Hugs 1.3

The Haskell User's Gofer System

User Manual

Mark P. Jones
Technical Report NOTTCS-TR-96-2
Department of Computer Science
The University of Nottingham
Nottingham NG7 2RD, England
August 1996

Contents

Conditions of use, duplication and distribution

1. *Introduction*

Hugs 1.3 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 1.3, 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: Pointers to further information.

Whether you are a beginner or a seasoned old-timer, I 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 my friends and colleagues in the functional programming groups at Nottingham and at Yale for their input to the current release, and to my family for the time that they have given to allow me to work on it.

2. A technical summary of Hugs 1.3

Hugs 1.3 provides an almost complete implementation of Haskell 1.3 [6], 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 1.3 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 1.3 primitives for monadic I/O, with support for simple interactive programs, access to text files, handle-based I/O, and exception handling.

The only Haskell 1.3 feature that is not supported is the module system.

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 script files, 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 features, including the labelled field syntax, do-notation, newtype, strictness annotations in datatypes, the `Eval` class, ISO character set, etc.
- User interface enhancements, particularly the `HUGSPATH` and import chasing features, both of which were motivated by a greater emphasis on the role of libraries in Haskell 1.3.
- Small improvements in runtime performance, and more reliable space usage, thanks to the use of non-conservative garbage collection during program execution.

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

Already, there are ambitious plans for the future, with collaboration and joint releases from the functional programming groups at Nottingham and Yale. For example, in the near future, we expect to release a system combining the best of Hugs 1.3 and the recent Yale release of Hugs 1.01. This will provide full support for modules, Haskell 1.3, and libraries for both X window and Win32 programming. However, we also intend to maintain the current release as a relatively small and stable system that will be suitable for teaching and research, even on fairly small machines.

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 already be familiar to readers 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:

```
     ___    ___   ___    ___   __________   __________
    /  /   /  /  /  /   /  /  /  _______/  /  _______/  The Haskell User's
   /  /___/  /  /  /   /  /  /  / _____   /  /______       Gofer System
  /  ____   /  /  /   /  /  /  / /_   /  /______   /
 /  /   /  /  /  /___/  /  /  /___/  /  _______/  /         Version 1.3
/__/   /__/  /_________/  /_________/  /_________/          August 1996

  Copyright (c) Mark P Jones, The University of Nottingham, 1994-1996.

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[1]. 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.

```
? (2+3)*8
40
? sum [1..10]
55
?
```

[1]If Hugs does not load correctly, and complains that it cannot find the prelude, then you will need to set the `HUGSPATH` environment variable, as described in Section 4.1.

The ? character on the first, third and fifth lines here is the Hugs prompt, indicating that the system is ready to accept input from the user. 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 built-in 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:

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

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:

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

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:

```
? putStr "hello, " >> putStr "world"
hello, world
?
```

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.

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 reading script 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 Scripts

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 *script* files that can be loaded and used by Hugs. For example, suppose that you put the following definition:

```
fact n = product [1..n]
```

into a file called `fact.hs`. (By convention, Hugs scripts are stored in files ending with the characters `.hs`.) 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:

```
? :load fact.hs
Reading script file "fact.hs":

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

Notice the list of filenames displayed after `Hugs session for:`; this tells you which files of definitions are currently being used by Hugs, the first of which is always the standard prelude. Now that the `fact.hs` file has been loaded, we can start to use the `fact` function that we have defined:

```
? fact 6
720
? fact 6 + fact 7
5760
? fact 7 ‘div‘ fact 6
7
?
```

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:

```
? comb 5 2 where comb n r = fact n ‘div‘ (fact r * fact (n-r))
10
?
```

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, we can use the `comb` function like any other built-in operator:

```
? comb 5 2
10
?
```

4. Starting Hugs

The Hugs interpreter is usually started with a command line of the form:

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

(This manual assumes that Hugs has already been successfully installed on your system, and that it can be invoked using the `hugs` command.) As soon as it starts, the interpreter will perform the following tasks:

- Process any command line options. These are distinguished from other command line arguments by a leading `+` or `-` and are used to customize the behaviour of the interpreter (Section 4.2).
- Initialize the interpreter's internal data structures. 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 the `-h` command line option when the interpreter is started.
- Load the prelude file. The interpreter will look for the prelude file on the path specified by the `HUGSPATH` environment variable (Section 4.1), or by the path supplied with a `-P` command line option. If the prelude cannot be found in one of the path directories, or if the path has not been set to an appropriate value, then the interpreter will look for `Prelude.hs` in the current directory. If the prelude cannot be found, or if an error occurs while it is being loaded, then the interpreter will terminate; Hugs will not run without the prelude file.
- Load any script files specified on the command line. 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 variables

There are two environment variables that should normally be set before the interpreter is used:

- `HUGSPATH` specifies the list of directories that will be searched for Hugs script files, including the standard prelude when the interpreter is first invoked. Directory names should be separated by colons or, on DOS/Windows machines, by semicolons. Empty components in the path will be treated as references to the current working directory.
- `HUGSEDIT` specifies the command line for an external editor. Any occurrences of `%d` and `%s` in the `HUGSEDIT` command line will be 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.

Values for these variables can be provided when the interpreter is invoked using the `-P` and `-E` command line options, respectively; these settings can be further inspected and modified while the interpreter is running using the `:set` command.

However, it is usually more convenient to save preferred settings in environment variables that will be used automatically each time the interpreter is started. The method for setting these variables 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 `HUGSPATH` and `HUGSEDIT` 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 HUGSPATH /usr/local/Hugs/lib:/usr/local/Hugs/libhugs
  set HUGSEDIT "vi +%d %s"
  ```

  The editor specified here is `vi` which allows the user to specify a start up line number by preceding it with a `+` character. The settings are easily changed to accommodate other editors. For example, you can use the following line to configure Hugs to use `emacs`:

  ```
  set HUGSEDIT "emacs +%d %s"
  ```

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
HUGSPATH=/usr/local/Hugs/lib:/usr/local/Hugs/libhugs
export HUGSPATH
HUGSEDIT="vi +%d %s"
export HUGSEDIT
exec /usr/local/bin/hugs +s $*
```

One advantage of this approach is that individual users do not have to worry about setting the environment variables themselves. In addition, the script can be used to specify startup command line options—like the `+s` in this example—without needing to change the default settings that are built in to the interpreter. 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 should add the following lines to `autoexec.bat`:

  ```
  set HUGSPATH=\hugs\lib;\hugs\libhugs
  set HUGSEDIT=vi +%%d %%s
  ```

  The setting for `HUGSPATH` 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 `HUGSEDIT` will only work if you have installed the editor program, `vi`, that it refers to. If you don't want to install a new editor, then you can `set HUGSEDIT=edit` for the standard DOS editor, or `set HUGSEDIT=notepad` for the Windows 95 notepad editor. Note, however, that neither `edit` or `notepad` allow the intepreter to specify a start line number.

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 the `HUGSEDIT` variable has not 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.

Previous versions of Hugs used a `HUGS` variable to locate the standard prelude file, but this variable is not used by Hugs 1.3.

4.2 Command line options

The behaviour of the interpreter, particularly the read-eval-print loop, can be customized using command line 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, most of the command line 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`.

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 command line 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 command line options is described in the sections below.

Print statistics	`+s,-s`

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

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

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:

```
? :set +s
? map (\x -> x*x) [1..10]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
(230 reductions, 408 cells)
? [1..]
[1, 2, 3, 4, {Interrupted!}
(18 reductions, 54 cells)
?
```

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:

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

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:

```
? [1 ‘div‘ 0]
[
Program error: {primDivInt 1 0}
```

```
? [1 ‘div‘ 0, 2]
[
Program error: {primDivInt 1 0}

?
```

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:

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

Notice that, reading an expression in braces as bottom, ⊥, both of the values printed by Hugs give the correct value. Of course, it is not possible to detect all occurrences of bottom like this, such as those produced by a nonterminating computation:

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

?
```

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 have been 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:

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

?
```

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

The effect of using literate scripts 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 scripts 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 scripts:

- Every input file must contain at least one line whose first character is `>`. This prevents scripts 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).

Display dots while loading	`+.,-.`

As Hugs loads each script file into the interpreter, it prints a short sequence of messages to indicate progress through the various stages of parsing the script, 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:

```
? :l Array
Reading script file "/Hugs/lib/Array.hs":

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

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:

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

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

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 script files; in one experiment, a particular program took nearly five times longer to load when the `+.` option was used.

List files loaded	`+w,-w`

By default, Hugs prints a complete list of all the script 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 script files.

Detailed kind errors	`+k,-k`

Hugs 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 lines:

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

will cause an error:

```
ERROR "Demo" (line 12): 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 "Demo.hs" (line 12): 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:

```
? :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)

?
```

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 most 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`.

Import chasing	`+i,-i`

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

For example, consider a script file `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 script 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 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`:

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

Initially, the interpreter reads only the first part of any script file loaded into the system, upto and including any import statements. If there are no imports, or if the files specified as imports have already been loaded, then the system carries on and loads the script file as normal. On the other hand, if the script includes import statements for files that have not already been loaded, then the interpreter postpones the task of reading the current script until all of the specified imports have been successfully loaded. This explains why the `Demo.hs` script and the `STArray` library 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 `HUGSPATH`, 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 script 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 script 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.

- Hugs allows multiple module definitions within a single script file, but the import chasing mechanisms do not work properly with such files because they only take account of import statements in the first module.

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*⟩ command line 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. Note that the heap is used to hold an intermediate (parsed) form of script files while they are being read, type checked and compiled. It follows that, the larger the script file, the larger the heap required to enable that file to be loaded into Hugs. In practice, most large programs are written (and loaded) as a number of separate scripts which means that this does not usually cause problems.

Unlike all of the other command line options described here, the heap size setting cannot be changed from within the interpreter using a `:set` command.

Set prompt	`-p`⟨*string*⟩

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

```
? :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.

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:

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

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:

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

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 Hugs script.

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.

Set search path	`-P`⟨*path*⟩

A `-P`⟨*path*⟩ option can be used to change the Hugs search path to the specified ⟨*path*⟩. The search path is usually initialized to the value of the `HUGSPATH` environment variable when the interpreter starts running. The format of the ⟨*path*⟩ string is described in more detail in Section 4.1.

Set editor command line	`-E`⟨*cmd*⟩

A `-E`⟨*cmd*⟩ option can be used to change the editor command line string to the specified ⟨*cmd*⟩ while the interpreter is running. The editor command line string is usually initialized to the value of the `HUGSEDIT` environment variable when the interpreter starts running. The format of the ⟨*cmd*⟩ string is described in more detail in Section 4.1.

5. *Hugs commands*

Hugs provides a number of commands that can be used to evaluate expressions, to load script 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 `My File` file whose name includes an embedded space:

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

However, 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	⟨*expr*⟩

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 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.

  ```
  ? putStr "hello, world"
  hello, world
  ?
  ```

- 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.

  ```
  ? "hello" ++ ", " ++ "world"
  "hello, world"
  ?
  ```

  Unlike previous versions of Hugs and Gofer, there is no special treatment for values of tye `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:

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

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, `getChar` has type `IO Char` which does not match the type `IO ()` for a program and is not an instance of `Show`:

```
? getChar
ERROR: Cannot find "show" function for:
*** expression : getChar
*** of type    : IO Char
?
```

In this particular case, the problem occurs because the user has not specified what they want to do with the character that is returned by `getChar`. If you want to run the program and discard its result, then you must do this explicitly:

```
? getChar >> return ()
w
?
```

If the errors occurs when you try to display the value of a user-defined datatype, then you will need to add an instance of the `Show` class to your program. One of the simplest way to achieve this is to request a derived instance of `Show` as part of the datatype definition, as in:

```
data Date = Date { day::Int, month::Month, year::Int }
            deriving Show

data Month = Jan | Feb | Mar | Apr | May | Jun
           | Jul | Aug | Sep | Oct | Nov | Dec
             deriving Show

xmas = Date { day = 25, month = Dec, year = 1996 }
```

For example, once these definitions have been loaded into Hugs, we can evaluate and display the date `xmas` as we would expect:

```
? xmas
Date{day=25,month=Dec,year=1996}
?
```

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 command line settings.

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

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

```
? :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 scripts as default
e    Warn about errors in literate scripts
```

```
.    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 script files 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"
?
```

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

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

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

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.

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:

```
? :gc
Garbage collection recovered 95766 cells
?
```

List commands `:?`

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

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

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

Exit the interpreter `:quit`

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

5.2 Loading and editing scripts and projects

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

The `:load` command removes any previously loaded script files, 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. The load process will also be restarted automatically after a `:edit` command.

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 `HUGSPATH` environment variable (Section 4.1) to search for script 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 script file that was entered by the user, while `path` is the list of directories in the `HUGSPATH`. 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 script files, without removing any previously loaded files. (However, if any of the previously loaded script files 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 script files.

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 script files has been modified, then it will be reloaded. Note that script files are loaded in a specific order, with the possibility that later scripts may depend on the definitions in earlier ones. To allow for this, if one script has been reloaded, then all subsequent scripts 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 script files 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 script 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 1.3 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 script 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 script 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.proj` is equivalent to starting Hugs without the any arguments and then giving the command `:p Demo.prj`.

The `:project` command uses the same mechanisms as `:load` to locate the script files mentioned in a project file, but it will not use the `HUGSPATH` setting 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 script files, leaving the system free to determine the order in which the files should be loaded. For example, if the `Main` script 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 suspends the current Hugs session and starts an editor program to modify or view a script file. The Hugs session will be resumed when the editor terminates; any script files that have been changed will be reloaded automatically. The `HUGSEDIT` variable (Section 4.1), or the `-E` command line 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 script file that it tried to load. This allows the `:edit` command to integrate smoothly with the facilities for loading script 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 value for the `HUGSEDIT` variable (Section 4.1), to allow the cursor to be positioned at the correct line in the source file. 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 data 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:

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

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:

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

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

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

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

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`):

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

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 script files that are currently loaded into the interpreter.

```
? :info
Hugs session for:
/Hugs/lib/Prelude.hs
Demo.hs
?
```

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:

  ```
  ? :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)

  ?
  ```

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:

```
? :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

?
```

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

  ```
  ? :info String
  -- type constructor
  type String = [Char]

  ?
  ```

  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:

  ```
  ? :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)

?
```

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

  ```
  ? :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

  ?
  ```

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:

```
? :info (:)
Unknown reference `(:)'
?
```

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 1.3 places much greater emphasis on the use of libraries than previous versions of the language. Following that lead, the Hugs 1.3 distribution includes a number of different libraries. Some of these are based on the current proposal for standardized libraries in Haskell, while others are Hugs-specific. All of these are described in the following sections. All that you need to do to use libraries is to import them at the beginning of a Hugs script. 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.

6.1 Standard libraries

The following libraries conform closely to current proposals for the Haskell 1.3 standard, and should be supported by all Haskell platforms. However, at the time of writing, the exact details and specification of the Haskell 1.3 libraries has not been finalized, so you can expect to see some changes and possible incompatibilities in future releases.

- `Array`: This library introduces a datatype and operations for using Haskell arrays. Unlike previous versions, support for arrays is not included as part of the standard prelude. The operations provided by this library are as follows:

```
module Array where

infixl 9  !, //

data Array a b -- Arrays indexed by values of type a,
               -- with results of type b.

array      :: Ix a => (a,a) -> [(a,b)] -> Array a b
listArray  :: Ix a => (a,a) -> [b] -> Array a b
```

```
(!)        :: Ix a => Array a b -> a -> b
bounds     :: Ix a => Array a b -> (a,a)
indices    :: Ix a => Array a b -> [a]
elems      :: Ix a => Array a b -> [b]
assocs     :: Ix a => Array a b -> [(a,b)]
accumArray :: Ix a => (b -> c -> b) -> b -> (a,a) -> [(a,c)]
               -> Array a b
(//)       :: Ix a => Array a b -> [(a,b)] -> Array a b
accum      :: Ix a => (b -> c -> b) -> Array a b -> [(a,c)]
               -> Array a b
amap       :: Ix a => (b -> c) -> Array a b -> Array a c
ixmap      :: (Ix a, Ix b) => (a,a) -> (a -> b) -> Array b c
               -> Array a c

instance (Ix a, Eq b) => Eq (Array a b)
instance (Ix a, Ord b) => Ord (Array a b)
instance (Ix a, Show a, Show b) => Show (Array a b)
```

- `Char`: According to the Haskell standard, this library provides support for functions on characters, including `isAlpha`, `isUpper`, `isDigit`, and `toUpper`. However, in the current distribution of Hugs, these functions are already defined in the standard prelude, so the Hugs `Char` library is just an empty stub, provided for compatibility.

- `Complex`: This library introduces a datatype and operations for complex numbers. Unlike previous versions, support for complex numbers is not included as part of the standard prelude. The operations provided by this library are as follows:

```
module Complex where

infix  6  :+

data RealFloat a => Complex a = !a :+ !a
     deriving (Eq, Read, Show)

realPart, imagPart :: RealFloat a => Complex a -> a
conjugate          :: RealFloat a => Complex a -> Complex a
mkPolar            :: RealFloat a => a -> a -> Complex a
cis                :: RealFloat a => a -> Complex a
polar              :: RealFloat a => Complex a -> (a, a)
magnitude, phase   :: RealFloat a => Complex a -> a

instance RealFloat a => Num (Complex a)
instance RealFloat a => Fractional (Complex a)
instance RealFloat a => Floating (Complex a)
```

- `IO`: This library provides a number of advanced features for the `IO` monad, complementing the basic facilities in the standard prelude. Note that the names and semantics of functions defined in this library are particularly likely to change as the details of the I/O standard are clarified.

```
module IO where

isUserError    :: IOError -> Maybe String
isIllegalError, isAlreadyExists,
 isAlreadyInUse, isFullError,
 isPermissionError,
 isEOFError    :: IOError -> Bool
ioeGetHandle   :: IOError -> Maybe Handle
ioeGetFileName :: IOError -> Maybe FilePath

data Handle    -- built-in datatype of IO handles

data IOMode     = ReadMode | WriteMode | AppendMode
                  deriving (Eq, Ord, Enum, Show, Read)

stdin          :: Handle
stdout         :: Handle
stderr         :: Handle
hGetContents   :: Handle -> IO String
openFile       :: FilePath -> IOMode -> IO Handle
hClose, hFlush :: Handle -> IO ()
hIsEOF         :: Handle -> IO Bool
hPutChar       :: Handle -> Char -> IO ()
hPutStr        :: Handle -> String -> IO ()
hGetChar       :: Handle -> IO Char
getCh          :: IO Char
isEOF          :: IO Bool
```

- `Ix`: According to the Haskell standard, this library provides support for the `Ix` type class. However, in the current distribution of Hugs, this class is already defined in the standard prelude, so the Hugs `Ix` library is just an empty stub, provided for compatibility.

- `List`: This library provides a large collection of list processing utilities. Some of these functions were previously included as part of the standard prelude, but there are also many new functions:

```
module List where

infix 5 \\

delete             :: (Eq a) => a -> [a] -> [a]
deleteBy           :: (a -> a -> Bool) -> a -> [a] -> [a]
(\\)               :: (Eq a) => [a] -> [a] -> [a]
deleteFirsts       :: (Eq a) => [a] -> [a] -> [a]
deleteFirstsBy     :: (a -> a -> Bool) -> [a] -> [a] -> [a]
```

```
elemBy, notElemBy  :: (a -> a -> Bool) -> a -> [a] -> Bool
lookupBy           :: (a -> a -> Bool) -> a -> [(a, b)] -> Maybe b

partition          :: (a -> Bool) -> [a] -> ([a],[a])

nub                :: Eq a => [a] -> [a]
nubBy              :: (a -> a -> Bool) -> [a] -> [a]
group              :: (Eq a) => [a] -> [[a]]
groupBy            :: (a -> a -> Bool) -> [a] -> [[a]]

sums, products     :: Num a => [a] -> [a]

transpose          :: [[a]] -> [[a]]

genericLength      :: (Num i) => [b] -> i
genericDrop        :: (Integral i) => i -> [a] -> [a]
genericTake        :: (Integral i) => i -> [a] -> [a]
genericSplitAt     :: (Integral i) => i -> [b] -> ([b],[b])
genericReplicate   :: (Integral i) => i -> a -> [a]

elemIndex          :: Eq a => [a] -> a -> Int
elemIndexBy        :: (a -> a -> Bool) -> [a] -> a -> Int

mapAccumL          :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])
mapAccumR          :: (a -> b -> (a, c)) -> a -> [b] -> (a, [c])

intersperse        :: a -> [a] -> [a]
inits              :: [a] -> [[a]]
tails              :: [a] -> [[a]]
subsequences       :: [a] -> [[a]]
permutations       :: [a] -> [[a]]
union              :: (Eq a) => [a] -> [a] -> [a]
intersect          :: (Eq a) => [a] -> [a] -> [a]

-- Extending the zip family of functions:

zip4               :: [a] -> [b] -> [c] -> [d]
                       -> [(a,b,c,d)]
zip5               :: [a] -> [b] -> [c] -> [d] -> [e]
                       -> [(a,b,c,d,e)]
zip6               :: [a] -> [b] -> [c] -> [d] -> [e] -> [f]
                       -> [(a,b,c,d,e,f)]
zip7               :: [a] -> [b] -> [c] -> [d] -> [e] -> [f] -> [g]
                       -> [(a,b,c,d,e,f,g)]

-- Extending the zipWith family of functions:

zipWith4           :: (a->b->c->d->e)
                       -> [a]->[b]->[c]->[d]->[e]
```

```
zipWith5            :: (a->b->c->d->e->f)
                        -> [a]->[b]->[c]->[d]->[e]->[f]
zipWith6            :: (a->b->c->d->e->f->g)
                        -> [a]->[b]->[c]->[d]->[e]->[f]->[g]
zipWith7            :: (a->b->c->d->e->f->g->h)
                        -> [a]->[b]->[c]->[d]->[e]->[f]->[g]->[h]

-- Extending the unzip family of functions:

unzip4              :: [(a,b,c,d)]
                        -> ([a],[b],[c],[d])
unzip5              :: [(a,b,c,d,e)]
                        -> ([a],[b],[c],[d],[e])
unzip6              :: [(a,b,c,d,e,f)]
                        -> ([a],[b],[c],[d],[e],[f])
unzip7              :: [(a,b,c,d,e,f,g)]
                        -> ([a],[b],[c],[d],[e],[f],[g])
```

- `Maybe`: This library provides utility functions for working with the `Maybe` datatype, as defined in the standard prelude:

```
module Maybe where

exists             :: Maybe a -> Bool
the                :: Maybe a -> a
theExists          :: Maybe a -> (a, Bool)
fromMaybe          :: a -> Maybe a -> a
maybeToList        :: Maybe a -> [a]
listToMaybe        :: [a] -> Maybe a
findMaybe          :: (a -> Bool) -> [a] -> Maybe a
catMaybes          :: [Maybe a] -> [a]
mapMaybe           :: (a -> Maybe b) -> [a] -> [b]
```

- `Monad`: This library provides general purpose operations for monadic programming using instances of the `Monad`, `MonadZero`, and `MonadPlus` classes:

```
module Monad where

join         :: Monad m => m (m a) -> m a
apply        :: Monad m => (a -> m b) -> (m a -> m b)
(@@)         :: Monad m => (a -> m b) -> (c -> m a) -> (c -> m b)
mapAndUnzipL :: Monad m => (a -> m (b,c)) -> [a] -> m ([b], [c])
mapAndUnzipR :: Monad m => (a -> m (b,c)) -> [a] -> m ([b], [c])
accumulateL  :: Monad m => [m a] -> m [a]
accumulateR  :: Monad m => [m a] -> m [a]
sequenceL    :: Monad m => [m a] -> m ()
sequenceR    :: Monad m => [m a] -> m ()
zipWithL     :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithR     :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
```

```
mapL         :: Monad m => (a -> m b) -> ([a] -> m [b])
mapR         :: Monad m => (a -> m b) -> ([a] -> m [b])
foldL        :: Monad m => (a -> b -> m a) -> a -> [b] -> m a
foldR        :: Monad m => (a -> b -> m b) -> b -> [a] -> m b
concatM      :: MonadPlus m => [m a] -> m a
unless       :: Monad m => Bool -> m () -> m ()
when         :: Monad m => Bool -> m () -> m ()
```

- `Rational`: According to the Haskell standard, this library provides support for rational numbers, including the `Ratio` and `Rational` type constructors. However, in the current distribution of Hugs, these functions are already defined in the standard prelude, so the Hugs `Rational` library is just an empty stub, provided for compatibility.

- `System`: This library extends the Haskell IO monad with facilities for writing stand-alone programs that need to deal with things like command line arguments, environment variables, and exit codes. In fact, the implementation of these features in Hugs 1.3 is very weak—the operations themselves are not well-suited to an interpreter-based environment like Hugs—and the library is included mainly for compatibility with Haskell.

```
module System where

data ExitCode = ExitSuccess | ExitFailure Int
                deriving (Eq, Ord, Read, Show)

getArgs      :: IO [String]
getProgName  :: IO String
getEnv       :: String -> IO String
system       :: String -> IO ExitCode
exitWith     :: ExitCode -> IO a
```

6.2 Hugs-specific libraries

The distribution for Hugs 1.3 also includes a number of Hugs-specific libraries, most of which are described below. Please note that these libraries are not part of the Haskell standard.

- `IORef`: This library extends the Hugs I/O monad with a datatype representing mutable reference cells, together with a small collection of related operators in the style of Peyton Jones and Wadler [7]:

```
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]. The operations provided by the library are as follows:

  ```
  module ST where

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

  returnST     :: a -> ST s a
  thenST       :: ST s a -> (a -> ST s b) -> ST s b
  newVar       :: a -> ST s (MutVar s a)
  readVar      :: MutVar s a -> ST s a
  writeVar     :: MutVar s a -> a -> ST s ()
  mutvarEq     :: MutVar s a -> MutVar s a -> Bool
  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 1.3 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]. The operations provided by the library are as follows:

  ```
  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)
```

- `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!

- `Number`: This library defines a numeric datatype, `Number`, of fixed width integers (whatever `Int` supplies). Unlike the built-in `Int` type, arithmetic overflows involving `Number` values will cause a run-time error:

  ```
  module Number where

  default (Number,Int,Float)

  type Number = <restricted>

  instance Eq Number
  instance Ord Number
  instance Show Number
  instance Enum Number
  instance Num Number
  instance Bounded Number
  instance Real Number
  instance Ix Number
  instance Integral Number
  ```

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

7. Pointers to further information

Hugs

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

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

or by anonymous ftp from:

`ftp://ftp.cs.nott.ac.uk/pub/haskell/hugs`.

The distribution includes source code, demo programs, library files, user documentation, and precompiled binaries for common platforms.

There is also a home page for Hugs at Yale:

`http://haskell.cs.yale.edu/hugs/`,

and there are mailing lists for Hugs users (`hugs-users@cs.yale.edu`), and for bug reports (`hugs-bugs@cs.yale.edu`). To subscribe, send an email message to `hugs-users-request@cs.yale.edu`, or to `hugs-bugs-request@cs.yale.edu`, respectively.

Haskell

An online version of the Haskell 1.3 report is available from:

`http://haskell.cs.yale.edu/haskell-report/haskell-report.html`.

Archives for Haskell are maintained at various sites around the world:

Chalmers, Sweden	`ftp://ftp.cs.chalmers.se/pub/haskell`
Glasgow, Scotland	`ftp://ftp.dcs.glasgow.ac.uk/pub/haskell`
Nottingham, England	`ftp://ftp.cs.nott.ac.uk/pub/haskell`
Yale, USA	`ftp://haskell.cs.yale.edu/pub/haskell`

There is a common mailing list for technical discussion of Haskell; details are available at the sites listed above.

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/mpj/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, my first recommendations would be for the *Introduction to Functional Programming* by Bird and Wadler [1], and the *Gentle Introduction to Haskell* by Hudak 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). Report on the Programming Language Haskell 1.3, A Non-strict Purely Functional Language. Research Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, May 1996.

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

Index