Hugs 1.4

The Nottingham and Yale Haskell User's System

User Manual

Mark P. Jones
Department of Computer Science
The University of Nottingham
Nottingham NG7 2RD, England
Technical Report
NOTTCS-TR-97-1

John C. Peterson
Department of Computer Science
Yale University
New Haven CT, 06520-8285, USA
Research Report
YALEU/DCS/RR-1123

April 1997

Contents

Conditions of use, duplication and distribution

1. Introduction

Hugs 1.4 is a functional programming system based on Haskell, the de facto standard for non-strict functional programming languages. This manual should give you all the information that you need to start using Hugs. However, it is not intended as a tutorial on either functional programming in general or on Haskell in particular.

The first two sections provide introductory material:

- Section 2: A brief technical summary of the main features of Hugs 1.4, and the ways that it differs from previous releases.
- Section 3: A short tutorial on the concepts that you need to understand to be able to use Hugs.

The remaining sections provide reference material, including:

- Section 4: A summary of the command line syntax, environment variables, and command line options used by Hugs.
- Section 5: A summary of commands that can be used within the interpreter.
- Section 6: An overview of the Hugs libraries.
- Section 7: Information about other ways of running Hugs programs.
- Section 8: A list of differences between Hugs 1.4 and standard Haskell.
- Section 9: Pointers to further information.

Whether you are a beginner or a seasoned old-timer, we hope that you will enjoy working with Hugs, and that, if you will pardon the pun, you will use it to embrace functional programming!

Acknowledgements: The development of Hugs has benefited considerably from the feedback, suggestions, and bug reports provided by its users. There are too many people to name here, but thanks are due for all of their contributions. A special thank you also to our friends and colleagues in the functional programming groups at Nottingham and at Yale for their input to the current release.

2. A technical summary of Hugs 1.4

Hugs 1.4 provides an almost complete implementation of Haskell 1.4 [7], including:

- Lazy evaluation, higher order functions, and pattern matching.
- A wide range of built-in types, from characters to bignums, and lists to functions, with comprehensive facilities for defining new datatypes and type synonyms.
- An advanced polymorphic type system with type and constructor class overloading.
- All of the features of the Haskell 1.4 expression and pattern syntax including lambda, case, conditional and let expressions, list comprehensions, do-notation, operator sections, and wildcard, irrefutable and 'as' patterns.
- An implementation of the main Haskell 1.4 primitives for monadic I/O, with support for simple interactive programs, access to text files, handle-based I/O, and exception handling.
- An almost complete implementation of the Haskell module system. The primary omission is that mutually recursive modules are not yet supported.

Hugs is implemented as an interpreter that provides:

- A relatively small, portable system that can be used on a range of different machines, from home computers, to Unix workstations.
- A read-eval-print loop for displaying the value of each expression that is entered into the interpreter.
- Fast loading, type checking, and compilation of Haskell programs, with facilities for automatic loading of imported modules.
- Integration with an external editor, chosen by the user, to allow for rapid development, and for location of errors.
- Modest browsing facilities that can be used to find information about the operations and types that are available.

Hugs is a successor to Gofer — an experimental functional programming system that was first released in September 1991 — and users of Gofer will see much that is familiar in Hugs. However, Hugs offers much greater compatibility with the Haskell standard; indeed, the name *Hugs* was originally chosen as a mnemonic for the "*Haskell users' Gofer system.*"

There have been many modifications and enhancements to Hugs since its first release on Valentines day, February 14, in 1995. Some of the most obvious improvements include:

- Full support for new Haskell 1.3 and 1.4 features, including the labelled field syntax, do-notation, newtype, strictness annotations in datatypes, the `Eval` class, ISO character set, etc.
- Support for Haskell modules, and a growing collection of library modules, that includes facilities for X window and Win32 programming.
- User interface enhancements, particularly the import chasing and search pathfeatures, which were motivated by a greater emphasis on the role of libraries in Haskell 1.4.
- Small improvements in runtime performance, and more reliable space usage, thanks to the use of non-conservative garbage collection during program execution.
- A graphical user interface for the Hugs systems that runs on the Windows operating system.

There have also been a number of other enhancements, and fixes for bugs in previous releases, some more serious than others.

3. *Hugs for beginners*

This section covers the basics that you need to understand to start using Hugs. Most of the points discussed here will be familiar to anyone with experience of previous versions of Hugs or Gofer. To begin with, we need to start the interpreter; the usual way to do this is by using the command `hugs`, which produces a startup banner something like the following[1]:

```
     ___    ___   ___    ___   __________   __________
    /  /   /  /  /  /   /  /  /  _______/  /  _______/        Hugs 1.4
   /  /___/  /  /  /   /  /  /  / _____   /  /______  The Nottingham and Yale
  /  ____   /  /  /   /  /  /  / /_   /  /______   /   Haskell User's System
 /  /   /  /  /  /___/  /  /  /___/  /  _______/  /       February, 1997
/__/   /__/  /_________/  /_________/  /_________/

  Copyright (c) The University of Nottingham and Yale University, 1994-1997.
   Bug reports: hugs-bugs@haskell.org.   Web: http://www.haskell.org/hugs.

Reading script file "/Hugs/lib/Prelude.hs":

Hugs session for:
/Hugs/lib/Prelude.hs
```

The file `/Hugs/lib/Prelude.hs` mentioned here contains standard definitions that are loaded into Hugs each time that the interpreter is started; the filename will vary from one installation to the next[2]. You may notice a pause while the interpreter is initialized and the prelude definitions are loaded into the system.

3.1 *Expressions*

In essence, using Hugs is just like using a calculator; the interpreter simply evaluates each expression that is entered, printing the results as it goes.

```
Prelude> (2+3)*8
40
```

[1] On Windows 95/NT, the installation procedure adds Hugs to the start menu. You can also start the interpreter by double clicking on a `.hs` or `.lhs` file.

[2] If Hugs does not load correctly, and complains that it cannot find the prelude, then Hugs has not been installed correctly and you should look at the installation instructions.

```
Prelude> sum [1..10]
55
Prelude>
```

The `Prelude>` characters at the begining of the first, third and fifth lines here is the Hugs prompt. This indicates that the system is ready to accept input from the user, and that it will use definitions from the module `Prelude` module to evaluate each expression that is entered; The Hugs prelude is a special module that contains definitions for the built-in operations of Haskell, such as `+`, `*`, and `sum`. In response to the first prompt, the user entered the expression `(2+3)*8`, which was evaluated to produce the result `40`. In response to the second prompt, the user typed the expression `sum [1..10]`. The notation `[1..10]` represents the list of integers between 1 and 10 inclusive, and `sum` is a prelude function that calculates the sum of a list of numbers. So the result obtained by Hugs is:

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10  =  55.
```

In fact, we could have typed this sum directly into Hugs:

```
Prelude> 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
55
Prelude>
```

Unlike many calculators, however, Hugs is not limited to working with numbers; expressions can involve many different types of value, including numbers, booleans, characters, strings, lists, functions, and user-defined datatypes. Some of these are illustrated in the following example:

```
Prelude> (not True) || False
False
Prelude> reverse "Hugs is cool"
"looc si sguH"
Prelude> filter even [1..10]
[2, 4, 6, 8, 10]
Prelude> take 10 fibs where fibs = 0:1:zipWith (+) fibs (tail fibs)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Prelude>
```

You cannot create new definitions at the command prompt—these must be placed in files and loaded, as described later. The definition of `fib` in the last example above is local to that expression and will not be remembered for later use. Also, the expressions entered must fit on a single line.

Hugs even allows whole programs to be used as values in calculations. For example, `putStr "hello, "` is a simple program that outputs the string `"hello, "`.

Combining this with a similar program to print the string `"world"`, gives:

```
Prelude> putStr "hello, " >> putStr "world"
hello, world
Prelude>
```

Just as there are standard operations for dealing with numbers, so there are standard operations for dealing with programs. For example, the `>>` operator used here constructs a new program from the programs supplied as its operands, running one after the other. Normally, Hugs just prints the value of each expression entered. But, as this example shows, if the expression evaluates to a program, then Hugs will run it instead. Hugs distinguishes programs from other expressions by looking at the type of the expression entered. The type of the `putStr` function, `IO ()`, identifies it as a program to be executed rather than a value to be printed.

3.2 Commands

Each line that you enter in response to the Hugs prompt is treated as a command to the interpreter. For example, when you enter an expression into Hugs, it is treated as a command to evaluate that expression, and to display the result. There are two commands that are particularly worth remembering:

- `:q` exits the interpreter. On most systems, you can also terminate Hugs by typing the end-of-file character.
- `:?` prints a list of all the commands, which can be useful if you forget the name of the command that you want to use.

Like most other commands in Hugs, these commands both start with a colon, `:`. The full set of Hugs commands is described in Section 5.

Note that the interrupt key (control-C or control-Break on most systems) can be used to abandon the process of compiling files or evaluating expressions. When the interrupt is detected, Hugs prints `{Interrupted!}` and returns to the prompt so that further commands can be entered.

3.3 Programs

Functions like `sum`, `>>` and `take`, used in the examples above, are all defined in the Hugs prelude; you can actually do quite a lot using just the types and operations provided by the prelude. But, in general, you will also want to define new types and operations, storing them in modules that can be loaded and used by Hugs. A module is simply a collection of definitions stored in a file. For example, suppose we enter the following module:

```
module Fact where
fact  :: Integer -> Integer
fact n = product [1..n]
```

into a file called `Fact.hs`. (By convention, Hugs modules are stored in files ending with the characters `.hs`. The file name should match the name of the module it contains.) The `product` function used here is also defined in the prelude, and can be used to calculate the product of a list of numbers, just as you might use `sum` to calculate the corresponding sum. So the line above defines a function `fact` that takes an argument `n` and calculates its factorial. In standard mathematical notation, `fact n = n!`, which is usually defined by an equation:

```
n! = 1 * 2 * ... * (n-1) * n
```

Once you become familiar with the notation, you will see that the Hugs definition is really very similar to this informal, mathematical version: the factorial of a number `n` is the product of the numbers from 1 to `n`.

Before we can use this definition in a Hugs session, we have to load `Fact.hs` into the interpreter. One of the simplest ways to do this uses the `:load` command:

```
Prelude> :load fact.hs
Reading file "fact.hs":

Hugs session for:
/Hugs/lib/Prelude.hs
Fact.hs
Fact>
```

Notice the list of filenames displayed after `Hugs session for:`; this tells you which module files are currently being used by Hugs, the first of which is always the standard prelude. The prompt is now `Fact` and evaluation will take place within this new module. We can start to use the `fact` function that we have defined:

```
Fact> fact 6
720
Fact> fact 6 + fact 7
5760
Fact> fact 7 `div` fact 6
7
Fact>
```

As another example, the standard formula for the number of different ways of choosing `r` objects from a collection of `n` objects is `n! 'div' (r! * (n-r)!)`. One simple and direct (but otherwise not particularly good) definition for this function in Hugs is as follows:

```
comb n r = fact n 'div' (fact r * fact (n-r))
```

One way to use this function is to include its definition as part of an expression entered in directly to Hugs:

```
Fact> comb 5 2 where comb n r = fact n 'div' (fact r * fact (n-r))
10
Fact>
```

The definition of `comb` here is local to this expression. If we want to use `comb` several times, then it would be sensible to add its definition to the file `Fact.hs`. Once this has been done, and the `Fact.hs` file has been reloaded, then we can use the `comb` function like any other built-in operator:

```
Fact> :reload
Reading file "fact.hs":

Hugs session for:
/Hugs/lib/Prelude.hs
Fact.hs
Fact> comb 5 2
10
Fact>
```

4. Starting Hugs

On Unix machines, the Hugs interpreter is usually started with a command line of the form:

```
hugs [option | file] ...
```

On Windows 95/NT, Hugs may be started by selecting it from the start menu or by double clicking on a file with the `.hs` or `.lhs` extension. (This manual assumes that Hugs has already been successfully installed on your system.)

Hugs uses *options* to set system parameters. These options are distinguished by a leading `+` or `-` and are used to customize the behaviour of the interpreter. When Hugs starts, the interpreter performs the following tasks:

- Options in the environment are processed. The variable `HUGSFLAGS` holds these options. On Windows 95/NT, the registry is queried if `HUGSFLAGS` is undefined.
- Command line options are processed.
- Internal data structures are initialized. In particular, the heap is initialized, and its size is fixed at this point; if you want to run the interpreter with a heap size other than the default, then this must be specified using options on the command line, in the environment or in the registry.
- The prelude file is loaded. The interpreter will look for the prelude file on the path specified by the `-P` option. If the prelude, located in the file `Prelude.hs`, cannot be found in one of the path directories or in the current directory, then Hugs will terminate; Hugs will not run without the prelude file.
- Program files specified on the command line are loaded. The effect of a command `hugs f1 ... fn` is the same as starting up Hugs with the `hugs` command and then typing `:load f1 ... fn`. In particular, the interpreter will not terminate if a problem occurs while it is trying to load one of the specified files, but it will abort the attempted load command.

The environment variables and command line options used by Hugs are described in the following sections.

4.1 Environment options

Before options on the command line are processed, initial option values are set from the environment. On Windows 95/NT, these settings are added to the registry during setup. On other systems, the initial settings are determined by the `HUGSFLAGS` environment variable. The syntax used in this case is the same as on the command line: options are single letters, preceeded by `+` or `-`, and sometimes followed by a value. Option settings are separated by spaces; option values containing spaces are encoded using Haskell string syntax. The environment should be set up before the interpreter is used so that the search path is correctly defined to include the prelude. The built-in defaults, however, may allow Hugs to be run without any help from the environment on some systems.

It is usually more convenient to save preferred option settings in the environment rather than specifying them on the command line; they will then be used automatically each time the interpreter is started. The method for setting these options depends on the machine and operating system that you are using, and on the way that the Hugs system was installed. The following examples show some typical settings for Unix machines and PCs:

- The method for setting `HUGSFLAGS` on a Unix machine depends on the choice of shell. For example, a C-shell user might add something like the following to their `.cshrc` file:

  ```
  set HUGSFLAGS -P/usr/Hugs/lib:/usr/Hugs/libhugs -E"vi +%d %s"
  ```

  The `P` option is used to set the search path and the `E` is used to set the editor. The string quotes are necessary for the value of the `E` option becauses it contains spaces. The setting for the path assumes that the system has been installed in `/usr/local/Hugs` and will need to be modified accordingly if a different directory was chosen. The editor specified here is `vi`, which allows the user to specify a startup line number by preceding it with a `+` character. The settings are easily changed to accommodate other editors.

  If you are installing Hugs for the benefit of several different users, then you should probably use a script file that sets appropriate values for the environment variables, and then invokes the interpreter:

  ```
  #!/bin/sh
  HUGSFLAGS=/usr/Hugs/lib:/usr/Hugs/libhugs -E"vi +%d %s" +s
  export HUGSFLAGS
  exec /usr/local/bin/hugs $*
  ```

  One advantage of this approach is that individual users do not have to worry about setting the environment variables themselves. In addition to the E and P options, other options—such as +s in this example—can be set. It is easy for more advanced users to copy and customize a script like this to suit their own needs.

- Users of DOS or Windows 3.1 might add the following line to autoexec.bat:

  ```
  set HUGSFLAGS=-P\hugs\lib;\hugs\libhugs -E"vi +%%d %%s"
  ```

  The setting for the path assumes that the system has been installed in a top-level hugs directory, and will need to be modified accordingly if a different directory was chosen. In a similar way, the setting for the editor will only work if you have installed the editor program, in this case vi, that it refers to.

- On Windows 95/NT, the setup program initializes the environment, and this can be changed subsequently (on these systems only) by using either the :set command or a registry editor. The InstallShield script that performs the installation initializes the path using the installation directory; other directories can be added using -P. Installed options are stored under the HKEY_LOCAL_MACHINE key; changes to these options using :set are placed under HKEY_CURRENT_USER so that different users do not alter each other's options.

For completeness, we should also mention the other environment variables that are used by Hugs:

- The SHELL variable on a Unix machine, or the COMSPEC variable on a DOS machine, determines which shell is used by the :! command.

- The EDITOR variable is used to try and locate an editor if no editor option has been set. Note, however, that this variable does not normally provide the extra information that is needed to be able to start the editor at a specific line in the input file.

4.2 Options

The behaviour of the interpreter, particularly the read-eval-print loop, can be customized using options. For example, you might use:

```
hugs -i +g +h30K
```

to start the interpreter with the i option (import chasing) disabled, the g option (garbage collector messages) enabled, and with a heap of thirty thousand cells.

As this example suggests, many of the options are toggles, meaning that they can either be switched on (by preceding the option with a `+` character) or off (by using a `-` character). Options may also be grouped together. For example, `hugs +stf -le` is equivalent to `hugs +s +t +f -l -e`.

Option settings can be specified in a number of different ways—the `HUGSFLAGS` environment variable, the Windows registry, the command line, and the `:set` command—but the same syntax is used in each case. To avoid any confusion with filenames entered on the command line, option settings must always begin with a leading `+` or `-` character. However, in some cases—the `h`, `p`, `r`, `P`, and `E` options—the choice is not significant. With the exception of the heap size option, `h`, all options can be changed while the interpreter is running using the `:set` command. The same command can be used (without any arguments) to display a summary of the available options and to inspect their current settings.

The complete set of Hugs options is described in the sections below.

Set search path	`-P`⟨*path*⟩

The `-P`⟨*path*⟩ option changes the Hugs search path to the specified ⟨*path*⟩. The search path is usually initialized in the environment and should always include the directory containing the Hugs prelude and the standard libraries. When an unknown module is imported, Hugs searches for a file with the same name as the module along this path. The current directory is always searched before the path is used. Directory names should be separated by colons or, on Windows/DOS machines, by semicolons. Empty components in the path refer to the prior value of the path. For example, setting the path to `dir:` (`dir;` on Windows/DOS) would add `dir` to the front of the current path. Within the path, `{Hugs}` refers to the directory containing the Hugs libraries so one might use a path such as `{Hugs}/lib:{Hugs}/lib/hugs`.

Set editor	`-E`⟨*cmd*⟩

A `-E`⟨*cmd*⟩ option can be used to change the editor string to the specified ⟨*cmd*⟩ while the interpreter is running. The editor string is usually initialized from the environment when the interpreter starts running.

Any occurrences of `%d` and `%s` in the editor option are replaced by the start line number and the name of the file to be edited, respectively, when the editor is invoked. If specified, the line number parameter is used to let the interpreter start the editor at the line where an error was detected, or, in the case of the `:find` command, where a specified variable was defined.

Other editors can be selected. For example, you can use the following value to configure Hugs to use `emacs`:

```
-E"emacs +%d %s"
```

More commonly, `emacsclient` or `gnuclient` is used to avoid starting a new `emacs` with every edit.

On Windows/DOS, you can use `-Eedit` for the standard DOS editor, or `-Enotepad` for the Windows notepad editor. However, neither `edit` or `notepad` allow you to specify a start line number, so you may prefer to install a different editor.

Print statistics	`+s,-s`

Normally, Hugs just shows the result of evaluating each expression:

```
Prelude> map (\x -> x*x) [1..10]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Prelude> [1..]
[1, 2, 3, 4, {Interrupted!}
Prelude>
```

With the `+s` option, the interpreter will also display statistics about the total number of *reductions* and *cells*; the former gives a measure of the work done, while the latter gives an indication of the amount of memory used. For example:

```
Prelude> :set +s
Prelude> map (\x -> x*x) [1..10]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
(248 reductions, 429 cells)
Prelude> [1..]
[1, 2, 3, 4, {Interrupted!}
(18 reductions, 54 cells)
Prelude>
```

Note that the statistics produced by `+s` are an extremely crude measure of the behaviour of a program, and can easily be misinterpreted. For example:

- The fact that one expression requires more reductions than another does not necessarily mean that the first is slower; some reductions require much more work than others, and it may be that the average cost of reductions in the first expression is much lower than the average for the second.

- The cell count does not give any information about *residency*, which is the number of cells that are being used at any given time. For example, it does not distinguish between computations that run in constant space and computations with residency proportional to the size of the input.

One reasonable use of the statistics produced by `+s` would be to observe general trends in the behaviour of a single algorithm with variations in its input.

Print type after evaluation	`+t`,`-t`

With the `+t` option, the interpreter will display both the result and type of each expression entered at the Hugs prompt:

```
Prelude> :set +t
Prelude> map (\x -> x*x) [1..10]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100] :: [Int]
Prelude> not True
False :: Bool
Prelude> \x -> x
<<function>> :: a -> a
Prelude>
```

Note that the interpreter will not display the type of an expression if its evaluation is interrupted or fails with a run-time error. In addition, the interpreter will not print the type, `IO ()`, of a program in the `IO` monad; the interpreter treats these as a special case, giving the programmer more control over the output that is produced.

Terminate on error	`+f`,`-f`

In normal use, the evaluation of an expression is abandoned completely if a run-time error occurs, such as a failed pattern match or an attempt to divide by zero. For example:

```
Prelude> [1 `div` 0]
[
Program error: {primDivInt 1 0}

Prelude> [1 `div` 0, 2]
[
Program error: {primDivInt 1 0}

Prelude>
```

This is often useful during program development because it means that errors are detected as soon as they occur. However, technically speaking, the two expressions above have different meanings; the first is a singleton list, while the second has two elements. Unfortunately, the output produced by Hugs does not allow us to distinguish between the values.

The -f option can be used to make the Hugs printing option a little more accurate; this should normally be combined with -u because the built-in printer is better than the user-defined show functions at recovering from evaluation errors. With these settings, if the interpreter encounters an irreducible subexpression, then it prints the expression between a matching pair of braces and attempts to continue with the evaluation of other parts of the original expression. For the examples above, we get:

```
Prelude> :set -u -f
Prelude> [1 `div` 0]        -- value is [bottom]
[{primDivInt 1 0}]
Prelude> [1 `div` 0, 2]
[{primDivInt 1 0}, 2]       -- value is [bottom, 2]
Prelude>
```

Reading an expression in braces as bottom, $\perp$, the output produced here shows the correct values, according to the semantics of Haskell. Of course, it is not possible to detect all occurrences of bottom like this, such as those produced by a nonterminating computation:

```
Prelude> last [1..]
^C{Interrupted!}      -- nothing printed until interrupted

Prelude>
```

Note that the basic method of evaluation is the same with both the +f and -f options; all that changes is the way that the printing mechanism deals with certain kinds of runtime error.

Garbage collector notification	+g,-g

It is sometimes useful to monitor uses of the garbage collector, and to determine how many cells are recovered with each collection. If the +g option is set, then the interpreter will print a message of the form {{Gc:num}} each time that the garbage collector is invoked. The number after the colon indicates the total number of cells that are recovered.

As a simple application, we can use garbage collector messages to observe that an attempt to sum an infinite list, although non-terminating, will at least run in constant space:

```
Prelude> :set +g
Prelude> sum [1..]
{{Gc:95763}}{{Gc:95760}}{{Gc:95760}}{{Gc:95760}}{Interrupted!}

Prelude>
```

Garbage collector messages may be printed at almost any stage in a computation (or indeed while loading, type checking or compiling a file of definitions). For this reason, it is often best to turn garbage collector messages off (using `:set -g`, for example) if they are not required.

Literate modules	`+l,-l,+e,-e`

Like most programming languages, Hugs usually treats source file input as a sequence of lines in which program text is the norm, and comments play a secondary role. In Hugs, as in Haskell, comments are introduced by the character sequences `--` and `{- ... -}`.

An alternative approach, using an idea described by Knuth as "literate programming," gives more emphasis to comments and documentation, with additional characters needed to distinguish program text from comments. Hugs supports a form of literate programming based on an idea due to Richard Bird and originally implemented as part of the functional programming language Orwell.

In a Hugs literate module, program lines are marked by a `>` character in the first column; any other line is treated as a program comment. This makes it particularly easy to write a document which is both an executable Hugs module and, at the same time, without need for any preprocessing, suitable for use with document preparation software such as LaTeX.

Hugs will treat any input file with a name ending in `.hs` as a normal module and any input file with a name ending in `.lhs` as a literate module. If the `-l` option is selected, then any other file loaded into Hugs will be treated as a normal module. Conversely, if `+l` is selected, then these files will be treated as literate modules.

The effect of using literate modules can be thought of as applying a preprocessor to each input file that is loaded into Hugs. This has a particularly simple definition in Hugs:

```
illiterate   :: String -> String
illiterate cs = unlines [ " " ++ xs | ('>':xs) <- lines cs ]
```

The system of literate modules that was used in Orwell is a little more complicated than this and requires the programmer to adopt two further conventions in an attempt to catch simple errors in literate modules:

- Every input file must contain at least one line whose first character is `>`. This prevents modules with no definitions (because the programmer has forgotten to use the `>` character to mark definitions) from being accepted.

- Lines containing definitions must be separated from comment lines by one or more blank lines (i.e., lines containing only space and tab characters). This is useful for catching programs where the leading `>` character has been omitted from one or more lines in the definition of a function. For example:

  ```
  > map f []     = []
    map f (x:xs) = f x : map f xs
  ```

  would be treated as an error.

Hugs will report on errors of this kind whenever the `-e` option is enabled (the default setting).

The Haskell Report defines a second style of literate programming in which code is surrounded by `\begin{code}` and `\end{code}`. See Appendix C of the Haskell Report for more information about literate programming in Haskell.

List files loaded	`+w`,`-w`

By default, Hugs prints a complete list of all the files that have been loaded into the system after every successful load or reload command. The `-w` option can be used to turn this feature off. Note that the `:info` command, without any arguments, can also be used to list the names of currently loaded files.

Display dots while loading	`+.`,`-.`

As Hugs loads each file into the interpreter, it prints a short sequence of messages to indicate progress through the various stages of parsing the module, dependency analysis, type checking, and compilation. With the default setting, `-.`, the interpreter prints the name of each stage, backspacing over it to erase it from the screen when the stage is complete. If you are fortunate enough to be using a fast machine, you may not always see the individual words as they flash past. After loading a file, your screen will typically look something like this:

```
Prelude> :l Array
Reading file "/Hugs/lib/Array.hs":

Hugs session for:
/Hugs/lib/Prelude.hs
/Hugs/lib/Array.hs
Prelude>
```

On some systems, the use of backspace characters to erase a line may not work properly—for example, if you try to run Hugs from within `emacs`. In this case, you may prefer to use the `+.` setting which prints a separate line for each stage, with a row of dots to indicate progress:

```
Prelude> :load Array
Reading file "/Hugs/lib/Array.hs":
Parsing.................................................
Dependency analysis.....................................
Type checking...........................................
Compiling...............................................

Hugs session for:
/Hugs/lib/Prelude.hs
/Hugs/lib/Array.hs
Prelude>
```

This setting can also be useful on very slow machines where the growing line of dots provides confirmation that the interpreter is making progress through the various stages involved in loading a file. You should note, however, that the mechanisms used to display the rows of dots can add a substantial overhead to the time that it takes to load files; in one experiment, a particular program took nearly five times longer to load when the `+.` option was used.

Use "show" to display results	`+u,-u`

In normal use, Hugs displays the value of each expression entered into the interpreter by applying the standard prelude function:

```
show :: Show a => a -> String
```

to it and displaying the resulting string of characters. This approach works well for any value whose type is an instance of the standard `Show` class; for example, the prelude defines instances of `Show` for all of the built-in datatypes. It is also easy for users to extend the class with new datatypes, either by providing a handwritten instance declaration, or by requesting an automatically derived instance as part of the datatype definition, as in:

```
data Rainbow = Red | Orange | Yellow | Green | Blue | Indigo | Violet
               deriving Show
```

The advantage of using `show` is that it allows programmers to display the results of evaluations in whatever form is most convenient for users—which is not always the same as the way in which the values are represented.

This is probably all that most users will ever need. However, there are some circumstances where it is not convenient, for example, for certain kinds of debugging or for work with datatypes that are not instances of `Show`. In these situations, the `-u` option can be used to prevent the use of `show`. In its place, Hugs will use a built-in printing mechanism that works for *all* datatypes, and uses the representation of a value to determine what gets printed. At any point, the default printing mechanism can be restored by setting `+u`.

Detailed kind errors	`+k,-k`

Haskell uses a system of kinds to ensure that type expressions are well-formed: for example, to make sure that each type constructor is applied to the appropriate number of arguments. For example, the following program:

```
module Main where
data Tree a  = Leaf a | Tree a :^: Tree a
type Example = Tree Int Bool
```

will cause an error:

```
ERROR "Main.hs" (line 3): Illegal type "Tree Int Bool" in
                          constructor application
```

The problem here is that `Tree` is a unary constructor of kind `* -> *`, but the definition of `Example` uses it as a binary constructor with at least two arguments, and hence expecting a kind of the form `(* -> * -> k)`, for some kind `k`.

By default, Hugs reports problems like this with a simple message like the one shown above. However, if the `+k` option is selected, then the interpreter will print a more detailed version of the error message, including details about the kinds of the type expressions that are involved:

```
ERROR "Main.hs" (line 3): Kind error in constructor application
*** expression     : Tree Int Bool
*** constructor    : Tree
*** kind           : * -> *
*** does not match : * -> a -> b
```

In addition, if the `+k` option is used, then Hugs will also include information about kinds in the information produced by the `:info` command:

```
Prelude> :info Tree
-- type constructor with kind * -> *
data Tree a

-- constructors:
Leaf :: a -> Tree a
(:^:) :: Tree a -> Tree a -> Tree a
```

```
-- instances:
instance Eval (Tree a)

Prelude>
```

Import chasing	+i,-i

Import chasing is a simple, but flexible mechanism for dealing with programs that involve multiple modules. It works in a natural way, using the information in import statements at the beginning of modules, and is particularly useful for large programs, or for programs that use standard Hugs libraries.

For example, consider a module `Demo.hs` that requires the facilities provided by the `STArray` library. This dependency might be reflected by including the following import statement at the beginning of `Demo.hs`:

```
import STArray
```

Now, if we try to load this module into Hugs, then the system will automatically search for the `STArray` library and load it into Hugs, before `Demo.hs` is loaded. In fact, the `STArray` library module also begins with some import statements:

```
import ST
import Array
```

So, Hugs will actually load the `ST` and `Array` libraries first, then the `STArray` library, and only then will it try to read the rest of `Demo.hs`:

```
Prelude> :load Demo
Reading file "Demo.hs":
Reading file "/hugs/libhugs/STArray.hs":
Reading file "/hugs/libhugs/ST.hs":
Reading file "/hugs/lib/Array.hs":
Reading file "/hugs/libhugs/STArray.hs":
Reading file "Demo.hs":
Demo>
```

Initially, the interpreter reads only the first part of any module loaded into the system, upto and including any import statements. Only one module is allowed in each file; files with no module declaration are assumed to declare the `Main` module. If there are no imports, or if the modules specified as imports have already been loaded, then the system carries on and loads the module as normal. On the other hand, if the module includes import statements for modules that have not already been loaded, then the interpreter postpones the task of reading the current module until all of the specified imports have been successfully loaded.

This explains why `Demo.hs` and `STArray.hs` are read twice in the example above; first to determine which imports are required, and then to read in the rest of the file once the necessary imports have been loaded.

The list of directories and filenames that Hugs tries in an attempt to locate the source for a module `Mod` named in an import statement can be specified by:

```
[ (dir,"Mod"++suf) | dir <- [d] ++ path ++ [""],
                     suf <- ["", ".hs", ".lhs"]]
```

The search starts in the directory `d` where the file containing the import statement was found, then tries each of the directories in the current path (as defined by the `-P` option), represented here by `path`, and ends with `""`, which gives a search relative to the current directory. The fact that the search starts in `d` is particularly important because it means that you can load a multi-file program into Hugs without having to change to the directory where its source code is located. For example, suppose that `/tmp` contains the files, `A.hs`, `B.hs`, and `C.hs`, that `B` imports `A`, and that `C` imports `B`. Now, regardless of the current working directory, you can load the whole program with the command `:load /tmp/C`; the import in `C` will be taken as a reference to `/tmp/B.hs`, while the import in that file will be taken as a reference to `/tmp/A.hs`.

Import chasing is often very useful, but you should also be aware of its limitations:

- Mutually recursive modules are not supported; if `A` imports `B`, then `B` must not import `A`, either directly or indirectly through another one of its imports.
- Import chasing assumes a direct mapping from module names to the names of the files that they are stored in. If `A` imports `B`, then the code for `B` must be in a file called either `B`, `B.hs`, or `B.lhs`, and must be located in one of the directories specified above.

  On rare occasions, it is useful to specify a particular pathname as the target for an import statement; Hugs allows string literals to be used as module identifiers for this purpose:

  ```
  import "../TypeChecker/Types.hs"
  ```

  Note, however, that this is a nonstandard feature of Hugs, and that it is not valid Haskell syntax. You should also be aware that Hugs uses the names of files in deciding whether a particular import has already been loaded, so you should avoid situations where a single file is referred to by more than one name. For example, you should not assume that Hugs will be able to determine whether `Demo.hs` and `./Demo.hs` are references to the same file.

Import chasing is usually enabled by default (setting `+i`), but it can also be disabled using the `-i` option.

Set heap size	`-h`⟨*size*⟩

A `-h`⟨*size*⟩ option can be used to request a particular heap size for the interpreter—the total number of cells that are available at any one time—when Hugs is first loaded. The request will only be honoured if it falls within a certain range, which depends on the machine, and the version of Hugs that is used. The ⟨*size*⟩ parameter may include a `K` or `k` suffix, which acts as a multiplier by 1,000. For example, either of the following commands:

```
hugs -h25000
hugs -h25K
```

will usually start the Hugs interpreter with a heap of 25,000 cells. Cells are generally 8 bytes wide (except on the 16 bit Hugs running on DOS) and Hugs allocates a single heap. Note that the heap is used to hold an intermediate (parsed) form of each module while it is being read, type checked and compiled. It follows that, the larger the module, the larger the heap required to enable that module to be loaded into Hugs. In practice, most large programs are written (and loaded) as a number of separate modules which means that this does not usually cause problems.

Unlike all of the other options described here, the heap size setting cannot be changed from within the interpreter using a `:set` command. However, on Window 95/NT, changing the heap size with `:set` will affect the next running of Hugs since it saves all options in the registry.

Set prompt	`-p`⟨*string*⟩

A `-p`⟨*str*⟩ option can be used to change the prompt to the specified string, ⟨*str*⟩:

```
Prelude> :set -p"Hugs> "
Hugs> :set -p"? "
?
```

Note that you will need to use quotes around the prompt string if you want to include spaces or special characters. Any `%s` in the prompt will be replaced by the current module name. The default prompt is `"%s> "`.

Set repeat string	`-r`⟨*string*⟩

Hugs allows the user to recall the last expression entered into the interpreter by typing the characters `$$` as part of the next expression:

```
Prelude> map (1+) [1..10]
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
Prelude> filter even $$
[2, 4, 6, 8, 10]
Prelude>
```

A `-r`⟨*str*⟩ option can be used to change the repeat string—the symbol used to recall the last expression—to ⟨*str*⟩. For example, users of Standard ML might be more comfortable using:

```
Prelude> :set -rit
Prelude> 6 * 7
42
Prelude> it + it
84
Prelude>
```

Another reason to change the repeat string is to avoid clashes with uses of the same symbol in a particular program; for example, if `$$` is defined as an operator in a program.

Note that the repeat string must be a valid Haskell identifier or symbol, although it will always be parsed as an identifier. If the repeat string is set to a value that is neither an identifier or symbol (for example, `-r0`), then the repeat last expression facility will be disabled.

5. Hugs commands

Hugs provides a number of commands that can be used to evaluate expressions, to load files, and to inspect or modify the behaviour of the system while the interpreter is running. Almost all of the commands in Hugs begin with the : character, followed by a short command word. For convenience, all but the first letter of a command may be omitted. For example, `:l`, `:s` and `:q` can be used as abbreviations for the `:load`, `:set` and `:quit` commands, respectively.

Most Hugs commands take arguments, separated from the command itself, and from one another, by spaces. The Haskell syntax for string constants can be used to enter parts of arguments that contain spaces, newlines, or other special characters. For example, the command:

```
:load My File
```

will be treated as a command to load two files, `My` and `File`. Any of the following commands can be used to load a single file, `My File`, whose name includes an embedded space:

```
:load "My File"
:load "My\SPFile"
:load "My\  \ File"
:load My" "File
```

You may wish to study the lexical syntax of Haskell strings to understand some of these examples. In practice, filenames do not usually include spaces or special characters and can be entered without surrounding quotes, as in:

```
:load fact.hs
```

The full set of Hugs commands is described in the following sections.

5.1 Basic commands

Evaluate expression	$\langle expr \rangle$

To evaluate an expression, the user simply enters it at the Hugs prompt. This is treated as a special case, without the leading colon that is required for other commands. The expression must fit on a single line; there is no way to continue an expression onto the next line of input to the interpreter. The actual behaviour of the evaluator depends on the type of ⟨*expr*⟩:

- If ⟨*expr*⟩ has type `IO ()`, then it will be treated as a program using the I/O facilities provided by the Haskell `IO` monad.

  ```
  Prelude> putStr "Hello, world"
  Hello, world
  Prelude>
  ```

- In any other case, the value produced by the expression is converted to a string by applying the `show` function from the standard prelude, and the interpreter uses this to print the result.

  ```
  Prelude> "Hello" ++ ", " ++ "world"
  "Hello, world"
  Prelude>
  ```

  Unlike some previous versions of Hugs, there is no special treatment for values of type `String`; to display a string without the enclosing quotes and special escapes, you should turn it into a program using the `putStr` function, as shown above.

The interpreter will not evaluate an expression that contains a syntax error, a type error, or a reference to an undefined variable:

```
Prelude> sum [1..)
ERROR: Syntax error in expression (unexpected `)')
Prelude> sum 'a'
ERROR: Type error in application
*** expression     : sum 'a'
*** term           : 'a'
*** type           : Char
*** does not match : [a]
Prelude> sum [1..n]
ERROR: Undefined variable "n"
Prelude>
```

Another common problem occurs if there is no `show` function for the expression entered—that is, if its type is not an instance of the `Show` class. For example, suppose that a module defines a type `T` without a `Show` instance:

```
module Test where
data T = A | B
```

With just these definitions, any attempt to evaluate an expression of type `T` will cause an error:

```
Test> A
ERROR: Cannot find "show" function for:
*** expression : A
*** of type    : T
Test>
```

To avoid problems like this, you will need to add an instance of the `Show` class to your program. One of the simplest ways to do that is to request a derived instance of `Show` as part of the datatype definition, as in:

```
module Test where
data T = A | B  deriving Show
```

Once this has been loaded, Hugs will evaluate and display values of type `T`:

```
Test> A
A
Test> take 5 (cycle [A,B])
[A, B, A, B, A]
Test>
```

Values in the `IO` monad are only treated as programs if they return `()`. For example, `getChar` has type `IO Char`, so it is printed using `show`:

```
Prelude> getChar
<<IO Action>>
Prelude>
```

Hugs will not execute this expression as a program because it does not specify what should be done with the character returned by `getChar`. If you want to run `getChar` as a program and, for example, discard its result, then you must do this explicitly:

```
Prelude> do getChar; return ()
w
Prelude>
```

You should also note that the behaviour of the evaluator can be changed while the interpreter is running by using the `:set` command to modify option settings.

View or change settings	`:set [`⟨*options*⟩`]`

Without any arguments, the `:set` command displays a list of the options and their current settings. The following output shows the default settings on a typical machine:

```
Prelude> :set
TOGGLES: groups begin with +/- to turn options on/off resp.
s    Print no. reductions/cells after eval
t    Print type after evaluation
f    Terminate evaluation on first error
g    Print no. cells recovered after gc
l    Literate files as default
e    Warn about errors in literate files
.    Print dots to show progress
w    Always show which files loaded
k    Show kind errors in full
u    Use "show" to display results
i    Chase imports while loading files

OTHER OPTIONS: (leading + or - makes no difference)
hnum Set heap size (cannot be changed within Hugs)
pstr Set prompt string to str
rstr Set repeat last expression string to str
Pstr Set search path for modules to str
Estr Use editor setting given by str

Current settings: +fekui -stgl.w -h100000 -p"? " -r$$
Search path     : -P/Hugs/lib:/Hugs/libhugs
Editor setting  : -E"vi +%d %s"
Prelude>
```

Refer to Section 4.2 for more detailed descriptions of each of these option settings.

The `:set` command can also be used to change options by supplying the required settings as arguments. For example:

```
Prelude> :set +st
Prelude> 1 + 3
4 :: Int
(4 reductions, 4 cells)
Prelude>
```

On Windows 95/NT, all option settings are written out to the registry when a `:set` command is executed, and will be used by subsequent executions of Hugs.

Shell escape	`:!`[⟨*command*⟩]

A `:!`⟨*cmd*⟩ command can be used to execute the system command ⟨*cmd*⟩ without leaving the Hugs interpreter. For example, `:!ls` (or `:!dir` on DOS machines) can be used to list the contents of the current directory. For convenience, the `:!` command can be abbreviated to a single `!` character.

The :! command, without any arguments, starts a new shell:

- On a Unix machine, the `SHELL` environment variable is used to determine which shell to use; the default is `/bin/sh`.
- On an DOS machine, the `COMSPEC` environment variable is used to determine which shell to use; this is usually `COMMAND.COM`.

Most shells provide an `exit` command to terminate the shell and return to Hugs.

List commands	:?

The :? command displays the following summary of all Hugs commands:

```
Prelude> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>   load modules from specified files
:load               clear all files except prelude
:also <filenames>   read additional module files
:reload             repeat last load command
:project <filename> use project file
:edit <filename>    edit file
:edit               edit last file
:module <module>    set module for evaluating expressions
<expr>              evaluate expression
:type <expr>        print type of expression
:?                  display this list of commands
:set <options>      set options
:set                help on options
:names [pat]        list names currently in scope
:info <names>       describe named objects
:find <name>        edit module containing definition of name
:!command           shell escape
:cd dir             change directory
:gc                 force garbage collection
:quit               exit Hugs interpreter
Prelude>
```

Change module	`:module` ⟨*module*⟩

A `:module` ⟨*module*⟩ command changes the current module to one given by ⟨*module*⟩. This is the module in which evaluation takes place and in which objects named in commands are resolved. The specified module must be part of the current program. If no module is specified, then the last module to be loaded is assumed. (Note that the name of the current module is usually displayed as part of the Hugs prompt.)

Change directory	`:cd` ⟨*directory*⟩

A `:cd` ⟨*dir*⟩ command changes the current working directory to the path given by ⟨*dir*⟩. If no path is specified, then the command is ignored.

Force a garbage collection	`:gc`

A `:gc` command can be used to force a garbage collection of the interpreter heap, and to print the number of unused cells obtained as a result:

```
Prelude> :gc
Garbage collection recovered 95766 cells
Prelude>
```

Exit the interpreter	`:quit`

The `:quit` command terminates the current Hugs session.

5.2 Loading and editing modules and projects

Load definitions from module	`:load [`⟨*filename*⟩ `...]`

The `:load` command removes any previously loaded modules, and then attempts to load the definitions from each of the listed files, one after the other. If one of these files contains an error, then the load process is suspended and a suitable error message will be displayed. Once the problem has been corrected, the load process can be restarted using a `:reload` command. On some systems, the load process will be restarted automatically after a `:edit` command. (The exception occurs on Windows 95/NT because of the way that the interpreter and editor are executed as independent processes.)

If no file names are specified, the `:load` command just removes any previously loaded definitions, leaving just the definitions provided by the prelude.

The `:load` command uses the list of directories specified by the current path to search for module files. We can specify the list of directory and filename pairs, in the order that they are searched, using a Haskell list comprehension:

```
[ (dir,file++suf) | dir <- [""] ++ path, suf <- ["", ".hs", ".lhs"]]
```

The `file` mentioned here is the name of the module file that was entered by the user, while `path` is the current Hugs search path. The search starts with the directory `""`, which usually represents a search relative to the current working directory. So, the very first filename that the system tries to load is *exactly* the same filename entered by the user. However, if the named file cannot be accessed, then the system will try adding a `.hs` suffix, and then a `.lhs` suffix, and then it will repeat the process for each directory in the path, until either a suitable file has been located, or, otherwise, until all of the possible choices have been tried. For example, this means that you do not have to type the `.hs` suffix to load a file `Demo.hs` from the current directory, provided that you do not already have a `Demo` file in the same directory. In the same way, it is not usually necesary to include the full pathname for one of the standard Hugs libraries. For example, provided that you do not have an `Array`, `Array.hs`, or `Array.lhs` file in the current working directory, you can load the standard `Array` library by typing just `:load Array`.

Load additional files	`:also [`⟨*filename*⟩ `...]`

The `:also` command can be used to load module files, without removing any that have previously been loaded. (However, if any of the previously modules have been modified since they were last read, then they will be reloaded automatically before the additional files are read.)

If successful, a command of the form `:load f1 .. fn` is equivalent to the sequence of commands:

```
:load
:also f1
  .
  .
:also fn
```

In particular, `:also` uses the same mechanisms as `:load` to search for modules.

Repeat last load command	`:reload`

The `:reload` command can be used to repeat the last load command. If none of the previously loaded files has been modified since the last time that it was loaded, then `:reload` will not have any effect. However, if one of the modules has been modified, then it will be reloaded. Note that modules are loaded in a specific order, with the possibility that later modules may import earlier ones. To allow for this, if one module has been reloaded, then all subsequent modules will also be reloaded.

This feature is particularly useful in a windowing environment. If the interpreter is running in one window, then `:reload` can be used to force the interpreter to take account of changes made by editing modules in other windows.

Load project	`:project` [⟨*project file*⟩]

Project files were originally introduced to ease the task of working with programs whose source code was spread over several files, all of which had to be loaded at the same time. The new facilities for import chasing usually provide a much better way to deal with multiple file projects, but the current version of Hugs 1.4 does still support the use of project files.

The `:project` command takes a single argument; the name of a text file containing a list of file names, separated from one another by whitespace (which may include spaces, newlines, or Haskell-style comments). For example, the following is a valid project file:

```
{- A simple project file, Demo.prj -}
Types   -- datatype definitions
Basics  -- basic operations
Main    -- the main program
```

If we load this into Hugs with a command `:project Demo.prj`, then the interpreter will read the project file and then try to load each of the named files. In this particular case, the overall effect is, essentially, the same as that of:

```
:load Types Basics Main
```

Once a project file has been selected, the `:project` command (without any arguments) can be used to force Hugs to reread both the project file and the module files that it lists. This might be useful if, for example, the project file itself has been modified since it was first read.

Project file names may also be specified on the command line when the interpreter is invoked by preceding the project file name with a single `+` character. Note that there must be at least one space on each side of the `+`. Standard command line options can also be used at the same time, but additional filename arguments will be ignored. Starting Hugs with a command of the form `hugs + Demo.prj` is equivalent to starting Hugs without any arguments and then giving the command `:p Demo.prj`.

The `:project` command uses the same mechanisms as `:load` to locate the files mentioned in a project file, but it will not use the current path to locate the project file itself; you must specify a full pathname.

As has already been said, import chasing usually provides a much better way to deal with multiple file programs than the old project file system. The big advantage of import chasing is that dependencies between modules are documented within individual modules, leaving the system free to determine the order in which the files should be loaded. For example, if the `Main` module in the example above actually needs the definitions in `Types` and `Basics`, then this will be documented by import statements, and the whole program could be loaded with a single `:load Main` command.

Edit file	`:edit [`⟨*file*⟩`]`

The `:edit` command starts an editor program to modify or view a module file. On Windows 95/NT, the editor and interpreter are executed as independent processes. On other systems, the current Hugs session will be suspended while the editor is running. Then, when the editor terminates, the Hugs session will be resumed and any files that have been changed will be reloaded automatically. The `-E` option should be used to configure Hugs to your preferred choice of editor.

If no filename is specified, then Hugs uses the name of the last file that it tried to load. This allows the `:edit` command to integrate smoothly with the facilities for loading files.

For example, suppose that you want to load four files, `f1.hs`, `f2.hs`, `f3.hs` and `f4.hs` into the interpreter, but the file `f3.hs` contains an error of some kind. If you give the command:

```
:load f1 f2 f3 f4
```

then Hugs will successfully load `f1.hs` and `f2.hs`, but will abort the load command when it encounters the error in `f3.hs`, printing an error message to describe the problem that occured. Now, if you use the command:

```
:edit
```

then Hugs will start up the editor with the cursor positioned at the relevant line of `f3.hs` (whenever this is possible) so that the error can be corrected and the changes saved in `f3.hs`. When you close down the editor and return to Hugs, the interpreter will automatically attempt to reload `f3.hs` and then, if successful, go on to load the next file, `f4.hs`. So, after just two commands in Hugs, the error in `f3.hs` has been corrected and all four of the files listed on the original command line have been loaded into the interpreter, ready for use.

Find definition	`:find` ⟨*name*⟩

The `:find` ⟨*name*⟩ command starts up the editor at the definition of a type constructor or function, specified by the argument ⟨*name*⟩, in one of the files currently loaded into Hugs. Note that Hugs must be configured with an appropriate editor for this to work properly. There are four possibilities:

- If there is a type constructor with the specified name, then the cursor will be positioned at the first line in the definition of that type constructor.
- If the name is defined by a function or variable binding, then the cursor will be positioned at the first line in the definition of the function or variable (ignoring any type declaration, if present).
- If the name is a constructor function or a selector function associated with a particular datatype, then the cursor will be positioned at the first line in the definition of the corresponding datatype definition.
- If the name represents an internal Hugs function, then the cursor will be positioned at the beginning of the standard prelude file.

Note that names of infix operators should be given without any enclosing them in parentheses. Thus `:f !!` starts an editor on the standard prelude at the first line in the definition of `(!!)`. If a given name could be interpreted both as a type constructor and as a value constructor, then the former is assumed.

5.3 Finding information about the system

List names	`:names [`⟨*pattern*⟩ `...]`

The `:names` command can be used to list the names of variables and functions whose definitions are currently loaded into the interpreter. Without any arguments, `:names` produces a list of all names known to the system; the names are listed in alphabetical order.

The `:names` command can also accept one or more pattern strings, limiting the list of names that will be printed to those matching one or more of the given pattern strings:

```
Prelude> :n fold*
foldl foldl' foldl1 foldr foldr1
(5 names listed)
Prelude>
```

Each pattern string consists of a string of characters and may use standard wildcard syntax: * (matches anything), ? (matches any single character), \c (matches exactly the character c) and ranges of characters of the form [a-zA-Z], etc. For example:

```
Prelude> :n *map* *[Ff]ile ?
$ % * + - . / : < > appendFile map mapM mapM_ readFile writeFile ^
(17 names listed)
Prelude>
```

Print type of expression	`:type` ⟨*expr*⟩

The `:type` command can be used to print the type of an expression without evaluating it. For example:

```
Prelude> :t "hello, world"
"hello, world" :: String
Prelude> :t putStr "hello, world"
putStr "hello, world" :: IO ()
Prelude> :t sum [1..10]
sum (enumFromTo 1 10) :: (Num a, Enum a) => a
Prelude>
```

Note that Hugs displays the most general type that can be inferred for each expression. For example, compare the type inferred for `sum [1..10]` above with the type printed by the evaluator (using `:set +t`):

```
Prelude> :set +t
Prelude> sum [1..10]
55 :: Int
Prelude>
```

The difference is explained by the fact that the evaluator uses the Haskell default mechanism to instantiate the type variable `a` in the most general type to the type `Int`, avoiding an error with unresolved overloading.

Display information about names	`:info [`⟨*name*⟩ `...]`

The `:info` command is useful for obtaining information about the files, classes, types and values that are currently loaded.

If there are no arguments, then `:info` prints a list of all the files that are currently loaded into the interpreter.

```
Prelude> :info
Hugs session for:
```

```
/Hugs/lib/Prelude.hs
Demo.hs
Prelude>
```

If there are arguments, then Hugs treats each one as a name, and displays information about any corresponding type constructor, class, or function. The following examples show the the kind of output that you can expect:

- Datatypes: The system displays the name of the datatype, the names and types of any constructors or selectors, and a summary of related instance declarations:

  ```
  Prelude> :info Either
  -- type constructor
  data Either a b

  -- constructors:
  Left :: a -> Either a b
  Right :: b -> Either a b

  -- instances:
  instance (Eq b, Eq a) => Eq (Either a b)
  instance (Ord b, Ord a) => Ord (Either a b)
  instance (Read b, Read a) => Read (Either a b)
  instance (Show b, Show a) => Show (Either a b)
  instance Eval (Either a b)

  Prelude>
  ```

  Newtypes are dealt with in exactly the same way. For a simple example of a datatype with selectors, the output produced for a `Time` datatype:

  ```
  data Time = MkTime { hours, mins, secs :: Int }
  ```

  is as follows:

  ```
  Time> :info Time
  -- type constructor
  data Time

  -- constructors:
  MkTime :: Int -> Int -> Int -> Time

  -- selectors:
  hours :: Time -> Int
  mins :: Time -> Int
  secs :: Time -> Int
  ```

```
-- instances:

instance Eval Time

Time>
```

- Type synonyms: The system displays the name and expansion:

```
Prelude> :info String
-- type constructor
type String = [Char]

Prelude>
```

  The expansion is not included in the output if the synonym is restricted.

- Type classes: The system lists the name, superclasses, members, and instance declarations for the specified class:

```
Prelude> :info Num
-- type class
class (Eq a, Show a, Eval a) => Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  fromInt :: Int -> a

-- instances:
instance Num Int
instance Num Integer
instance Num Float
instance Num Double
instance Integral a => Num (Ratio a)

Prelude>
```

- Other values: For example, named functions and individual constructor, selector, and member functions are displayed with their name and type:

```
Time> :info . : hours min
(.) :: (a -> b) -> (c -> a) -> c -> b

(:) :: a -> [a] -> [a]  -- data constructor
```

```
hours :: Time -> Int  -- selector function

min :: Ord a => a -> a -> a  -- class member

Time>
```

As the last example shows, the `:info` command can take several arguments and prints out information about each in turn. A warning message is displayed if there are no known references to an argument:

```
Prelude> :info (:)
Unknown reference `(:)'
Prelude>
```

This illustrates that the arguments are treated as textual names for operators, not syntactic expressions (for example, identifiers). The type of the `(:)` operator can be obtained using the command `:info :` as above. There is no provision for including wildcard characters of any form in the arguments of `:info` commands.

If a particular argument can be interpreted as, for example, both a constructor function, and a type constructor, depending on context, then the output for both possibilities will be displayed.

6. Library overview

Haskell 1.4 places much greater emphasis on the use of libraries than previous versions of the language. Following that lead, the Hugs 1.4 distribution includes most of the official libraries defined in the Haskell Library Report [6]. The distribution also includes a number of unofficial libraries, which fall into two categories: portable libraries, which are implemented using standard Haskell or widely implemented Haskell extensions; and Hugs-specific libraries, which use features that are not available in other Haskell implementations.

All that you need to do to use libraries is to import them using an `import` declaration. For example:

```
module MandlebrotSet where
import Array
import Complex
...
```

Of course, this assumes that `HUGSPATH` has been set to point to the directories where the libraries are stored (Section 4.1), and that import chasing is enabled. The default search path includes the directories containing both the standard and unofficial libraries.

6.1 Standard Libraries

The Hugs 1.4 distribution includes the following standard libraries: `Array`, `Char`, `Complex`, `IO`, `Ix`, `List`, `Maybe`, `Monad`, `Prelude`, `Ratio`, and `System`. The library report [6] contains full descriptions of the standard libraries. Differences between the library report and the libraries supplied with Hugs are described in Section 8.

6.2 Portable Libraries

These libraries are not part of the Haskell standard but can be ported to most Haskell systems.

- `Number` This library defines a numeric datatype of fixed width integers (whatever `Int` supplies). However, unlike the built-in `Int` type, overflows are detected and cause a run-time error. To ensure that all integer arithmetic in a given module includes overflow protection you must include a default declaration for `Number`.

```
module Number where
data Number              -- fixed width integers
instance Eq       Number -- class instances
instance Ord      Number
instance Show     Number
instance Enum     Number
instance Num      Number
instance Bounded  Number
instance Real     Number
instance Ix       Number
instance Integral Number
```

- `Random` This library provides a random number generator. We expect that this will be moved into an official Library module soon.

- `IOExtensions` This module provides non-standard extensions to the `IO` monad. Some of these are unsafe (they may break referential transparency) and must be used carefully.

```
module IOExtensions where

readBinaryFile     :: FilePath -> IO String
writeBinaryFile    :: FilePath -> String -> IO ()
appendBinaryFile   :: FilePath -> String -> IO ()
openBinaryFile     :: FilePath -> IOMode -> IO Handle

getCh              :: IO Char
fixIO              :: (a -> IO a) -> IO a
argv               :: [String]
garbageCollect     :: IO ()

unsafePerformIO    :: IO a -> a
unsafeInterleaveIO :: IO a -> IO a
```

- `ListUtils` This module includes list functions that were removed from the Prelude in the move from Haskell 1.2 to Haskell 1.3.

```
module ListUtils where

deleteFirstsBy   :: (a -> a -> Bool) -> [a] -> [a] -> [a]
sums, products   :: Num a => [a] -> [a]

subsequences     :: [a] -> [[a]]
permutations     :: [a] -> [[a]]
```

- `ParseLib` This module provides a library of parser combinators, as described in the paper on *Monadic Parser Combinators* by Graham Hutton and Erik Meijer [3].

- `Trace`: This library provides a single function, that can sometimes be useful for debugging:

  ```
  module Trace where
  trace :: String -> a -> a
  ```

  When called, `trace` prints the string in its first argument, and then returns the second argument as its result. The `trace` function is not referentially transparent, and should only be used for debugging, or for monitoring execution. You should also be warned that, unless you understand some of the details about the way that Hugs programs are executed, results obtained using `trace` can be rather confusing. For example, the messages may not appear in the order that you expect. Even ignoring the output that they produce, adding calls to `trace` can change the semantics of your program. Consider this a warning!

- `Interact`: This library provides facilities for writing simple interactive programs.

  ```
  module Interact where

  type Interact = String -> String

  end                :: Interact
  readChar, peekChar :: Interact -> (Char -> Interact) -> Interact
  pressAnyKey        :: Interact -> Interact
  unreadChar         :: Char -> Interact -> Interact
  writeChar          :: Char -> Interact -> Interact
  writeStr           :: String -> Interact -> Interact
  ringBell           :: Interact -> Interact
  readLine           :: String -> (String -> Interact) -> Interact
  ```

  An expression `e` of type `Interact` can be executed as a program by evaluating `run e`.

- `AnsiScreen` This library defines some basic ANSI escape seqences for terminal control.

  ```
  module AnsiScreen where

  type Pos  = (Int,Int)
  ```

```
at        :: Pos -> String -> String
highlight :: String -> String
goto      :: Int -> Int -> String
home      :: String
cls       :: String
```

The definitions in this module will need to be adapted to work with terminals that do not support ANSI escape sequences.

- `AnsiInteract` This library includes both `Interact` and `AnsiScreen`, and also contains further support for screen oriented interactive I/O.

```
module AnsiInteract(module AnsiInteract,
                    module Interact,
                    module AnsiScreen) where
import AnsiScreen
import Interact

clearScreen      :: Interact -> Interact
writeAt          :: Pos -> String -> Interact -> Interact
moveTo           :: Pos -> Interact -> Interact
readAt           :: Pos                    -> -- start coords
                    Int                    -> -- max input length
                    (String -> Interact) -> -- continuation
                    Interact
defReadAt        :: Pos                    -> -- start coords
                    Int                    -> -- max input length
                    String                 -> -- default value
                    (String -> Interact) -> -- continuation
                    Interact
promptReadAt     :: Pos                    -> -- start coords
                    Int                    -> -- max input length
                    String                 -> -- prompt
                    (String -> Interact) -> -- continuation
                    Interact
defPromptReadAt  :: Pos                    -> -- start coords
                    Int                    -> -- max input length
                    String                 -> -- prompt
                    String                 -> -- default value
                    (String -> Interact) -> -- continuation
                    Interact
```

- `IORef`: This library extends the Hugs `IO` monad with a datatype representing mutable reference cells, together with a small collection of related operators in the style of Peyton Jones and Wadler [9]:

```
module IORef where

data Ref a  -- Mutable reference cells, holding values of type a.
```

```
newRef      :: a -> IO (Ref a)
getRef      :: Ref a -> IO a
setRef      :: Ref a -> a -> IO ()
eqRef       :: Ref a -> Ref a -> Bool

instance Eq (Ref a)
```

- ST: This library provides support for lazy state threads and for the ST monad, as described by John Launchbury and Simon Peyton Jones [5].

```
module ST where

data MutVar s a   -- mutable variables containing values
                  -- of type a in state thread s.

newVar       :: a -> ST s (MutVar s a)
readVar      :: MutVar s a -> ST s a
writeVar     :: MutVar s a -> a -> ST s ()
interleaveST :: ST s a -> ST s a

instance Eq (MutVar s a)
instance Monad (ST s)
```

The runST operation, used to specify encapsulation, is currently implemented as a language construct, and runST is treated as a keyword.

Note that it is possible to install Hugs 1.4 without support for lazy state threads, and hence the primitives described here may not be available in all implementations. Also, in contrast with the implementation of lazy state threads in previous releases of Hugs and Gofer, there is no direct relationship between the ST and the IO monads.

- STArray: This library extends both the ST and Array libraries with support for mutable arrays in the form described by John Launchbury and Simon Peyton Jones [5].

```
module STArray where

data MutArr s a b -- Mutable arrays, indexed by type a, with
                  -- results of type b in state thread s.

newArr    :: Ix a => (a,a) -> b -> ST s (MutArr s a b)
readArr   :: Ix a => MutArr s a b -> a -> ST s b
writeArr  :: Ix a => MutArr s a b -> a -> b -> ST s ()
freezeArr :: Ix a => MutArr s a b -> ST s (Array a b)
```

- `MVar` This library provides an implementation of synchronisation objects (`MVar`s), exactly as described in the paper by Simon Peyton Jones, Andrew Gordon and Sigbjorn Finne [8].

  There is one significant difference between the implementation of these features in GHC and in Hugs:

  - GHC uses preemptive multitasking: Context switches can occur at any time, except if you call a C function (like `getchar`) that blocks waiting for input.
  - Hugs uses cooperative multitasking: Context switches only occur when you use one of the primitives defined in this module. This means that programs such as:

    ```
    main = forkIO (write 'a') >> write 'b'
     where write c = putChar c >> write c
    ```

    will print either `aaaaaaaaaaaaaa...` or `bbbbbbbbbbbb...`, instead of some random interleaving of `a`s and `b`s.

  Cooperative multitasking is sufficient for writing coroutines.

  ```
  module MVar where

  data MVar a               -- datatype of MVars

  forkIO   :: IO a -> IO () -- Spawn a thread

  newMVar  :: IO (MVar a)
  takeMVar :: MVar a -> IO a
  putMVar  :: MVar a -> a -> IO ()

  instance Eq (MVar a)
  ```

- `Channel` This library provides an implementation of buffered channels, exactly as described in the paper by Simon Peyton Jones, Andrew Gordon and Sigbjorn Finne [8].

  ```
  module Channel where
  data Channel a  -- datatype of buffered channels
  newChan         :: IO (Channel a)
  putChan         :: Channel a -> a -> IO ()
  getChan         :: Channel a -> IO a
  dupChan         :: Channel a -> IO (Channel a)
  ```

- `CVar` This library provides an implementation of channel variables, exactly as described in the paper by Simon Peyton Jones, Andrew Gordon and Sigbjorn Finne [8].

```
module CVar where

import MVar

type CVar a = (MVar a,  -- Producer -> consumer
               MVar ()) -- Consumer -> producer

newCVar    :: IO (CVar a)
putCVar    :: CVar a -> a -> IO ()
getCVar    :: CVar a -> IO a
```

6.3 Hugs-Specific Libraries

- `Trex` This library supports TREX extensible records. These can only be used when Hugs is compiled with TREX support using the `-enable-TREX` configuration option. TREX documentation is included in this release.

- `Dynamic` This library provides support for dynamic typing.

- `HugsInternals` This library provides primitives for accessing Hugs internals; for example, they provide the means with which to implement simple error-recovery and debugging facilities in Haskell. They should be regarded as an *experimental* feature and may not be supported in future versions of Hugs. They can only be used if Hugs was configured with the `--enable-internal-prims` flag.

- `GenericPrint` This library provides a "generic" (or "polymorphic") print function in Haskell, that works in essentially the same way as Hugs' builtin printer when the `-u` option is used. The module `HugsInternals` is required.

- `CVHAssert` This library provides a simple implementation of Cordy Hall's assertions for performance debugging. These primitives are an *experimental* feature that may be removed in future versions of Hugs. They can only be used if Hugs was configured with the `--enable-internal-prims` flag.

- `Win32` This library contains Haskell versions for many of the functions in the Microsoft Win32 library. It is only available on Windows 95/NT. The `--with-plugins` configuration option must be used in conjunction with this and the other Microsoft libraries.

- `Graphics` A comprehensive graphics and windowing library for Win32 is described separately. This is a higher level interface to much of the `Win32` library.

- `XLib` This library provides facilities for X window programming.

- `RandomIO` This library provides a random number generator, using the clock as a seed. Currently, it is available only on Win32, although it may be moved into an official Library module at some point in the future.

7. Other Hugs programs

The Hugs 1.4 interpreter is available in two other guises: a stand-alone system that executes programs in a 'load and go' style, without the surrounding command system; and a Windows user interface, layered on top of the basic Hugs system.

7.1 Stand-alone program execution

Once a program has been developed and debugged, the Hugs command loop can be eliminated and the program can be executed immediately without any command to run it. A slightly modified version of the interpreter called `runhugs` loads the literate program specified as its first argument and runs `main` in module `Main`. Unlike the standard Hugs system, `runhugs` makes command arguments available to the running Hugs system. The first argument, specifying the program, is removed from the argument list.

On Unix systems, executable programs may be created by placing `runhugs` in the first line of an executable file, like so:

```
#!/hugs/runhugs

> module Main where
> main = putStr "Hello, World\n"
```

Because `runHugs` uses literate Haskell only, the line starting with `#!` is viewed as a comment. Stand-alone programs can import other modules using import chasing—these modules need not be literate. The `runhugs` program uses the same environment variables to set Hugs options as the standard Hugs systems. However, `runhugs` does not set options from the command line; all command line options are passed into the executing Hugs program. The stand-alone Hugs program may return an exit code.

On Windows 95/NT, `runhugs` is invoked using a separate file extension that is set up to call `runhugs` rather than `hugs`. Installation sets up the `.hsx` extension for this purpose. A `.hsx` program will run when it is clicked on; a console window will appear if the program writes to standard output or reads from standard input. This window is closed immedately upon exiting the program. There is no way to pass parameters to the `.hsx` program when it is double-clicked. Windows 95/NT can also use `runhugs` to open files of a given type; this involves setting the "open" command for the file type to call `runhugs`, passing it the Haskell program to run and the file being opened. The online documentation has some examples of this.

7.2 Hugs for Windows

Hugs for Windows (`winhugs`) offers a GUI front-end to the Hugs interpreter on Microsoft Windows platforms. The user interface features a scrolling console window that mimics the normal Hugs interface, together with a menu and toolbar that provide additional facilities for browsing Haskell programs. Most of the additional features are self-explanatory, although short descriptions of menu and toolbar choices are displayed in a status line. Hugs for Windows uses the same command line options and environment/registry variables as Hugs. It also stores options in a `.ini` file.

Further development and enhancement of the current Hugs for Windows front-end is planned for future releases.

8. Conformance with Haskell 1.4

A number of Haskell 1.4 features are not yet implemented in Hugs 1.4. All known differences between the specification and implementation are described here.

8.1 Haskell 1.4 features not in Hugs

- Mutually recursive modules have not been implemented.
- Some library functions have been moved into the Prelude. This is necessary because the Prelude and the standard libraries, as defined in the 1.4 report, are mutually recursive. This mutual recursion has been avoided by moving the following functions into the Prelude:
  - From `Ix`: `range, index, inRange, rangeSize`.
  - From `Char`: `isAscii`, `isControl`, `isPrint`, `isSpace`, `isUpper`, `isLower`, `isAlpha`, `isDigit`, `isOctDigit`, `isHexDigit`, `isAlphanum`, `digitToInt`, `intToDigit`, `toUpper`, `toLower`, `ord`, and `chr`.
  - From `Ratio`: `Ratio`, `Rational`, `(%)`, `numerator`, `denominator`, and `approxRational`.
- Derived `Read` instances do not work for some infix constructors. If an infix constructor has left associativity and the type appears recursively on the left side of the constructor, then the read instance will loop.
- Hugs does not use the Unicode character set yet. Characters are currently drawn from the ISO Latin-1 set.
- The floating point printer is not exactly as defined in the report. The printed form of a floating point number may re-read as a slightly different number.
- Derived instances for large tuples are not supplied. Instances for tuples larger than 5 (3 in the 16 bit PC system) are not in the Prelude.
- When using getArgs, only the stand-alone system passes arguments to the executing program. The interactive system always uses an empty argument list when runnning a program.

- Fixities are global instead of localized in each module. The same name cannot be associated with more than one fixity even if the names are in different modules.

- The syntax for `n+k` patterns is slightly different. For example, this parses as an `n+k` pattern: `f ((+) x 1) = x`.

- The syntax of sections is slightly different. For example, the Haskell expression `(2*3+)` must instead be written as `((2*3)+)`.

- There are some subtle differences between the Hugs and Haskell type systems. In particular:
  - Polymorphic recursion is only supported for values whose declared types do not include any class constraints.
  - Some valid Haskell programs that make essential use of the local class constraints that are sometimes associated with individual member functions are treated as type errors in Hugs.

- Instead of `IO.hIsEOF`, Hugs provides `IO.hugsHIsEOF`. Whereas `hIsEOF` should tell you if the next call of `hGetChar` would raise an EOF error; `hugsHIsEOF` tells you if the last call of `hGetCHar` raised an EOF error (the same as ANSI C's `feof`).

- We ignore entity lists in qualified imports (but unqualified imports are treated correctly). For example, you can write:

  ```
  import qualified Prelude ( foo )
  ```

  even though `foo` is not exported from the Prelude and you can write:

  ```
  module M() where
  import qualified Prelude () -- import nothing
  x = Prelude.length "abcd"
  ```

- The `Double` type is implemented as a single precision float (this isn't forbidden by the standard but it is unusual).

8.2 Libraries

The following libraries are not yet available: `Directory`, `Time`, `Locale`, `CPUTime`, `Random`, `Bit`, `Nat`, and `Signed`.

In the IO library, these functions are not defined: `handlePosn`, `ReadWriteMode` `hFileSize`, `hIsEOF`, `isEOF`, `hSetBuffering`, `hGetBuffering`, `hSeek`, `hIsSeekable`, `hReady`, and `hLookahead`. The following non-standard functions are exported:

```
hugsGetCh :: IO Char   -- getchar without echoing to screen
hugsHIsEOF :: Handle -> IO Bool
  -- same semantics as C's "feof" (different from Haskell's hIsEOF)
hugsIsEOF  :: IO Bool
  -- same semantics as C's "feof(stdin)"
hPutStrLn  :: String -> IO ()
  -- corresponds to Prelude.putStrLn
```

8.3 *Haskell 1.4 extensions*

Hugs 1.4 contains some modest extensions to the Haskell language.

- Restricted type synonyms (as implemented in Gofer).
- Import declarations may specify a file name instead of a module name.
- TREX extensible records may be compiled into the system.
- The T(..) syntax is allowed for type synonyms in import and export lists.

9. Pointers to further information

Hugs

The full distribution for Hugs is available on the World Wide Web from:

`http://haskell.org/hugs`

or by anonymous ftp from

`ftp://ftp.haskell.org/ftp/hugs`.

The distribution includes source code, demo programs, library files, user documentation, and precompiled binaries for common platforms. These pages are mirrored locally in Nottingham at

`http://www.cs.nott.ac.uk/Department/Staff/mpj/hugs.html`.

A mailing lists for Hugs users is at (`hugs-users@haskell.org`), and another for bug reports is at (`hugs-bugs@haskell.org`). To subscribe, send an email message to `hugs-users-request@haskell.org`, or to `hugs-bugs-request@haskell.org`, respectively. An overview of nearly all Haskell related resources can be found at

`http://haskell.org`.

Functional programming

The usenet newsgroup `comp.lang.functional` provides a forum for general discussion about functional programming languages. A list of frequently asked questions (FAQs), and their answers, is available from:

`http://www.cs.nott.ac.uk/Department/Staff/gmh/faq.html`.

The FAQ list contains many pointers to other functional programming resources around the world.

Further reading

As we said at the very beginning, this manual is not intended as a tutorial on either functional programming in general, or Haskell in particular. For these things, our first recommendations would be for the *Introduction to Functional Programming* by Bird and Wadler [1], and the *Gentle Introduction to Haskell* by Hudak, Peterson and Fasel [2], respectively. Note, however, that there are several other good textbooks dealing either with Haskell or related languages.

For those with an interest in the implementation of Hugs, the report about the implementation of Gofer [4], Hugs' predecessor, should be a useful starting point.

Bibliography

[1] R. Bird and P. Wadler. *Introduction to functional programming.* Prentice Hall, 1988.

[2] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992. Also available as Research Report YALEU/DCS/RR-901, Yale University, Department of Computer Science, April 1992.

[3] G. Hutton and E. Meijer. Monadic parser combinators. Available from `http://www.cs.nott.ac.uk/Department/Staff/gmh/bib.html`, 1996.

[4] M. Jones. The implementation of the Gofer functional programming system. Research Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA, May 1994. Available on the World-Wide Web from `http://www.cs.nott.ac.uk/Department/Staff/mpj/pubs.html`.

[5] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[6] J. Peterson and K. Hammond (editors). The Haskell library report version 1.4. Research Report YALEU/DCS/RR-1105, Yale University, Department of Computer Science, April 1997.

[7] J. Peterson and K. Hammond (editors). Report on the Programming Language Haskell 1.4, A Non-strict Purely Functional Language. Research Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, April 1997.

[8] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent haskell. In *Conference record of POPL '96: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, FL, January 1996. ACM press.

[9] S. Peyton Jones and P. Wadler. Imperative functional programming. In *Proceedings 20th Symposium on Principles of Programming Languages.* ACM, January 1993.