, grab an `Int#`, then coerce it.)

Lastly, we have static byte-arrays, of type `Addr#` [mentioned previously]. (Remember the duality between arrays and pointers in C.) Arrays of this types are represented by a pointer to an array in the world outside Haskell, so this pointer is not followed by the garbage collector. In other respects they are just like `ByteArray#`. They are only needed in order to pass values from C to Haskell.

6.4.10 Reading and writing

Primitive arrays are linear, and indexed starting at zero.

The size and indices of a `ByteArray#`, `Addr#`, and `MutableByteArray#` are all in bytes. It's up to the program to calculate the correct byte offset from the start of the array. This allows a `ByteArray#` to contain a mixture of values of different type, which is often needed when preparing data for and unpicking results from C. (Umm... not true of indices... WDP 95/09)

Should we provide some `sizeOfDouble#` constants?

Out-of-range errors on indexing should be caught by the code which uses the primitive operation; the primitive operations themselves do *not* check for out-of-range indexes. The intention is that the primitive ops compile to one machine instruction or thereabouts.

We use the terms "reading" and "writing" to refer to accessing *mutable* arrays (see Section 6.4.13 (Mutable arrays)), and "indexing" to refer to reading a value from an *immutable* array.

Immutable byte arrays are straightforward to index (all indices in bytes):

```
indexCharArray#   :: ByteArray# -> Int# -> Char#
indexIntArray#    :: ByteArray# -> Int# -> Int#
indexAddrArray#   :: ByteArray# -> Int# -> Addr#
indexFloatArray#  :: ByteArray# -> Int# -> Float#
indexDoubleArray# :: ByteArray# -> Int# -> Double#

indexCharOffAddr#   :: Addr# -> Int# -> Char#
indexIntOffAddr#    :: Addr# -> Int# -> Int#
indexFloatOffAddr#  :: Addr# -> Int# -> Float#
indexDoubleOffAddr# :: Addr# -> Int# -> Double#
indexAddrOffAddr#   :: Addr# -> Int# -> Addr#
 -- Get an Addr# from an Addr# offset
```

The last of these, `indexAddrOffAddr#`, extracts an `Addr#` using an offset from another `Addr#`, thereby providing the ability to follow a chain of C pointers.

Something a bit more interesting goes on when indexing arrays of boxed objects, because the result is simply the boxed object. So presumably it should be entered — we never usually return an unevaluated object! This is a pain: primitive ops aren't supposed to do complicated things like enter objects. The current solution is to return a single element unboxed tuple (see Section 6.4.1 (Unboxed Tuples)).

```
indexArray#       :: Array# elt -> Int# -> (# elt #)
```

6.4.11 The state type

The primitive type `State#` represents the state of a state transformer. It is parameterised on the desired type of state, which serves to keep states from distinct threads distinct from one another. But the *only* effect of this parameterisation is in the type system: all values of type `State#` are represented in the same way. Indeed, they are all represented by nothing at all! The code generator "knows" to generate no code, and allocate no registers etc, for primitive states.

```
type State# s
```

The type `GHC.RealWorld` is truly opaque: there are no values defined of this type, and no operations over it. It is "primitive" in that sense - but it is *not unlifted!* Its only role in life is to be the type which distinguishes the `IO` state transformer.

```
data RealWorld
```

6.4.12 State of the world

A single, primitive, value of type `State# RealWorld` is provided.

```
realWorld# :: State# RealWorld
```

(Note: in the compiler, not a `PrimOp`; just a mucho magic `Id`. Exported from `GHC`, though).

6.4.13 Mutable arrays

Corresponding to `Array#` and `ByteArray#`, we have the types of mutable versions of each. In each case, the representation is a pointer to a suitable block of (mutable) heap-allocated storage.

```
type MutableArray# s elt
type MutableByteArray# s
```

Allocation Mutable arrays can be allocated. Only pointer-arrays are initialised; arrays of non-pointers are filled in by "user code" rather than by the array-allocation primitive. Reason: only the pointer case has to worry about GC striking with a partly-initialised array.

```
newArray#       :: Int# -> elt -> State# s -> (# State# s, MutableArray# s elt #)

newCharArray#   :: Int# -> State# s -> (# State# s, MutableByteArray# s elt #)
newIntArray#    :: Int# -> State# s -> (# State# s, MutableByteArray# s elt #)
newAddrArray#   :: Int# -> State# s -> (# State# s, MutableByteArray# s elt #)
newFloatArray#  :: Int# -> State# s -> (# State# s, MutableByteArray# s elt #)
newDoubleArray# :: Int# -> State# s -> (# State# s, MutableByteArray# s elt #)
```

The size of a `ByteArray#` is given in bytes.

Reading and writing

```
readArray#       :: MutableArray# s elt -> Int# -> State# s -> (# State# s, elt #)
readCharArray#   :: MutableByteArray# s -> Int# -> State# s -> (# State# s, Char# #)
readIntArray#    :: MutableByteArray# s -> Int# -> State# s -> (# State# s, Int# #)
readAddrArray#   :: MutableByteArray# s -> Int# -> State# s -> (# State# s, Addr# #)
readFloatArray#  :: MutableByteArray# s -> Int# -> State# s -> (# State# s, Float# #)
readDoubleArray# :: MutableByteArray# s -> Int# -> State# s -> (# State# s, Double# #)

writeArray#       :: MutableArray# s elt -> Int# -> elt     -> State# s -> State# s
writeCharArray#   :: MutableByteArray# s -> Int# -> Char#   -> State# s -> State# s
writeIntArray#    :: MutableByteArray# s -> Int# -> Int#    -> State# s -> State# s
writeAddrArray#   :: MutableByteArray# s -> Int# -> Addr#   -> State# s -> State# s
writeFloatArray#  :: MutableByteArray# s -> Int# -> Float#  -> State# s -> State# s
writeDoubleArray# :: MutableByteArray# s -> Int# -> Double# -> State# s -> State# s
```

Equality One can take "equality" of mutable arrays. What is compared is the *name* or reference to the mutable array, not its contents.

```
sameMutableArray#     :: MutableArray# s elt -> MutableArray# s elt -> Bool
sameMutableByteArray# :: MutableByteArray# s -> MutableByteArray# s -> Bool
```

Freezing mutable arrays Only unsafe-freeze has a primitive. (Safe freeze is done directly in Haskell by copying the array and then using `unsafeFreeze`.)

```
unsafeFreezeArray#     :: MutableArray# s elt -> State# s -> (# State# s, Array# s elt #)
unsafeFreezeByteArray# :: MutableByteArray# s -> State# s -> (# State# s, ByteArray# #)
```

6.4.14 Stable pointers

A stable pointer is a name for a Haskell object which can be passed to the external world. It is "stable" in the sense that the name does not change when the Haskell garbage collector runs — in contrast to the address of the object which may well change.

The stable pointer type is parameterised by the type of the thing which is named.

```
type StablePtr# a
```

A stable pointer is represented by an index into the (static) `StablePointerTable`. The Haskell garbage collector treats the `StablePointerTable` as a source of roots for GC.

The `makeStablePointer` function converts a value into a stable pointer. It is part of the `IO` monad, because we want to be sure we don't allocate one twice by accident, and then only free one of the copies.

```
makeStablePointer#  :: a -> State# RealWorld -> (# State# RealWord, StablePtr# a #)
freeStablePointer#  :: StablePtr# a -> State# RealWorld -> State# RealWorld
deRefStablePointer# :: StablePtr# a -> State# RealWorld -> (# State# RealWorld, a #)
```

There is also a C procedure `FreeStablePtr` which frees a stable pointer.

6.4.15 Foreign objects

A `ForeignObj#` is a reference to an object outside the Haskell world (i.e., from the C world, or a reference to an object on another machine completely.), where the Haskell world has been told "Let me know when you're finished with this ...".

```
type ForeignObj#
```

GHC provides two primitives on `ForeignObj#`:

```
makeForeignObj#
        :: Addr# -- foreign reference
        -> Addr# -- pointer to finalisation routine
        -> (# State# RealWorld, ForeignObj# )
writeForeignObj
        :: ForeignObj#        -- foreign object
        -> Addr#              -- datum
        -> State# RealWorld
        -> State# RealWorld
```

The module `Foreign` (see library documentation) provides a more programmer-friendly interface to foreign objects.

6.4.16 Synchronizing variables (M-vars)

Synchronising variables are the primitive type used to implement Concurrent Haskell's MVars (see the Concurrent Haskell paper for the operational behaviour of these operations).

```
type MVar# s elt        -- primitive

newMVar#    :: State# s -> (# State# s, MVar# s elt #)
takeMVar#   :: SynchVar# s elt -> State# s -> (# State# s, elt #)
putMVar#    :: SynchVar# s elt -> State# s -> State# s
```

6.5 The Posix library

The `Posix` interface gives you access to the set of OS services standardised by POSIX 1003.1b (or the *IEEE Portable Operating System Interface for Computing Environments* - IEEE Std. 1003.1). The interface is accessed by `import Posix` and adding `-syslib posix` on your command-line.

6.5.1 Posix data types

```
data ByteCount  -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `ByteCount` is a primitive of type `unsigned`. At a minimum, an conforming implementation must support values in the range `[0, UINT_MAX]`.

```
data ClockTick  -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `ClockTick` is a primitive of type `clock_t`, which is used to measure intervals of time in fractions of a second. The resolution is determined by `getSysVar ClockTick`.

```
data DeviceID  -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `DeviceID` is a primitive of type `dev_t`. It must be an arithmetic type.

```
data EpochTime -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `EpochTime` is a primitive of type `time_t`, which is used to measure seconds since the Epoch. At a minimum, the implementation must support values in the range `[0, INT_MAX]`.

```
data FileID -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `FileID` is a primitive of type `ino_t`. It must be an arithmetic type.

```
data FileMode -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `FileMode` is a primitive of type `mode_t`. It must be an arithmetic type.

```
data FileOffset -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `FileOffset` is a primitive of type `off_t`. It must be an arithmetic type.

```
data GroupID -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `GroupID` is a primitive of type `gid_t`. It must be an arithmetic type.

```
data Limit -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `Limit` is a primitive of type `long`. At a minimum, the implementation must support values in the range `[LONG_MIN, LONG_MAX]`.

```
data LinkCount -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `LinkCount` is a primitive of type `nlink_t`. It must be an arithmetic type.

```
data ProcessID -- instances of : Eq Ord Num Real Integral Ix Enum Show
type ProcessGroupID = ProcessID
```

A `ProcessID` is a primitive of type `pid_t`. It must be a signed arithmetic type.

```
data UserID -- instances of : Eq Ord Num Real Integral Ix Enum Show
```

A `UserID` is a primitive of type `uid_t`. It must be an arithmetic type.

```
data DirStream
```

A `DirStream` is a primitive of type `DIR *`.

```
data FileStatus
```

A `FileStatus` is a primitive of type `struct stat`.

```
data GroupEntry
```

A `GroupEntry` is a primitive of type `struct group`.

```
data ProcessTimes
```

`ProcessTimes` is a primitive structure containing a `clock_t` and a `struct tms`.

```
data SignalSet
```

An `SignalSet` is a primitive of type `sigset_t`.

```
data SystemID
```

A `SystemID` is a primitive of type `struct utsname`.

```
data TerminalAttributes
```

`TerminalAttributes` is a primitive of type `struct termios`.

```
data UserEntry
```

A `UserEntry` is a primitive of type `struct passwd`.

```
data BaudRate = B0 | B50 | B75 | B110 | B134 | B150 | B200 | B300 | B600
              | B1200 | B1800 | B2400 | B4800 | B9600 | B19200 | B38400
              deriving (Eq, Show)

data Fd

intToFd :: Int -> Fd -- use with care.

data FdOption = AppendOnWrite
              | CloseOnExec
              | NonBlockingRead

data ControlCharacter = EndOfFile
                      | EndOfLine
                      | Erase
                      | Interrupt
                      | Kill
                      | Quit
                      | Suspend
                      | Start
                      | Stop

type ErrorCode = Int

type FileLock = (LockRequest, SeekMode, FileOffset, FileOffset)
--                            whence    start       length
```

```
data FlowAction = SuspendOutput | RestartOutput | TransmitStop | TransmitStart

data Handler = Default | Ignore | Catch (IO ())

data LockRequest = ReadLock | WriteLock | Unlock
                 deriving (Eq, Show)

data OpenMode = ReadOnly | WriteOnly | ReadWrite

data PathVar = LinkLimit
             | InputLineLimit
             | InputQueueLimit
             | FileNameLimit
             | PathNameLimit
             | PipeBufferLimit
             | SetOwnerAndGroupIsRestricted
             | FileNamesAreNotTruncated

data QueueSelector = InputQueue | OutputQueue | BothQueues

type Signal = Int

data SysVar = ArgumentLimit
            | ChildLimit
            | ClockTick
            | GroupLimit
            | OpenFileLimit
            | PosixVersion
            | HasSavedIDs
            | HasJobControl

data TerminalMode = InterruptOnBreak       -- BRKINT
                  | MapCRtoLF              -- ICRNL
                  | IgnoreBreak            -- IGNBRK
                  | IgnoreCR               -- IGNCR
                  | IgnoreParityErrors     -- IGNPAR
                  | MapLFtoCR              -- INLCR
                  | CheckParity            -- INPCK
                  | StripHighBit           -- ISTRIP
                  | StartStopInput         -- IXOFF
                  | StartStopOutput        -- IXON
                  | MarkParityErrors       -- PARMRK
                  | ProcessOutput          -- OPOST
                  | LocalMode              -- CLOCAL
                  | ReadEnable             -- CREAD
                  | TwoStopBits            -- CSTOPB
                  | HangupOnClose          -- HUPCL
                  | EnableParity           -- PARENB
                  | OddParity              -- PARODD
                  | EnableEcho             -- ECHO
                  | EchoErase              -- ECHOE
```

```
                | EchoKill                -- ECHOK
                | EchoLF                 -- ECHONL
                | ProcessInput           -- ICANON
                | ExtendedFunctions      -- IEXTEN
                | KeyboardInterrupts     -- ISIG
                | NoFlushOnInterrupt     -- NOFLSH
                | BackgroundWriteInterrupt -- TOSTOP

data TerminalState = Immediately | WhenDrained | WhenFlushed

data ProcessStatus = Exited ExitCode
                   | Terminated Signal
                   | Stopped Signal
                   deriving (Eq, Show)
```

6.5.2 Posix Process Primitives

```
forkProcess :: IO (Maybe ProcessID)
```

`forkProcess` calls `fork`, returning `Just pid` to the parent, where `pid` is the ProcessID of the child, and returning `Nothing` to the child.

```
executeFile :: FilePath                  -- Command
            -> Bool                      -- Search PATH?
            -> [String]                  -- Arguments
            -> Maybe [(String, String)]  -- Environment
            -> IO ()
```

`executeFile cmd args env` calls one of the `execv*` family, depending on whether or not the current PATH is to be searched for the command, and whether or not an environment is provided to supersede the process's current environment. The basename (leading directory names suppressed) of the command is passed to `execv*` as `arg[0]`; the argument list passed to `executeFile` therefore begins with `arg[1]`.

```
Search PATH?    Supersede environ?      Call
~~~~~~~~~~~~    ~~~~~~~~~~~~~~~~~~      ~~~~~~~
False           False                   execv
False           True                    execve
True            False                   execvp
True            True                    execvpe*
```

Note that `execvpe` is not provided by the POSIX standard, and must be written by hand. Care must be taken to ensure that the search path is extracted from the original environment, and not from the environment to be passed on to the new image.

NOTE: In general, sharing open files between parent and child processes is potential bug farm, and should be avoided unless you really depend on this 'feature' of POSIX' `fork()` semantics. Using Haskell, there's the extra complication that arguments to `executeFile` might come from files that are read lazily (using `hGetContents`, or some such.) If this is the case, then for your own sanity, please ensure that the arguments to `executeFile` have been fully evaluated before calling `forkProcess` (followed by `executeFile`.) Consider yourself warned :-)

A successful `executeFile` overlays the current process image with a new one, so it only returns on failure.

```
runProcess :: FilePath                    -- Command
           -> [String]                    -- Arguments
           -> Maybe [(String, String)]    -- Environment (Nothing -> Inherited)
           -> Maybe FilePath              -- Working directory (Nothing -> inherited)
           -> Maybe Handle                -- stdin  (Nothing -> inherited)
           -> Maybe Handle                -- stdout (Nothing -> inherited)
           -> Maybe Handle                -- stderr (Nothing -> inherited)
           -> IO ()
```

`runProcess` is our candidate for the high-level OS-independent primitive.

`runProcess cmd args env wd inhdl outhdl errhdl` runs `cmd` (searching the current `PATH`) with arguments `args`. If `env` is `Just pairs`, the command is executed with the environment specified by `pairs` of variables and values; otherwise, the command is executed with the current environment. If `wd` is `Just dir`, the command is executed with working directory `dir`; otherwise, the command is executed in the current working directory. If {`in,out,err`hdl} is `Just handle`, the command is executed with the `Fd` for `std{in,out,err}` attached to the specified `handle`; otherwise, the `Fd` for `std{in,out,err}` is left unchanged.

```
getProcessStatus :: Bool              -- Block?
                 -> Bool              -- Stopped processes?
                 -> ProcessID
                 -> IO (Maybe ProcessStatus)
```

`getProcessStatus blk stopped pid` calls `waitpid`, returning `Just tc`, the `ProcessStatus` for process `pid` if it is available, `Nothing` otherwise. If `blk` is `False`, then `WNOHANG` is set in the options for `waitpid`, otherwise not. If `stopped` is `True`, then `WUNTRACED` is set in the options for `waitpid`, otherwise not.

```
getGroupProcessStatus :: Bool         -- Block?
                      -> Bool         -- Stopped processes?
                      -> ProcessGroupID
                      -> IO (Maybe (ProcessID, ProcessStatus))
```

`getGroupProcessStatus blk stopped pgid` calls `waitpid`, returning `Just (pid, tc)`, the `ProcessID` and `ProcessStatus` for any process in group `pgid` if one is available, `Nothing` otherwise. If `blk` is `False`, then `WNOHANG` is set in the options for `waitpid`, otherwise not. If `stopped` is `True`, then `WUNTRACED` is set in the options for `waitpid`, otherwise not.

```
getAnyProcessStatus :: Bool           -- Block?
                    -> Bool           -- Stopped processes?
                    -> IO (Maybe (ProcessID, ProcessStatus))
```

`getAnyProcessStatus blk stopped` calls `waitpid`, returning `Just (pid, tc)`, the `ProcessID` and `ProcessStatus` for any child process if one is available, `Nothing` otherwise. If `blk` is `False`, then `WNOHANG` is set in the options for `waitpid`, otherwise not. If `stopped` is `True`, then `WUNTRACED` is set in the options for `waitpid`, otherwise not.

```
exitImmediately :: ExitCode -> IO ()
```

`exitImmediately status` calls `_exit` to terminate the process with the indicated exit `status`. The operation never returns.

```
getEnvironment :: IO [(String, String)]
```

`getEnvironment` parses the environment variable mapping provided by `environ`, returning `(variable, value)` pairs. The operation never fails.

```
setEnvironment :: [(String, String)] -> IO ()
```

`setEnvironment` replaces the process environment with the provided mapping of `(variable, value)` pairs.

```
getEnvVar :: String -> IO String
```

`getEnvVar var` returns the value associated with variable `var` in the current environment (identical functionality provided through standard Haskell library function `System.getEnv`).

The operation may fail with:

`NoSuchThing`
: The variable has no mapping in the current environment.

```
setEnvVar :: String -> String -> IO ()
```

`setEnvVar var val` sets the value associated with variable `var` in the current environment to be `val`. Any previous mapping is superseded.

```
removeEnvVar :: String -> IO ()
```

`removeEnvVar var` removes any value associated with variable `var` in the current environment. Deleting a variable for which there is no mapping does not generate an error.

```
nullSignal :: Signal
nullSignal = 0

backgroundRead, sigTTIN        :: Signal
backgroundWrite, sigTTOU       :: Signal
continueProcess, sigCONT       :: Signal
floatingPointException, sigFPE :: Signal
illegalInstruction, sigILL     :: Signal
internalAbort, sigABRT         :: Signal
keyboardSignal, sigINT         :: Signal
keyboardStop, sigTSTP          :: Signal
keyboardTermination, sigQUIT   :: Signal
killProcess, sigKILL           :: Signal
lostConnection, sigHUP         :: Signal
openEndedPipe, sigPIPE         :: Signal
processStatusChanged, sigCHLD  :: Signal
```

```
realTimeAlarm, sigALRM         :: Signal
segmentationViolation, sigSEGV :: Signal
softwareStop, sigSTOP          :: Signal
softwareTermination, sigTERM   :: Signal
userDefinedSignal1, sigUSR1    :: Signal
userDefinedSignal2, sigUSR2    :: Signal

signalProcess :: Signal -> ProcessID -> IO ()
```

`signalProcess int pid` calls `kill` to signal process `pid` with interrupt signal `int`.

```
raiseSignal :: Signal -> IO ()
```

`raiseSignal int` calls `kill` to signal the current process with interrupt signal `int`.

```
signalProcessGroup :: Signal -> ProcessGroupID -> IO ()
```

`signalProcessGroup int pgid` calls `kill` to signal all processes in group `pgid` with interrupt signal `int`.

```
setStoppedChildFlag :: Bool -> IO Bool
```

`setStoppedChildFlag bool` sets a flag which controls whether or not the `NOCLDSTOP` option will be used the next time a signal handler is installed for `SIGCHLD`. If `bool` is `True` (the default), `NOCLDSTOP` will not be used; otherwise it will be. The operation never fails.

```
queryStoppedChildFlag :: IO Bool
```

`queryStoppedChildFlag` queries the flag which controls whether or not the `NOCLDSTOP` option will be used the next time a signal handler is installed for `SIGCHLD`. If `NOCLDSTOP` will be used, it returns `False`; otherwise (the default) it returns `True`. The operation never fails.

```
emptySignalSet :: SignalSet
fullSignalSet  :: SignalSet
addSignal      :: Signal -> SignalSet -> SignalSet
deleteSignal   :: Signal -> SignalSet -> SignalSet
inSignalSet    :: Signal -> SignalSet -> Bool

installHandler :: Signal
               -> Handler
               -> Maybe SignalSet       -- other signals to block
               -> IO Handler            -- old handler
```

`installHandler int handler iset` calls `sigaction` to install an interrupt handler for signal `int`. If `handler` is `Default`, `SIG_DFL` is installed; if `handler` is `Ignore`, `SIG_IGN` is installed; if `handler` is `Catch action`, a handler is installed which will invoke `action` in a new thread when (or shortly after) the signal is received. See Section 5.12.1 (Concurrent Haskell) for details on how to communicate between threads.

If `iset` is `Just s`, then the `sa_mask` of the `sigaction` structure is set to `s`; otherwise it is cleared. The previously installed signal handler for `int` is returned.

```
getSignalMask :: IO SignalSet
```

`getSignalMask` calls `sigprocmask` to determine the set of interrupts which are currently being blocked.

```
setSignalMask :: SignalSet -> IO SignalSet
```

`setSignalMask mask` calls `sigprocmask` with `SIG_SETMASK` to block all interrupts in `mask`. The previous set of blocked interrupts is returned.

```
blockSignals :: SignalSet -> IO SignalSet
```

`setSignalMask mask` calls `sigprocmask` with `SIG_BLOCK` to add all interrupts in `mask` to the set of blocked interrupts. The previous set of blocked interrupts is returned.

```
unBlockSignals :: SignalSet -> IO SignalSet
```

`setSignalMask mask` calls `sigprocmask` with `SIG_UNBLOCK` to remove all interrupts in `mask` from the set of blocked interrupts. The previous set of blocked interrupts is returned.

```
getPendingSignals :: IO SignalSet
```

`getPendingSignals` calls `sigpending` to obtain the set of interrupts which have been received but are currently blocked.

```
awaitSignal :: Maybe SignalSet -> IO ()
```

`awaitSignal iset` suspends execution until an interrupt is received. If `iset` is `Just s`, `awaitSignal` calls `sigsuspend`, installing `s` as the new signal mask before suspending execution; otherwise, it calls `pause`. `awaitSignal` returns on receipt of a signal. If you have installed any signal handlers with `installHandler`, it may be wise to call `yield` directly after `awaitSignal` to ensure that the signal handler runs as promptly.

```
scheduleAlarm :: Int -> IO Int
```

`scheduleAlarm i` calls `alarm` to schedule a real time alarm at least `i` seconds in the future.

```
sleep :: Int -> IO ()
```

`sleep i` calls `sleep` to suspend execution of the program until at least `i` seconds have elapsed or a signal is received.

6.5.3 Posix Process Environment

```
getProcessID :: IO ProcessID
```

`getProcessID` calls `getpid` to obtain the `ProcessID` for the current process.

```
getParentProcessID :: IO ProcessID
```

`getProcessID` calls `getppid` to obtain the `ProcessID` for the parent of the current process.

```
getRealUserID :: IO UserID
```

`getRealUserID` calls `getuid` to obtain the real `UserID` associated with the current process.

```
getEffectiveUserID :: IO UserID
```

`getRealUserID` calls `geteuid` to obtain the effective `UserID` associated with the current process.

```
setUserID :: UserID -> IO ()
```

`setUserID uid` calls `setuid` to set the real, effective, and saved set-user-id associated with the current process to `uid`.

```
getLoginName :: IO String
```

`getLoginName` calls `getlogin` to obtain the login name associated with the current process.

```
getRealGroupID :: IO GroupID
```

`getRealGroupID` calls `getgid` to obtain the real `GroupID` associated with the current process.

```
getEffectiveGroupID :: IO GroupID
```

`getEffectiveGroupID` calls `getegid` to obtain the effective `GroupID` associated with the current process.

```
setGroupID :: GroupID -> IO ()
```

`setGroupID gid` calls `setgid` to set the real, effective, and saved set-group-id associated with the current process to `gid`.

```
getGroups :: IO [GroupID]
```

`getGroups` calls `getgroups` to obtain the list of supplementary `GroupID`s associated with the current process.

```
getEffectiveUserName :: IO String
```

`getEffectiveUserName` calls `cuserid` to obtain a name associated with the effective `UserID` of the process.

```
getProcessGroupID :: IO ProcessGroupID
```

`getProcessGroupID` calls `getpgrp` to obtain the `ProcessGroupID` for the current process.

```
createProcessGroup :: ProcessID -> IO ProcessGroupID
```

`createProcessGroup pid` calls `setpgid` to make process `pid` a new process group leader.

```
joinProcessGroup :: ProcessGroupID -> IO ProcessGroupID
```

`joinProcessGroup pgid` calls `setpgid` to set the `ProcessGroupID` of the current process to `pgid`.

```
setProcessGroupID :: ProcessID -> ProcessGroupID -> IO ()
```

`setProcessGroupID pid pgid` calls `setpgid` to set the `ProcessGroupID` for process `pid` to `pgid`.

```
createSession :: IO ProcessGroupID
```

`createSession` calls `setsid` to create a new session with the current process as session leader.

```
systemName :: SystemID -> String
nodeName :: SystemID -> String
release :: SystemID -> String
version :: SystemID -> String
machine :: SystemID -> String

getSystemID :: IO SystemID
```

`getSystemID` calls `uname` to obtain information about the current operating system.

```
> epochTime :: IO EpochTime
```

`epochTime` calls `time` to obtain the number of seconds that have elapsed since the epoch (Jan 01 00:00:00 GMT 1970).

```
elapsedTime     :: ProcessTimes -> ClockTick
userTime        :: ProcessTimes -> ClockTick
systemTime      :: ProcessTimes -> ClockTick
childUserTime   :: ProcessTimes -> ClockTick
childSystemTime :: ProcessTimes -> ClockTick

getProcessTimes :: IO ProcessTimes
```

`getProcessTimes` calls `times` to obtain time-accounting information for the current process and its children.

```
getControllingTerminalName :: IO FilePath
```

`getControllingTerminalName` calls `ctermid` to obtain a name associated with the controlling terminal for the process. If a controlling terminal exists, `getControllingTerminalName` returns the name of the controlling terminal.

The operation may fail with:

`NoSuchThing`

There is no controlling terminal, or its name cannot be determined.

`SystemError`

Various other causes.

```
getTerminalName :: Fd -> IO FilePath
```

`getTerminalName fd` calls `ttyname` to obtain a name associated with the terminal for `Fd fd`. If `fd` is associated with a terminal, `getTerminalName` returns the name of the terminal.

The operation may fail with:

`InappropriateType`

The channel is not associated with a terminal.

`NoSuchThing`

The channel is associated with a terminal, but it has no name.

`SystemError`

Various other causes.

```
queryTerminal :: Fd -> IO Bool
```

`queryTerminal fd` calls `isatty` to determine whether or not `Fd fd` is associated with a terminal.

```
getSysVar :: SysVar -> IO Limit
```

`getSysVar var` calls `sysconf` to obtain the dynamic value of the requested configurable system limit or option. For defined system limits, `getSysVar` returns the associated value. For defined system options, the result of `getSysVar` is undefined, but not failure.

The operation may fail with:

`NoSuchThing`

The requested system limit or option is undefined.

6.5.4 Posix operations on files and directories

```
openDirStream :: FilePath -> IO DirStream
```

`openDirStream dir` calls `opendir` to obtain a directory stream for `dir`.

```
readDirStream :: DirStream -> IO String
```

`readDirStream dp` calls `readdir` to obtain the next directory entry (`struct dirent`) for the open directory stream `dp`, and returns the `d_name` member of that structure.

The operation may fail with:

`EOF`

End of file has been reached.

`SystemError`

Various other causes.

```
rewindDirStream :: DirStream -> IO ()
```

`rewindDirStream dp` calls `rewinddir` to reposition the directory stream `dp` at the beginning of the directory.

```
closeDirStream :: DirStream -> IO ()
```

`closeDirStream dp` calls `closedir` to close the directory stream `dp`.

```
getWorkingDirectory :: IO FilePath
```

`getWorkingDirectory` calls `getcwd` to obtain the name of the current working directory.

```
changeWorkingDirectory :: FilePath -> IO ()
```

`changeWorkingDirectory dir` calls `chdir` to change the current working directory to `dir`.

```
nullFileMode       :: FileMode       -- ---------
ownerReadMode      :: FileMode       -- r--------
ownerWriteMode     :: FileMode       -- -w-------
ownerExecuteMode   :: FileMode       -- --x------
groupReadMode      :: FileMode       -- ---r-----
groupWriteMode     :: FileMode       -- ----w----
groupExecuteMode   :: FileMode       -- -----x---
otherReadMode      :: FileMode       -- ------r--
otherWriteMode     :: FileMode       -- -------w-
otherExecuteMode   :: FileMode       -- --------x
setUserIDMode      :: FileMode       -- --S------
setGroupIDMode     :: FileMode       -- -----S---

stdFileMode        :: FileMode       -- rw-rw-rw-

ownerModes         :: FileMode       -- rwx------
groupModes         :: FileMode       -- ---rwx---
otherModes         :: FileMode       -- ------rwx
accessModes        :: FileMode       -- rwxrwxrwx

unionFileModes     :: FileMode -> FileMode -> FileMode
intersectFileModes :: FileMode -> FileMode -> FileMode

stdInput  :: Fd
stdInput  = intToFd 0

stdOutput :: Fd
```

```
stdOutput = intToFd 1

stdError  :: Fd
stdError  = intToFd 2

data OpenFileFlags =
 OpenFileFlags {
    append    :: Bool,
    exclusive :: Bool,
    noctty    :: Bool,
    nonBlock  :: Bool,
    trunc     :: Bool
 }

openFd :: FilePath
       -> OpenMode
       -> Maybe FileMode  -- Just x => O_CREAT, Nothing => must exist
       -> OpenFileFlags
       -> IO Fd
```

`openFd path acc mode (OpenFileFlags app excl noctty nonblock trunc)` calls `open` to obtain a `Fd` for the file `path` with access mode `acc`. If `mode` is `Just m`, the `O_CREAT` flag is set and the file's permissions will be based on `m` if it does not already exist; otherwise, the `O_CREAT` flag is not set. The arguments `app`, `excl`, `noctty`, `nonblock`, and `trunc` control whether or not the flags `O_APPEND`, `O_EXCL`, `O_NOCTTY`, `O_NONBLOCK`, and `O_TRUNC` are set, respectively.

```
createFile :: FilePath -> FileMode -> IO Fd
```

`createFile path mode` calls `creat` to obtain a `Fd` for file `path`, which will be created with permissions based on `mode` if it does not already exist.

```
setFileCreationMask :: FileMode -> IO FileMode
```

`setFileCreationMask mode` calls `umask` to set the process's file creation mask to `mode`. The previous file creation mask is returned.

```
createLink :: FilePath -> FilePath -> IO ()
```

`createLink old new` calls `link` to create a new path, `new`, linked to an existing file, `old`.

```
createDirectory :: FilePath -> FileMode -> IO ()
```

`createDirectory dir mode` calls `mkdir` to create a new directory, `dir`, with permissions based on `mode`.

```
createNamedPipe :: FilePath -> FileMode -> IO ()
```

`createNamedPipe fifo mode` calls `mkfifo` to create a new named pipe, `fifo`, with permissions based on `mode`.

```
removeLink :: FilePath -> IO ()
```

`removeLink path` calls `unlink` to remove the link named `path`.

```
removeDirectory :: FilePath -> IO ()
```

`removeDirectory dir` calls `rmdir` to remove the directory named `dir`.

```
rename :: FilePath -> FilePath -> IO ()
```

`rename old new` calls `rename` to rename a file or directory from `old` to `new`.

```
fileMode          :: FileStatus -> FileMode

fileID            :: FileStatus -> FileID
deviceID          :: FileStatus -> DeviceID

linkCount         :: FileStatus -> LinkCount

fileOwner         :: FileStatus -> UserID
fileGroup         :: FileStatus -> GroupID
fileSize          :: FileStatus -> FileOffset

accessTime        :: FileStatus -> EpochTime
modificationTime  :: FileStatus -> EpochTime
statusChangeTime  :: FileStatus -> EpochTime

isDirectory       :: FileStatus -> Bool
isCharacterDevice :: FileStatus -> Bool
isBlockDevice     :: FileStatus -> Bool
isRegularFile     :: FileStatus -> Bool
isNamedPipe       :: FileStatus -> Bool

getFileStatus     :: FilePath -> IO FileStatus
```

`getFileStatus path` calls `stat` to get the `FileStatus` information for the file `path`.

```
getFdStatus :: Fd -> IO FileStatus
```

`getFdStatus fd` calls `fstat` to get the `FileStatus` information for the file associated with `Fd fd`.

```
queryAccess :: FilePath -> Bool -> Bool -> Bool -> IO Bool
```

`queryAccess path r w x` calls `access` to test the access permissions for file `path`. The three arguments, `r`, `w`, and `x` control whether or not `access` is called with `R_OK`, `W_OK`, and `X_OK` respectively.

```
queryFile :: FilePath -> IO Bool
```

`queryFile path` calls `access` with `F_OK` to test for the existence for file `path`.

```
setFileMode :: FilePath -> FileMode -> IO ()
```

`setFileMode path mode` calls `chmod` to set the permission bits associated with file `path` to `mode`.

```
setOwnerAndGroup :: FilePath -> UserID -> GroupID -> IO ()
```

`setOwnerAndGroup path uid gid` calls `chown` to set the `UserID` and `GroupID` associated with file `path` to `uid` and `gid`, respectively.

```
setFileTimes :: FilePath -> EpochTime -> EpochTime -> IO ()
```

`setFileTimes path atime mtime` calls `utime` to set the access and modification times associated with file `path` to `atime` and `mtime`, respectively.

```
touchFile :: FilePath -> IO ()
```

`touchFile path` calls `utime` to set the access and modification times associated with file `path` to the current time.

```
getPathVar :: PathVar -> FilePath -> IO Limit
```

`getPathVar var path` calls `pathconf` to obtain the dynamic value of the requested configurable file limit or option associated with file or directory `path`. For defined file limits, `getPathVar` returns the associated value. For defined file options, the result of `getPathVar` is undefined, but not failure. The operation may fail with:

`NoSuchThing`
: The requested file limit or option is undefined.

`SystemError`
: Various other causes.

```
getFdVar :: PathVar -> Fd -> IO Limit
```

`getFdVar var fd` calls `fpathconf` to obtain the dynamic value of the requested configurable file limit or option associated with the file or directory attached to the open channel `fd`. For defined file limits, `getFdVar` returns the associated value. For defined file options, the result of `getFdVar` is undefined, but not failure.

The operation may fail with:

`NoSuchThing`
: The requested file limit or option is undefined.

`SystemError`
: Various other causes.

6.5.5 Posix Input and Output Primitives

```
createPipe :: IO (Fd, Fd)
```

`createPipe` calls `pipe` to create a pipe and returns a pair of `Fd`s, the first for reading and the second for writing.

```
dup :: Fd -> IO Fd
```

`dup fd` calls `dup` to duplicate `Fd fd` to another `Fd`.

```
dupTo :: Fd -> Fd -> IO ()
```

`dupTo src dst` calls `dup2` to duplicate `Fd src` to `Fd dst`.

```
fdClose :: Fd -> IO ()
```

`fdClose fd` calls `close` to close `Fd fd`.

```
fdRead :: Fd -> ByteCount -> IO (String, ByteCount)
```

`fdRead fd nbytes` calls `read` to read at most `nbytes` bytes from `Fd fd`, and returns the result as a string paired with the number of bytes actually read.

The operation may fail with:

`EOF`
: End of file has been reached.

`SystemError`
: Various other causes.

```
fdWrite :: Fd -> String -> IO ByteCount
```

`fdWrite fd s` calls `write` to write the string `s` to `Fd fd` as a contiguous sequence of bytes. It returns the number of bytes successfully written.

```
queryFdOption :: FdOption -> Fd -> IO Bool
```

`getFdOption opt fd` calls `fcntl` to determine whether or not the flag associated with `FdOption opt` is set for `Fd fd`.

```
setFdOption :: Fd -> FdOption -> Bool -> IO ()
```

`setFdOption fd opt val` calls `fcntl` to set the flag associated with `FdOption opt` on `Fd fd` to `val`.

```
getLock :: Fd -> FileLock -> IO (Maybe (ProcessID, FileLock))
```

`getLock fd lock` calls `fcntl` to get the first `FileLock` for `Fd fd` which blocks the `FileLock lock`. If no such `FileLock` exists, `getLock` returns `Nothing`. Otherwise, it returns `Just (pid, block)`, where `block` is the blocking `FileLock` and `pid` is the `ProcessID` of the process holding the blocking `FileLock`.

```
setLock :: Fd -> FileLock -> IO ()
```

`setLock fd lock` calls `fcntl` with `F_SETLK` to set or clear a lock segment for `Fd fd` as indicated by the `FileLock lock`. `setLock` does not block, but fails with `SystemError` if the request cannot be satisfied immediately.

```
waitToSetLock :: Fd -> FileLock -> IO ()
```

`waitToSetLock fd lock` calls `fcntl` with `F_SETLKW` to set or clear a lock segment for `Fd fd` as indicated by the `FileLock lock`. If the request cannot be satisfied immediately, `waitToSetLock` blocks until the request can be satisfied.

```
fdSeek :: Fd -> SeekMode -> FileOffset -> IO FileOffset
```

`fdSeek fd whence offset` calls `lseek` to position the `Fd fd` at the given `offset` from the starting location indicated by `whence`. It returns the resulting offset from the start of the file in bytes.

6.5.6 Posix, Device- and Class-Specific Functions

```
terminalMode    :: TerminalMode -> TerminalAttributes -> Bool
withMode        :: TerminalAttributes -> TerminalMode -> TerminalAttributes
withoutMode     :: TerminalAttributes -> TerminalMode -> TerminalAttributes

bitsPerByte     :: TerminalAttributes -> Int
withBits        :: TerminalAttributes -> Int -> TerminalAttributes

controlChar     :: TerminalAttributes -> ControlCharacter -> Maybe Char
withCC          :: TerminalAttributes
                -> (ControlCharacter, Char)
                -> TerminalAttributes
withoutCC       :: TerminalAttributes
                -> ControlCharacter
                -> TerminalAttributes

inputTime       :: TerminalAttributes -> Int
withTime        :: TerminalAttributes -> Int -> TerminalAttributes

minInput        :: TerminalAttributes -> Int
withMinInput    :: TerminalAttributes -> Int -> TerminalAttributes

inputSpeed      :: TerminalAttributes -> BaudRate
withInputSpeed  :: TerminalAttributes -> BaudRate -> TerminalAttributes

outputSpeed     :: TerminalAttributes -> BaudRate
withOutputSpeed :: TerminalAttributes -> BaudRate -> TerminalAttributes
```

```
getTerminalAttributes :: Fd -> IO TerminalAttributes
```

`getTerminalAttributes fd` calls `tcgetattr` to obtain the `TerminalAttributes` associated with `Fd fd`.

```
setTerminalAttributes :: Fd
                      -> TerminalAttributes
                      -> TerminalState
                      -> IO ()
```

`setTerminalAttributes fd attr ts` calls `tcsetattr` to change the `TerminalAttributes` associated with `Fd fd` to `attr`, when the terminal is in the state indicated by `ts`.

```
sendBreak :: Fd -> Int -> IO ()
```

`sendBreak fd duration` calls `tcsendbreak` to transmit a continuous stream of zero-valued bits on `Fd fd` for the specified implementation-dependent `duration`.

```
drainOutput :: Fd -> IO ()
```

`drainOutput fd` calls `tcdrain` to block until all output written to `Fd fd` has been transmitted.

```
discardData :: Fd -> QueueSelector -> IO ()
```

`discardData fd queues` calls `tcflush` to discard pending input and/or output for `Fd fd`, as indicated by the `QueueSelector` `queues`.

```
controlFlow :: Fd -> FlowAction -> IO ()
```

`controlFlow fd action` calls `tcflow` to control the flow of data on `Fd fd`, as indicated by `action`.

```
getTerminalProcessGroupID :: Fd -> IO ProcessGroupID
```

`getTerminalProcessGroupID fd` calls `tcgetpgrp` to obtain the `ProcessGroupID` of the foreground process group associated with the terminal attached to `Fd fd`.

```
setTerminalProcessGroupID :: Fd -> ProcessGroupID -> IO ()
```

`setTerminalProcessGroupID fd pgid` calls `tcsetpgrp` to set the `ProcessGroupID` of the foreground process group associated with the terminal attached to `Fd fd` to `pgid`.

6.5.7 Posix System Databases

```
groupName    :: GroupEntry -> String
groupID      :: GroupEntry -> GroupID
groupMembers :: GroupEntry -> [String]

getGroupEntryForID :: GroupID -> IO GroupEntry
```

`getGroupEntryForID gid` calls `getgrgid` to obtain the `GroupEntry` information associated with `GroupID gid`.

The operation may fail with:

`NoSuchThing`
: There is no group entry for the GroupID.

```
getGroupEntryForName :: String -> IO GroupEntry
```

`getGroupEntryForName name` calls `getgrnam` to obtain the `GroupEntry` information associated with the group called `name`.

The operation may fail with:

`NoSuchThing`
: There is no group entry for the name.

```
userName      :: UserEntry -> String
userID        :: UserEntry -> UserID
userGroupID   :: UserEntry -> GroupID
homeDirectory :: UserEntry -> String
userShell     :: UserEntry -> String

getUserEntryForID :: UserID -> IO UserEntry
```

`getUserEntryForID gid` calls `getpwuid` to obtain the `UserEntry` information associated with `UserID uid`. The operation may fail with:

`NoSuchThing`
: There is no user entry for the UserID.

```
getUserEntryForName :: String -> IO UserEntry
```

`getUserEntryForName name` calls `getpwnam` to obtain the `UserEntry` information associated with the user login `name`.

The operation may fail with:

`NoSuchThing`
: There is no user entry for the name.

6.5.8 POSIX Errors

```
getErrorCode :: IO ErrorCode
```

`getErrorCode` returns the current value of the external variable `errno`. It never fails.

```
setErrorCode :: ErrorCode -> IO ()
```

`setErrorCode err` sets the external variable `errno` to `err`. It never fails.

```
noError :: ErrorCode
noError = 0

argumentListTooLong, e2BIG              :: ErrorCode
badFd, eBADF                            :: ErrorCode
brokenPipe, ePIPE                       :: ErrorCode
directoryNotEmpty, eNOTEMPTY            :: ErrorCode
execFormatError, eNOEXEC                :: ErrorCode
fileAlreadyExists, eEXIST               :: ErrorCode
fileTooLarge, eFBIG                     :: ErrorCode
filenameTooLong, eNAMETOOLONG           :: ErrorCode
improperLink, eXDEV                     :: ErrorCode
inappropriateIOControlOperation, eNOTTY :: ErrorCode
inputOutputError, eIO                   :: ErrorCode
interruptedOperation, eINTR             :: ErrorCode
invalidArgument, eINVAL                 :: ErrorCode
invalidSeek, eSPIPE                     :: ErrorCode
isADirectory, eISDIR                    :: ErrorCode
noChildProcess, eCHILD                  :: ErrorCode
noLocksAvailable, eNOLCK                :: ErrorCode
noSpaceLeftOnDevice, eNOSPC             :: ErrorCode
noSuchOperationOnDevice, eNODEV         :: ErrorCode
noSuchDeviceOrAddress, eNXIO            :: ErrorCode
noSuchFileOrDirectory, eNOENT           :: ErrorCode
noSuchProcess, eSRCH                    :: ErrorCode
notADirectory, eNOTDIR                  :: ErrorCode
notEnoughMemory, eNOMEM                 :: ErrorCode
operationNotImplemented, eNOSYS         :: ErrorCode
operationNotPermitted, ePERM            :: ErrorCode
permissionDenied, eACCES                :: ErrorCode
readOnlyFileSystem, eROFS               :: ErrorCode
resourceBusy, eBUSY                     :: ErrorCode
resourceDeadlockAvoided, eDEADLK        :: ErrorCode
resourceTemporarilyUnavailable, eAGAIN  :: ErrorCode
tooManyLinks, eMLINK                    :: ErrorCode
tooManyOpenFiles, eMFILE                :: ErrorCode
tooManyOpenFilesInSystem, eNFILE        :: ErrorCode
```

6.6 Miscellaneous libraries

This section describes a collection of Haskell libraries we've collected over the years. Access to any of these modules is provided by giving the `-syslib misc`.

6.6.1 The `Bag` type

A *bag* is an unordered collection of elements which may contain duplicates. To use, `import Bag`.

```
data Bag elt    -- abstract

emptyBag        :: Bag elt
unitBag         :: elt -> Bag elt

consBag         :: elt       -> Bag elt -> Bag elt
snocBag         :: Bag elt   -> elt     -> Bag elt

unionBags       :: Bag elt   -> Bag elt -> Bag elt
unionManyBags   :: [Bag elt] -> Bag elt

isEmptyBag      :: Bag elt   -> Bool
elemBag         :: Eq elt => elt -> Bag elt -> Bool

filterBag       :: (elt -> Bool) -> Bag elt -> Bag elt
partitionBag    :: (elt -> Bool) -> Bag elt-> (Bag elt, Bag elt)
        -- returns the elements that do/don't satisfy the predicate

concatBag       :: Bag (Bag a) -> Bag a
foldBag         :: (r -> r -> r) -> (a -> r) -> r -> Bag a -> r
mapBag          :: (a -> b) -> Bag a -> Bag b

listToBag       :: [elt] -> Bag elt
bagToList       :: Bag elt -> [elt]
```

6.6.2 The `FiniteMap` type

What functional programmers call a *finite map*, everyone else calls a *lookup table*.

Out code is derived from that in this paper:

> S Adams "Efficient sets: a balancing act" Journal of functional programming 3(4) Oct 1993, pages 553-562

Guess what? The implementation uses balanced trees.

```
data FiniteMap key elt  -- abstract

--      BUILDING
emptyFM         :: FiniteMap key elt
unitFM          :: key -> elt -> FiniteMap key elt
listToFM        :: Ord key => [(key,elt)] -> FiniteMap key elt
                        -- In the case of duplicates, the last is taken

--      ADDING AND DELETING
                   -- Throws away any previous binding
                   -- In the list case, the items are added starting with the
                   -- first one in the list
addToFM         :: Ord key => FiniteMap key elt -> key -> elt  -> FiniteMap key elt
addListToFM     :: Ord key => FiniteMap key elt -> [(key,elt)] -> FiniteMap key elt
```

```
                -- Combines with previous binding
                -- In the combining function, the first argument is
                -- the "old" element, while the second is the "new" one.
addToFM_C       :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> key -> elt
                           -> FiniteMap key elt
addListToFM_C   :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> [(key,elt)]
                           -> FiniteMap key elt

                -- Deletion doesn't complain if you try to delete something
                -- which isn't there
delFromFM       :: Ord key => FiniteMap key elt -> key   -> FiniteMap key elt
delListFromFM   :: Ord key => FiniteMap key elt -> [key] -> FiniteMap key elt

--      COMBINING
                -- Bindings in right argument shadow those in the left
plusFM          :: Ord key => FiniteMap key elt -> FiniteMap key elt
                           -> FiniteMap key elt

                  -- Combines bindings for the same thing with the given function
plusFM_C        :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> FiniteMap key elt -> FiniteMap key elt

minusFM         :: Ord key => FiniteMap key elt -> FiniteMap key elt -> FiniteMap key elt
                  -- (minusFM a1 a2) deletes from a1 any bindings which are bound in a2

intersectFM     :: Ord key => FiniteMap key elt -> FiniteMap key elt -> FiniteMap key elt
intersectFM_C   :: Ord key => (elt -> elt -> elt)
                           -> FiniteMap key elt -> FiniteMap key elt -> FiniteMap key elt

--      MAPPING, FOLDING, FILTERING
foldFM          :: (key -> elt -> a -> a) -> a -> FiniteMap key elt -> a
mapFM           :: (key -> elt1 -> elt2) -> FiniteMap key elt1 -> FiniteMap key elt2
filterFM        :: Ord key => (key -> elt -> Bool)
                           -> FiniteMap key elt -> FiniteMap key elt

--      INTERROGATING
sizeFM          :: FiniteMap key elt -> Int
isEmptyFM       :: FiniteMap key elt -> Bool

elemFM          :: Ord key => key -> FiniteMap key elt -> Bool
lookupFM        :: Ord key => FiniteMap key elt -> key -> Maybe elt
lookupWithDefaultFM
                :: Ord key => FiniteMap key elt -> elt -> key -> elt
                -- lookupWithDefaultFM supplies a "default" elt
                -- to return for an unmapped key

--      LISTIFYING
fmToList        :: FiniteMap key elt -> [(key,elt)]
keysFM          :: FiniteMap key elt -> [key]
```

```
eltsFM          :: FiniteMap key elt -> [elt]
```

6.6.3 The `ListSetOps` type

Just a few set-sounding operations on lists. If you want sets, use the `Set` module.

```
unionLists          :: Eq a => [a] -> [a] -> [a]
intersectLists      :: Eq a => [a] -> [a] -> [a]
minusList           :: Eq a => [a] -> [a] -> [a]
disjointLists       :: Eq a => [a] -> [a] -> Bool
intersectingLists   :: Eq a => [a] -> [a] -> Bool
```

6.6.4 The `Maybes` type

The `Maybe` type is in the Haskell 1.4 prelude. Moreover, the required `Maybe` library provides many useful functions on `Maybe`s. This (pre-1.3) module provides some more:

An `Either`-like type called `MaybeErr`:

```
data MaybeErr val err = Succeeded val | Failed err
```

Some operations to do with `Maybe` (some commentary follows):

```
maybeToBool :: Maybe a -> Bool      -- Nothing => False; Just => True
allMaybes   :: [Maybe a] -> Maybe [a]
firstJust   :: [Maybe a] -> Maybe a
findJust    :: (a -> Maybe b) -> [a] -> Maybe b

assocMaybe  :: Eq a => [(a,b)] -> a -> Maybe b
mkLookupFun :: (key -> key -> Bool) -- Equality predicate
            -> [(key,val)]          -- The assoc list
            -> (key -> Maybe val)   -- A lookup fun to use
mkLookupFunDef :: (key -> key -> Bool)  -- Equality predicate
               -> [(key,val)]           -- The assoc list
               -> val                   -- Value to return on failure
               -> key                   -- The key
               -> val                   -- The corresponding value

    -- a monad thing
thenMaybe   :: Maybe a -> (a -> Maybe b) -> Maybe b
returnMaybe :: a -> Maybe a
failMaybe   :: Maybe a
mapMaybe    :: (a -> Maybe b) -> [a] -> Maybe [b]
```

NB: `catMaybes` which used to be here, is now available via the standard `Maybe` interface (`Maybe` is an instance of `MonadPlus`).

`allMaybes` collects a list of `Just`s into a single `Just`, returning `Nothing` if there are any `Nothing`s.

`firstJust` takes a list of `Maybe`s and returns the first `Just` if there is one, or `Nothing` otherwise.

`assocMaybe` looks up in an association list, returning `Nothing` if it fails.

Now, some operations to do with `MaybeErr` (comments follow):

```
    -- a monad thing (surprise, surprise)
thenMaB   :: MaybeErr a err -> (a -> MaybeErr b err) -> MaybeErr b err
returnMaB :: val -> MaybeErr val err
failMaB   :: err -> MaybeErr val err

listMaybeErrs :: [MaybeErr val err] -> MaybeErr [val] [err]
foldlMaybeErrs :: (acc -> input -> MaybeErr acc err)
               -> acc
               -> [input]
               -> MaybeErr acc [err]
```

`listMaybeErrs` takes a list of `MaybeErrs` and, if they all succeed, returns a `Succeeded` of a list of their values. If any fail, it returns a `Failed` of the list of all the errors in the list.

`foldlMaybeErrs` works along a list, carrying an accumulator; it applies the given function to the accumulator and the next list item, accumulating any errors that occur.

6.6.5 The `Memo` library

The `Memo` library provides fast polymorphic memo functions using hash tables. The interface is:

```
memo :: (a -> b) -> a -> b
```

So, for example, `memo f` is a version of `f` that caches the results of previous calls.

The searching is very fast, being based on pointer equality. One consequence of this is that the caching will only be effective if *exactly the same argument is passed again to the memoised function*. This means not just a copy of a previous argument, but the same instance. It's not useful to memoise integer functions using this interface, because integers are generally copied a lot and two instances of '27' are unlikely to refer to the same object.

This memoisation library works well when the keys are large (or even infinite).

The memo table implementation uses weak pointers and stable names (see the GHC/Hugs library document) to avoid space leaks and allow hashing for arbitrary Haskell objects. NOTE: while individual memo table entries will be garbage collected if the associated key becomes garbage, the memo table itself will not be collected if the function becomes garbage. We plan to fix this in a future version.

There's another version of `memo` if you want to explicitly give a size for the hash table (the default size is 1001 buckets):

```
memo_sized :: Int -> (a -> b) -> a -> b
```

6.6.6 The `PackedString` type

You need to `import PackedString` and heave in your `-syslib ghc` to use `PackedStrings`.

The basic type and functions available are:

```
data PackedString -- abstract

packString          :: [Char] -> PackedString
packStringST        :: [Char] -> ST s PackedString
packCBytesST        :: Int -> Addr -> ST s PackedString
packBytesForCST     :: [Char] -> ST s (ByteArray Int)
byteArrayToPS       :: ByteArray Int -> PackedString
unsafeByteArrayToPS :: ByteArray a   -> Int -> PackedString
psToByteArray       :: PackedString -> ByteArray Int
psToByteArrayST     :: PackedString -> ST s (ByteArray Int)

unpackPS        :: PackedString -> [Char]
```

We also provide a wad of list-manipulation-like functions:

```
nilPS       :: PackedString
consPS      :: Char -> PackedString -> PackedString

headPS      :: PackedString -> Char
tailPS      :: PackedString -> PackedString
nullPS      :: PackedString -> Bool
appendPS    :: PackedString -> PackedString -> PackedString
lengthPS    :: PackedString -> Int
indexPS     :: PackedString -> Int -> Char
            -- 0-origin indexing into the string
mapPS       :: (Char -> Char) -> PackedString -> PackedString
filterPS    :: (Char -> Bool) -> PackedString -> PackedString
foldlPS     :: (a -> Char -> a) -> a -> PackedString -> a
foldrPS     :: (Char -> a -> a) -> a -> PackedString -> a
takePS      :: Int -> PackedString -> PackedString
dropPS      :: Int -> PackedString -> PackedString
splitAtPS   :: Int -> PackedString -> (PackedString, PackedString)
takeWhilePS :: (Char -> Bool) -> PackedString -> PackedString
dropWhilePS :: (Char -> Bool) -> PackedString -> PackedString
spanPS      :: (Char -> Bool) -> PackedString -> (PackedString, PackedString)
breakPS     :: (Char -> Bool) -> PackedString -> (PackedString, PackedString)
linesPS     :: PackedString -> [PackedString]
wordsPS     :: PackedString -> [PackedString]
reversePS   :: PackedString -> PackedString
concatPS    :: [PackedString] -> PackedString
elemPS      :: Char -> PackedString -> Bool
  -- Perl-style split&join
splitPS     :: Char -> PackedString -> [PackedString]
splitWithPS :: (Char -> Bool) -> PackedString -> [PackedString]
joinPS      :: PackedString -> [PackedString] -> PackedString

substrPS   :: PackedString -> Int -> Int -> PackedString
           -- pluck out a piece of a PackedString
           -- start and end chars you want; both 0-origin-specified
```

6.6.7 The `Pretty` type

This is the pretty-printer that is currently used in GHC:

```
type Pretty

ppShow          :: Int{-width-} -> Pretty -> [Char]

pp'SP           :: Pretty -- "comma space"
ppComma         :: Pretty -- ,
ppEquals        :: Pretty -- =
ppLbrack        :: Pretty -- [
ppLparen        :: Pretty -- (
ppNil           :: Pretty -- nothing
ppRparen        :: Pretty -- )
ppRbrack        :: Pretty -- ]
ppSP            :: Pretty -- space
ppSemi          :: Pretty -- ;

ppChar          :: Char -> Pretty
ppDouble        :: Double -> Pretty
ppFloat         :: Float -> Pretty
ppInt           :: Int -> Pretty
ppInteger       :: Integer -> Pretty
ppRational      :: Rational -> Pretty
ppStr           :: [Char] -> Pretty

ppAbove         :: Pretty -> Pretty -> Pretty
ppAboves        :: [Pretty] -> Pretty
ppBeside        :: Pretty -> Pretty -> Pretty
ppBesides       :: [Pretty] -> Pretty
ppCat           :: [Pretty] -> Pretty
ppHang          :: Pretty -> Int -> Pretty -> Pretty
ppInterleave    :: Pretty -> [Pretty] -> Pretty -- spacing between
ppIntersperse   :: Pretty -> [Pretty] -> Pretty -- no spacing between
ppNest          :: Int -> Pretty -> Pretty
ppSep           :: [Pretty] -> Pretty
```

6.6.8 The `Set` type

Our implementation of *sets* (key property: no duplicates) is just a variant of the `FiniteMap` module.

```
data Set        -- abstract
                -- instance of: Eq

emptySet        :: Set a
mkSet           :: Ord a => [a]  -> Set a
setToList       :: Set a -> [a]
unitSet         :: a -> Set a
singletonSet    :: a -> Set a  -- deprecated, use unitSet.
```

```
union           :: Ord a => Set a -> Set a -> Set a
unionManySets   :: Ord a => [Set a] -> Set a
minusSet        :: Ord a => Set a -> Set a -> Set a
mapSet          :: Ord a => (b -> a) -> Set b -> Set a
intersect       :: Ord a => Set a -> Set a -> Set a

elementOf       :: Ord a => a -> Set a -> Bool
isEmptySet      :: Set a -> Bool

cardinality     :: Set a -> Int
```

6.6.9 The `BitSet` interface

Bit sets are a fast implementation of sets of integers ranging from 0 to one less than the number of bits in a machine word (typically 31). If any element exceeds the maximum value for a particular machine architecture, the results of these operations are undefined. You have been warned.

```
data BitSet   -- abstract
              -- instance of:

emptyBS       :: BitSet
mkBS          :: [Int] -> BitSet
unitBS        :: Int -> BitSet
unionBS       :: BitSet -> BitSet -> BitSet
minusBS       :: BitSet -> BitSet -> BitSet
isEmptyBS     :: BitSet -> Bool
intersectBS   :: BitSet -> BitSet -> BitSet
elementBS     :: Int -> BitSet -> Bool
listBS        :: BitSet -> [Int]
```

6.6.10 The `Util` type

Stuff that has been generally useful to use in writing the compiler. Don't be too surprised if this stuff moves/gets-renamed/etc.

```
-- general list processing
forall          :: (a -> Bool) -> [a] -> Bool
exists          :: (a -> Bool) -> [a] -> Bool

nOfThem         :: Int -> a -> [a]
lengthExceeds   :: [a] -> Int -> Bool
isSingleton     :: [a] -> Bool

--paranoid zip'ing (equal length lists)
zipEqual        :: [a] -> [b] -> [(a,b)]
zipWithEqual    :: String -> (a->b->c) -> [a]->[b]->[c]
zipWith3Equal   :: String -> (a->b->c->d) -> [a]->[b]->[c]->[d]
zipWith4Equal   :: String -> (a->b->c->d->e) -> [a]->[b]->[c]->[d]->[e]
```

```
-- lazy in second argument
zipLazy :: [a] -> [b] -> [(a,b)]

mapAndUnzip :: (a -> (b, c)) -> [a] -> ([b], [c])
mapAndUnzip3 :: (a -> (b, c, d)) -> [a] -> ([b], [c], [d])

-- prefix and suffix matching on lists of characters.
startsWith :: {-prefix-}String -> String -> Maybe String
endsWith   :: {-suffix-}String -> String -> Maybe String

-- association lists
assoc       :: Eq a => String -> [(a, b)] -> a -> b

-- duplicate handling
hasNoDups    :: Eq a => [a] -> Bool
equivClasses :: (a -> a -> Ordering) -> [a] -> [[a]]
runs         :: (a -> a -> Bool)     -> [a] -> [[a]]
removeDups   :: (a -> a -> Ordering) -> [a] -> ([a], [[a]])

-- sorting (don't complain of no choice...)
quicksort          :: (a -> a -> Bool)     -> [a] -> [a]
sortLt             :: (a -> a -> Bool)     -> [a] -> [a]
stableSortLt       :: (a -> a -> Bool)     -> [a] -> [a]
mergesort          :: (a -> a -> _CMP_TAG) -> [a] -> [a]
mergeSort          :: Ord a => [a] -> [a]
naturalMergeSort   :: Ord a => [a] -> [a]
mergeSortLe        :: Ord a => [a] -> [a]
naturalMergeSortLe :: Ord a => [a] -> [a]

-- transitive closures
transitiveClosure :: (a -> [a])         -- Successor function
                  -> (a -> a -> Bool)   -- Equality predicate
                  -> [a]
                  -> [a]                -- The transitive closure

-- accumulating (Left, Right, Bi-directional)
mapAccumL :: (acc -> x -> (acc, y))
                        -- Function of elt of input list and
                        -- accumulator, returning new accumulator and
                        -- elt of result list
          -> acc        -- Initial accumulator
          -> [x]        -- Input list
          -> (acc, [y]) -- Final accumulator and result list

mapAccumR :: (acc -> x -> (acc, y)) -> acc -> [x] -> (acc, [y])

mapAccumB :: (accl -> accr -> x -> (accl, accr,y))
          -> accl -> accr -> [x]
          -> (accl, accr, [y])

--list comparison with explicit element comparer.
```

```
cmpList :: (a -> a -> Ordering) -> [a] -> [a] -> Ordering

-- pairs
applyToPair :: ((a -> c), (b -> d)) -> (a, b) -> (c, d)
applyToFst  :: (a -> c) -> (a, b) -> (c, b)
applyToSnd  :: (b -> d) -> (a, b) -> (a, d)
foldPair    :: (a->a->a, b->b->b) -> (a, b) -> [(a, b)] -> (a, b)
unzipWith   :: (a -> b -> c) -> [(a, b)] -> [c]
```

6.7 Interfaces to C libraries

The GHC system library (`-syslib misc`) also provides interfaces to several useful C libraries, mostly from the GNU project.

6.7.1 The `Readline` interface

(Darren Moffat supplied the `Readline` interface.)

The `Readline` module is a straightforward interface to the GNU Readline library. As such, you will need to look at the GNU documentation (and have a `libreadline.a` file around somewhere...)

You'll need to link any Readlining program with `-lreadline -ltermcap`, besides the usual `-syslib ghc` (and `-fhaskell-1.3`).

The main function you'll use is:

```
readline :: String{-the prompt-} -> IO String
```

If you want to mess around with Full Readline G(l)ory, we also provide:

```
rlInitialize, addHistory,

rlBindKey, rlAddDefun, RlCallbackFunction(..),

rlGetLineBuffer, rlSetLineBuffer, rlGetPoint, rlSetPoint, rlGetEnd,
rlSetEnd, rlGetMark, rlSetMark, rlSetDone, rlPendingInput,

rlPrompt, rlTerminalName, rlSetReadlineName, rlGetReadlineName
```

(All those names are just Haskellised versions of what you will see in the GNU readline documentation.)

6.7.2 The `Regex` and `MatchPS` interfaces

(Sigbjorn Finne supplied the regular-expressions interface.)

The `Regex` library provides quite direct interface to the GNU regular-expression library, for doing manipulation on `PackedString`s. You probably need to see the GNU documentation if you are operating at this level. Alternatively, you can use the simpler and higher-level `RegexString` interface.

The datatypes and functions that `Regex` provides are:

```
data PatBuffer  # just a bunch of bytes (mutable)

data REmatch
 = REmatch (Array Int GroupBounds)  -- for $1, ... $n
           GroupBounds              -- for $` (everything before match)
           GroupBounds              -- for $& (entire matched string)
           GroupBounds              -- for $' (everything after)
           GroupBounds              -- for $+ (matched by last bracket)

-- GroupBounds hold the interval where a group
-- matched inside a string, e.g.
--
-- matching "reg(exp)" "a regexp" returns the pair (5,7) for the
-- (exp) group. (PackedString indices start from 0)

type GroupBounds = (Int, Int)

re_compile_pattern
        :: PackedString         -- pattern to compile
        -> Bool                 -- True <=> assume single-line mode
        -> Bool                 -- True <=> case-insensitive
        -> PrimIO PatBuffer

re_match :: PatBuffer           -- compiled regexp
         -> PackedString        -- string to match
         -> Int                 -- start position
         -> Bool                -- True <=> record results in registers
         -> PrimIO (Maybe REmatch)

-- Matching on 2 strings is useful when you're dealing with multiple
-- buffers, which is something that could prove useful for
-- PackedStrings, as we don't want to stuff the contents of a file
-- into one massive heap chunk, but load (smaller chunks) on demand.

re_match2 :: PatBuffer          -- 2-string version
          -> PackedString
          -> PackedString
          -> Int
          -> Int
          -> Bool
          -> PrimIO (Maybe REmatch)

re_search :: PatBuffer          -- compiled regexp
          -> PackedString       -- string to search
          -> Int                -- start index
          -> Int                -- stop index
          -> Bool               -- True <=> record results in registers
          -> PrimIO (Maybe REmatch)

re_search2 :: PatBuffer         -- Double buffer search
           -> PackedString
```

```
            -> PackedString
            -> Int              -- start index
            -> Int              -- range (?)
            -> Int              -- stop index
            -> Bool             -- True <=> results in registers
            -> PrimIO (Maybe REmatch)
```

The `MatchPS` module provides Perl-like "higher-level" facilities to operate on `PackedStrings`. The regular expressions in question are in Perl syntax. The "flags" on various functions can include: `i` for case-insensitive, `s` for single-line mode, and `g` for global. (It's probably worth your time to peruse the source code...)

```
matchPS :: PackedString    -- regexp
        -> PackedString    -- string to match
        -> [Char]          -- flags
        -> Maybe REmatch   -- info about what matched and where

searchPS :: PackedString    -- regexp
         -> PackedString    -- string to match
         -> [Char]          -- flags
         -> Maybe REmatch

-- Perl-like match-and-substitute:
substPS :: PackedString     -- regexp
        -> PackedString     -- replacement
        -> [Char]           -- flags
        -> PackedString     -- string
        -> PackedString

-- same as substPS, but no prefix and suffix:
replacePS :: PackedString  -- regexp
          -> PackedString  -- replacement
          -> [Char]        -- flags
          -> PackedString  -- string
          -> PackedString

match2PS :: PackedString   -- regexp
         -> PackedString   -- string1 to match
         -> PackedString   -- string2 to match
         -> [Char]         -- flags
         -> Maybe REmatch

search2PS :: PackedString  -- regexp
          -> PackedString  -- string to match
          -> PackedString  -- string to match
          -> [Char]        -- flags
          -> Maybe REmatch

-- functions to pull the matched pieces out of an REmatch:

getMatchesNo    :: REmatch -> Int
getMatchedGroup :: REmatch -> Int -> PackedString -> PackedString
```

```
getWholeMatch   :: REmatch -> PackedString -> PackedString
getLastMatch    :: REmatch -> PackedString -> PackedString
getAfterMatch   :: REmatch -> PackedString -> PackedString

-- (reverse) brute-force string matching;
-- Perl equivalent is index/rindex:
findPS, rfindPS :: PackedString -> PackedString -> Maybe Int

-- Equivalent to Perl "chop" (off the last character, if any):
chopPS :: PackedString -> PackedString

-- matchPrefixPS: tries to match as much as possible of strA starting
-- from the beginning of strB (handy when matching fancy literals in
-- parsers):
matchPrefixPS :: PackedString -> PackedString -> Int
```

6.7.3 The `RegexString` interface

(Simon Marlow supplied the String Regex wrapper.)

For simple regular expression operations, the `Regex` library is a little heavyweight. `RegexString` permits regex matching on ordinary Haskell `String`s.

The datatypes and functions that `RegexString` provides are:

```
data Regex              -- a compiled regular expression

mkRegEx
        :: String       -- regexp to compile
        -> Regex        -- compiled regexp

matchRegex
        :: Regex        -- compiled regexp
        -> String       -- string to match
        -> Maybe [String] -- text of $1, $2, ... (if matched)
```

6.7.4 Network-interface toolkit—`Socket` and `SocketPrim`

(Darren Moffat supplied the network-interface toolkit.)

Your best bet for documentation is to look at the code—really!— normally in `fptools/ghc/lib/misc/{BSD,Socket,SocketPrim.lhs}`.

The `BSD` module provides functions to get at system-database info; pretty straightforward if you're into this sort of thing:

```
getHostName         :: IO String

getServiceByName    :: ServiceName -> IO ServiceEntry
getServicePortNumber:: ServiceName -> IO PortNumber
getServiceEntry     :: IO ServiceEntry
setServiceEntry     :: Bool -> IO ()
```

```
endServiceEntry     :: IO ()

getProtocolByName   :: ProtocolName -> IO ProtocolEntry
getProtocolByNumber :: ProtocolNumber -> IO ProtcolEntry
getProtocolNumber   :: ProtocolName -> ProtocolNumber
getProtocolEntry    :: IO ProtocolEntry
setProtocolEntry    :: Bool -> IO ()
endProtocolEntry    :: IO ()

getHostByName       :: HostName -> IO HostEntry
getHostByAddr       :: Family -> HostAddress -> IO HostEntry
getHostEntry        :: IO HostEntry
setHostEntry        :: Bool -> IO ()
endHostEntry        :: IO ()
```

The `SocketPrim` interface provides quite direct access to the socket facilities in a BSD Unix system, including all the complications. We hope you don't need to use it! See the source if needed...

The `Socket` interface is a "higher-level" interface to sockets, and it is what we recommend. Please tell us if the facilities it offers are inadequate to your task!

The interface is relatively modest:

```
connectTo       :: Hostname -> PortID -> IO Handle
listenOn        :: PortID -> IO Socket

accept          :: Socket -> IO (Handle, HostName)
sendTo          :: Hostname -> PortID -> String -> IO ()

recvFrom        :: Hostname -> PortID -> IO String
socketPort      :: Socket -> IO PortID

data PortID     -- PortID is a non-abstract type
  = Service String        -- Service Name eg "ftp"
  | PortNumber PortNumber -- User defined Port Number
  | UnixSocket String     -- Unix family socket in file system

type Hostname = String

 -- 16-bit value (stored in network byte order).
data PortNumber
 -- instance of: Eq, Num, Show.

mkPortNumber :: Int -> PortNumber
```

Various examples of networking Haskell code are provided in

7 What to do when something goes wrong

If you still have a problem after consulting this section, then you may have found a *bug*—please report it! See Section 7.3 (How to report a bug in the GHC system) for a list of things we'd like to know about your

bug. If in doubt, send a report—we love mail from irate users :-!

(Section 5.13 (Haskell 98 vs. Glasgow Haskell: language non-compliance), which describes Glasgow Haskell's shortcomings vs. the Haskell language definition, may also be of interest.)

7.1 When the compiler "does the wrong thing"

"Help! The compiler crashed (or 'panic'd)!"

These events are *always* bugs in the GHC system—please report them.

"The compiler ran out of heap (or stack) when compiling itself!"

It happens. We try to supply reasonable `-H<n>` flags for `ghc/compiler/` and `ghc/lib/`, but GHC's memory consumption can vary by platform (e.g., on a 64-bit machine).

Just say `make all EXTRA_HC_OPTS=-H<a reasonable number>` and see how you get along.

Note that this is less likely to happen if you are compiling with GHC 4.00 or later, since the introduction of the dynamically expanding heap.

"The compiler died with a pattern-matching error."

This is a bug just as surely as a "panic." Please report it.

"This is a terrible error message."

If you think that GHC could have produced a better error message, please report it as a bug.

"What about these 'trace' messages from GHC?"

Almost surely not a problem. About some specific cases...

Simplifier still going after N iterations:

Sad, but harmless. You can change the number with a `-fmax-simplifier-iterations<N>` option (no space); and you can see what actions took place in each iteration by turning on the `-fshow-simplifier-progress` option.

If the simplifier definitely seems to be "looping," please report it.

"What about this warning from the C compiler?"

For example: "...warning: 'Foo' declared 'static' but never defined." Unsightly, but shouldn't be a problem.

Sensitivity to `.hi` interface files:

GHC is very sensitive about interface files. For example, if it picks up a non-standard `Prelude.hi` file, pretty terrible things will happen. If you turn on `-fno-implicit-prelude`, the compiler will almost surely die, unless you know what you are doing.

Furthermore, as sketched below, you may have big problems running programs compiled using unstable interfaces.

"I think GHC is producing incorrect code":

Unlikely :-) A useful be-more-paranoid option to give to GHC is `-dcore-lint`; this causes a "lint" pass to check for errors (notably type errors) after each Core-to-Core transformation pass. We run with `-dcore-lint` on all the time; it costs about 5% in compile time.

"Why did I get a link error?"

If the linker complains about not finding `_<something>_fast`, then something is inconsistent: you probably didn't compile modules in the proper dependency order.

"What's a 'consistency error'?"

(These are reported just after linking your program.)

You tried to link incompatible object files, e.g., normal ones (registerised, Appel garbage-collector) with profiling ones (two-space collector). Or those compiled by a previous version of GHC with an incompatible newer version.

If you run `nm -o *.o | egrep 't (cc|hsc)\.'` (or, on unregisterised files: `what *.o`), you'll see all the consistency tags/strings in your object files. They must all be the same! (ToDo: tell you what they mean...)

"Is this line number right?"

On this score, GHC usually does pretty well, especially if you "allow" it to be off by one or two. In the case of an instance or class declaration, the line number may only point you to the declaration, not to a specific method.

Please report line-number errors that you find particularly unhelpful.

7.2 When your program "does the wrong thing"

(For advice about overly slow or memory-hungry Haskell programs, please see Section 4 (Advice on: sooner, faster, smaller, stingier)).

"Help! My program crashed!"

(e.g., a 'segmentation fault' or 'core dumped')

If your program has no `_ccall_`s/`_casm_`s in it, then a crash is always a BUG in the GHC system, except in one case: If your program is made of several modules, each module must have been compiled after any modules on which it depends (unless you use `.hi-boot` files, in which case these *must* be correct with respect to the module source).

For example, if an interface is lying about the type of an imported value then GHC may well generate duff code for the importing module. *This applies to pragmas inside interfaces too!* If the pragma is lying (e.g., about the "arity" of a value), then duff code may result. Furthermore, arities may change even if types do not.

In short, if you compile a module and its interface changes, then all the modules that import that interface *must* be re-compiled.

A useful option to alert you when interfaces change is `-hi-diffs`. It will run `diff` on the changed interface file, before and after, when applicable.

If you are using `make`, a useful tool to make sure that every module *is* up-to-date with respect to its imported interfaces is `mkdependHS` (which comes with GHC). Please see Section 8.1 (Makefile dependencies in Haskell: using mkdependHS).

If you are down to your last-compile-before-a-bug-report, we would recommend that you add a `-dcore-lint` option (for extra checking) to your compilation options.

So, before you report a bug because of a core dump, you should probably:

```
% rm *.o        # scrub your object files
% make my_prog  # re-make your program; use -hi-diffs to highlight changes;
                # as mentioned above, use -dcore-lint to be more paranoid
% ./my_prog ... # retry...
```

Of course, if you have `_ccall_`s/`_casm_`s in your program then all bets are off, because you can trash the heap, the stack, or whatever.

If you are interested in hard-core debugging of a crashing GHC-compiled program, please see Section 7.4 (Hard-core debugging of GHC-compiled programs).

"My program entered an 'absent' argument."

This is definitely caused by a bug in GHC. Please report it.

"What's with this 'arithmetic (or 'floating') exception' "?

`Int`, `Float`, and `Double` arithmetic is *unchecked*. Overflows, underflows and loss of precision are either silent or reported as an exception by the operating system (depending on the architecture). Divide-by-zero *may* cause an untrapped exception (please report it if it does).

7.3 How to report a bug in the GHC system

Glasgow Haskell is a changing system so there are sure to be bugs in it. Please report them to *glasgow-haskell-bugs@haskell.org*! (However, please check the earlier part of this section to be sure it's not a known not-really-a problem.)

The name of the bug-reporting game is: facts, facts, facts. Don't omit them because "Oh, they won't be interested..."

1. What kind of machine are you running on, and exactly what version of the operating system are you using? (`uname -a` or `cat /etc/motd` will show the desired information.)
2. What version of GCC are you using? `gcc -v` will tell you.
3. Run the sequence of compiles/runs that caused the offending behaviour, capturing all the input/output in a "script" (a UNIX command) or in an Emacs shell window. We'd prefer to see the whole thing.
4. Be sure any Haskell compilations are run with a `-v` (verbose) flag, so we can see exactly what was run, what versions of things you have, etc.
5. What is the program behaviour that is wrong, in your opinion?
6. If practical, please send enough source files/interface files for us to duplicate the problem.
7. If you are a Hero and track down the problem in the compilation-system sources, please send us patches relative to a known released version of GHC, or whole files if you prefer.

7.4 Hard-core debugging of GHC-compiled programs

If your program is crashing, you should almost surely file a bug report, as outlined in previous sections.

This section suggests ways to Make Further Progress Anyway.

The first thing to establish is: Is it a garbage-collection (GC) bug? Try your program with a very large heap and a `-Sstderr` RTS flag.

- If it crashes *without* garbage-collecting, then it is definitely *not* a GC bug.
- If you can make it crash with one heap size but not with another, then it *probably is* a GC bug.
- If it crashes with the normal collector, but not when you force two-space collection (`-F2s` runtime flag), then it *probably is* a GC bug.

If it *is* a GC bug, you may be able to avoid it by using a particular heap size or by using a `-F2s` runtime flag. (But don't forget to report the bug!!!)

ToDo: more here?

8 Other Haskell utility programs

This section describes other program(s) which we distribute, that help with the Great Haskell Programming Task.

8.1 Makefile dependencies in Haskell: using `mkdependHS`

You run `mkdependHS` like this:

```
mkdependHS [mkdependHS options] [-- GHC options --] srcfile1 [srcfile2 ...]
```

or

```
ghc -M [mkdependHS options(prefix with -optdep)] [ GHC options ] srcfile1 [srcfile2 ...]
```

To see `mkdependHS`'s command-line flags, give it a duff flag, e.g., `mkdependHS -help`.

In general, if module `A` contains the line

```
import B ...blah...
```

then `mkdependHS` will generate a dependency line of the form:

```
A.o : B.hi
```

If module `A` contains the line

```
import {-# SOURCE #-} B ...blah...
```

then `mkdependHS` will generate a dependency line of the form:

```
A.o : B.hi-boot
```

(See Section 2.7.1 (Interface files) for details of interface files.) If `A` imports multiple modules, then there will be multiple lines with `A.o` as the target.

By default, `mkdependHS` generates all the dependencies, and then concatenates them onto the end of `makefile` (or `Makefile` if `makefile` doesn't exist) bracketed by the lines `"# DO NOT DELETE: Beginning of Haskell dependencies"` and `"# DO NOT DELETE: End of Haskell dependencies"`. If these lines already exist in the `makefile`, `mkdependHS` deletes the old dependencies first.

`mkdependHS` takes GHC options between `-` brackets. It understands the following ones. Any options between `-` brackets that it doesn't understand are simply ignored; this way you can feed your Makefile's standard GHC options to `mkdependHS` un-filtered.

`-cpp`

Run the C pre-processor over the input files. The default is not to.

`-D<blah>`

A cpp `#define`; usual meaning.

`-i<dirs>`

Add `<dirs>` (colon-separated) to list of directories to search for "import"ed modules.

`-I<dir>`

Add `<dir>` to list of directories to search for .h files (i.e., usual meaning).

`-syslib <blah>`

This program uses this GHC system library; take appropriate action (e.g., recognise when they are "import"ing a module from that library).

Here are the `mkdependHS`-specific options (not between `-`'s):

`-v`

Be verbose.

`-v -v`

Be very verbose.

`-w`

Turn off warnings about interface file shadowing.

`-f blah`

Use `blah` as the makefile, rather than `makefile` or `Makefile`. If `blah` doesn't exist, `mkdependHS` creates it. We often use `-f .depend` to put the dependencies in `.depend` and then `include` the file `.depend` into `Makefilpe`.

`-o <osuf>`

Use `.<osuf>` as the "target file" suffix (default: `o`). Multiple `-o` flags are permitted (GHC2.05 onwards). Thus `"-o hc -o o"` will generate dependencies for `.hc` and `.o` files.

`-s <suf>`

Make extra dependencies that declare that files with suffix `.<suf>_<osuf>` depend on interface files with suffix `.<suf>_hi`, or (for {`-# SOURCE #-`} imports) on `.hi-boot`. Multiple `-s` flags are permitted. For example, "`-o hc -s a -s b`" will make dependencies for `.hc` on `.hi`, `.a_hc` on `.a_hi`, and `.b_hc` on `.b_hi`. (Useful in conjunction with NoFib "ways".)

`-exclude-module=<file>`

Regard `<file>` as "stable"; i.e., exclude it from having dependencies on it.

`-x`

same as `-exclude-module`

`-exclude-directory=<dirs>`

Regard the colon-separated list of directories `<dirs>` as containing stable, don't generate any dependencies on modules therein.

`-Xdirs`

same as `-exclude-directory`.

`-include-module=<file>`

Regard `<file>` as not "stable"; i.e., generate dependencies on it (if any). This option is normally used in conjunction with the `-exclude-directory` option.

`-include-prelude`

Regard prelude libraries as unstable, i.e., generate dependencies on the prelude modules used (including `Prelude`). This option is normally only used by the various system libraries. If a `-syslib` option is used, dependencies will also be generated on the library's interfaces.

8.2 Emacs 'TAGS' for Haskell: `hstags`

'Tags' is a facility for indexing the definitions of programming-language things in a multi-file program, and then using that index to jump around among these definitions.

Rather than scratch your head, saying "Now where did we define 'foo'?", you just do (in Emacs) `M-. foo RET`, and You're There! Some people go wild over this stuff...

GHC comes with a program `hstags`, which build Emacs-able TAGS files. The invocation syntax is:

```
hstags [GHC-options] file [files...]
```

The best thing is just to feed it your GHC command-line flags. A good Makefile entry might be:

```
tags:
        $(RM) TAGS
        hstags $(GHC_FLAGS) *.lhs
```

The only flags of its own are: `-v` to be verbose; `-a` to **APPEND** to the TAGS file, rather than write to it.

Shortcomings: (1) Instance declarations don't get into the TAGS file (but the definitions inside them do); as instances aren't named, this is probably just as well. (2) Data-constructor definitions don't get in. Go for the corresponding type constructor instead.

(Actually, GHC also comes with `etags` [for C], and `perltags` [for You Know What]. And—I cannot tell a lie—there is Denis Howe's `fptags` [for Haskell, etc.] in the `ghc/CONTRIB` section...)

8.3 "Yacc for Haskell": `happy`

Andy Gill and Simon Marlow have written a parser-generator for Haskell, called `happy`. `Happy` is to Haskell what `Yacc` is to C.

You can get `happy` by FTP from `ftp.dcs.gla.ac.uk` in `pub/haskell/happy`, the file `happy-1.5-src.tar.gz`.

`Happy` is at its shining best when compiled by GHC.

8.4 Pretty-printing Haskell: `pphs`

Andrew Preece has written `pphs`, a utility to pretty-print Haskell code in LaTeX documents. Keywords in bolds, variables in italics—that sort of thing. It is good at lining up program clauses and equals signs, things that are very tiresome to do by hand.

The code is distributed with GHC in `ghc/CONTRIB/pphs`. %**********
%* *

9 Building and using Win32 DLLs

%* * %**********

On Win32 platforms, the compiler is capable of both producing and using dynamic link libraries (DLLs) containing ghc-compiled code. This section shows you how to make use of this facility.

%********** %* *

9.1 Linking with DLLs

%* * %**********

The default on Win32 platforms is to link applications in such a way that the executables will use the Prelude and system libraries DLLs, rather than contain (large chunks of) them. This is transparent at the command-line, so

```
sh$ cat main.hs
module Main where
main = putStrLn "hello, world!"
sh$ ghc -o main main.hs
ghc: module version changed to 1; reason: no old .hi file
sh$ strip main.exe
```

```
sh$ ls -l main.exe
-rwxr-xr-x   1 544      everyone     6144 May  3 17:11 main.exe*
sh$ ./main
hello, world!
sh$
```

will give you a binary as before, but the `main.exe` generated will use the Prelude and RTS DLLs instead.

6K for a `"hello, world"` application - not bad, huh? :-)

%** %* *

9.2 Not linking with DLLs

%* * %**

If you want to build an executable that doesn't depend on any ghc-compiled DLLs, use the `-static` option to link in the code statically.

Notice that you cannot mix code that has been compiled with `-static` and not, so you have to use the `-static` option on all the Haskell modules that make up your application.

%** %* *

9.3 Creating a DLL

%* * %**

Sealing up your Haskell library inside a DLL is quite straightforward; compile up the object files that make up the library, and then build the DLL by issuing the following command:

```
sh$ ghc --mk-dll -o HSsuper.dll A.o Super.o B.o libmine.a -lgdi32
```

By feeding the ghc compiler driver the option `-mk-dll`, it will build a DLL rather than produce an executable. The DLL will consist of all the object files and archives given on the command line.

A couple of things to notice:

- When compiling the module `A`, the code emitted by the compiler differs depending on whether or not the functions and data it is importing from other Haskell modules correspond to symbols that are packaged up in a ghc-compiled DLL. To resolve whether such imports are 'DLL imports' or not, the following rules are used:
  - If the compiler imports from a module that's in the same directory as the one being compiled, it is assumed to not belong to a different DLL (or executable) than the module being processed, so none of the same-directory imports are considered 'DLL imports'.
  - If a directory contains the (probably empty) file `dLL_ifs.hi`, the code corresponding to the interface files found in that directory are assumed to live in a DLL separate from the one being compiled.

    Notice that the first rule takes precedence over this one, so if you're compiling a module that imports from a Haskell module whose interface file live in the same directory, *and* that directory also contains the file `dLL_ifs.hi`, the import is still not being considered to be a 'DLL import'.

– If compiling with the option `-static`, the previous rule is disabled.

So, in short, after having built your Haskell DLL, make sure you create the file `dLL_ifs.hi` in the directory that contains its interface files. If you don't, Haskell code that calls upon entry points in that DLL, will do so incorrectly, and a crash will result. (it is unfortunate that this isn't currently caught at compile-time).

- By default, the entry points of all the object files will be exported from the DLL when using `-mk-dll`. Should you want to constrain this, you can specify the *module definition file* to use on the command line as follows:

```
sh$ ghc --mk-dll -o .... -optdll--def -optdllMyDef.def
```

See Microsoft documentation for details, but a module definition file simply lists what entry points you want to export. Here's one that's suitable when building a Haskell COM server DLL:

```
EXPORTS
 DllCanUnloadNow     = DllCanUnloadNow@0
 DllGetClassObject   = DllGetClassObject@12
 DllRegisterServer   = DllRegisterServer@0
 DllUnregisterServer = DllUnregisterServer@0
```

- In addition to creating a DLL, the `-mk-dll` option will also create an import library. The import library name is derived from the name of the DLL, as follows:

```
DLL: HScool.dll  ==> import lib: libHScool_imp.a
```

The naming scheme may look a bit weird, but it has the purpose of allowing the co-existence of import libraries with ordinary static libraries (e.g., `libHSfoo.a` and `libHSfoo_imp.a`.

Additionally, when the compiler driver is linking in non-static mode, it will rewrite occurrence of `-lHSfoo` on the command line to `-lHSfoo_imp`. By doing this for you, switching from non-static to static linking is simply a question of adding `-static` to your command line.