Appendix C
Haskell operators

The operators in the Haskell prelude are listed below in decreasing order of binding power: see Section 3.7 for a discussion of associativity and binding power.

	Left associative	Non-associative	Right associative
9	`!!`		`.`
8			`**`, `^`, `^^`
7	`*`, `/`, `` `div` ``, `` `mod` ``, `` `rem` ``, `` `quot` ``		
6	`+`, `-`		
5			`:`, `++`
4		`/=`, `<`, `<=`, `==`, `>`, `>=`, `` `elem` ``, `` `notElem` ``	
3			`&&`
2			`||`
1	`>>`, `>>=`		
0			`$`, `$!`, `` `seq` ``

Also defined in this text are the operators

9	`>.>`		
5			`>*>`

The restrictions on names of operators, which are formed using the characters

```
! # $ % & * + . / < = > ?  \ ^ | : - ~
```

are that operators must not start with a colon; this character starts an infix *constructor*. The operators `-` and `!` can be user-defined, but note that they have a special meaning in certain circumstances – the obvious advice here is not to use them. Finally, certain combinations of symbols are reserved, and cannot be used: `..` `:` `::` `=>` `=` `@` `\` `|` `^` `<-` `->`.

To change the associativity or binding power of an operator, `&&&` say, we make a declaration like

```
infixl 7 &&&
```

which states that `&&&` has binding power 7, and is a left associative operator. We can also declare operators as non-associative (`infix`) and right associative (`infixr`). Omitting the binding power gives a default of 9. These declarations can also be used for back-quoted function names, as in

```
infix 0 `poodle`
```

Appendix D
Haskell practicalities

It's not difficult to get going using Haskell, and most of the relevant information is easily accessible from the `haskell.org` page. This appendix points you in the right direction.

Implementations

Implementations of Haskell have been built at various sites around the world. This text uses GHCi, an interactive front-end to the Glasgow Haskell Compiler (GHC). GHCi provides much of the functionality of the Hugs interpreter, which was developed in a joint effort by staff at the Universities of Nottingham in the UK and Yale in the USA. The first compilers for Haskell were developed at the University of Glasgow, UK, and Chalmers Technical University, Göteborg, Sweden. More recent developments have taken place elsewhere, including at York and Utrecht. An up-to-date list of implementations and their status can be found at

`http://www.haskell.org/haskellwiki/Implementations`

In this book we have used the **Haskell Platform** as our foundation. This is documented at

`http://hackage.haskell.org/platform/`

from where it can also be downloaded. Installation instructions for Windows, Mac OS X and Linux are listed on the relevant downloads page.

Getting the Craft3e code

The modules for this text are available as a package on Hackage: full details on how to download are available at the homepage for the book:

`www.haskellcraft.com`

Using GHCi

An overview of the main commands of GHCi can be found in Figure 2.4, page 32, and full details of other aspects of GHCi are in the online documentation for GHC.

Editors for Haskell

While there is no preferred editor for Haskell, emacs is probably the best loved and most used. To tune emacs to work with Haskell, it's good to use the Haskell mode, which is documented extensively at

```
http://www.haskell.org/haskellwiki/Haskell_mode_for_Emacs
```

Not everyone gets on with emacs, and vim is an alternative for many, with its Haskell mode available from

```
http://projects.haskell.org/haskellmode-vim/
```

Other editors include Yi, a text editor written in Haskell and extensible in Haskell.

```
http://www.haskell.org/haskellwiki/Yi
```

and an overview of all those available is at

```
http://www.haskell.org/haskellwiki/Editors
```

Appendix E
GHCi errors

This appendix examines some of the more common programming errors in Haskell, and shows the error messages to which they give rise in GHCi.

The programs we write all too often contain errors. On encountering an error, the system either halts, and gives an **error message**, or continues, but gives a **warning message** to tell us that something unusual has happened, which might signal that we have made an error. In this appendix, we look at a selection of the messages output by GHCi; we have chosen the messages which are both common and require some explanation; messages like

```
*** Exception: Prelude.head: empty list
```

are self-explanatory. The messages are classified into roughly distinct areas. Syntax errors show up malformed programs, while type errors show well-formed programs in which objects are used at the wrong types. In fact, an ill-formed expression can often show itself as a type error and not as a syntax error, so the boundaries are not clear.

Syntax errors

A Haskell system attempts to match the input we give to the syntax of the language. Commonly, when something goes wrong, we type something *unexpected*.

- Typing '2==3)' will provoke the error message

  ```
  <interactive>:1:4: parse error on input `)'
  ```

- If a part of a definition is missing, as in

  ```
  fun x
  fun 2 = 34
  ```

  we receive the message

  ```
  Errors.hs:3:0: Not in scope: `fun'
  Errors.hs:3:4: Not in scope: `x'
  ```

  The problem here is that the system tries to understand `fun` and `x` and these are not (yet) defined.

- A similar error results when the two lines are reversed, as in

```
fun 2 = 34
fun x
```

when the error message is

```
Errors.hs:4:4: Not in scope: 'x'
```

- The inclusion of a type definition in a `where` clause, like this

```
fun x = x+1
        where
          type MyInt = Int
```

is signalled by

```
Errors.hs:8:10: parse error on input 'type'
```

- The syntax of patterns is more restricted than the full expression syntax, and so we get error messages like

```
Errors.hs:6:5:
    Conflicting definitions for 'x'
    Bound at: Errors.hs:6:5
              Errors.hs:6:7
    In the definition of 'fun'
```

when we use the same variable more than once within a pattern, as in the definition

```
fun (x,x) = x+1
```

- In specifying constants, we can make errors: floating-point numbers can be too large, and characters specified by an out-of-range ASCII code: for example, on typing `'\9999999999999999'` as input we get

```
<interactive>:1:18:
    lexical error in string/character literal at character '\''
```

- Not every string can be used as a name; some words in Haskell are **keywords** or **reserved identifiers**, and will give an error if used as an identifier. The keywords are

```
case class data default deriving do else if import in infix
infixl infixr instance let module newtype of then type where
```

For example, including the definition

```
do (x,y) = x+1
```

gives this error message

```
Errors.hs:6:0: Parse error in pattern
```

and the definition

```
data (x,y) = x+1
```

gives this message

```
Errors.hs:6:13: Not a data constructor: 'x'
```

As you can see from the two examples, the error message in a case like this reflects the meaning of the keyword.

- The special identifiers `as`, `qualified` and `hiding` have special meanings in certain contexts but can be used as ordinary identifiers.
- The final restriction on names is that names of constructors and types must begin with a capital letter; nothing else can do so, and hence we get error messages like

```
Errors.hs:6:0: Not in scope: data constructor 'Montana'
```

if we try to define a function called `Montana`.

Type errors

In this section we look at various different type errors that we can provoke in GHCi.

- As we have seen in the body of the text, the main type error we meet is exemplified by the response to typing `'c' && True` to the GHCi prompt:

```
Couldn't match expected type 'Bool' against inferred type 'Char'
In the first argument of '(&&)', namely 'c'
In the expression: 'c' && True
In the definition of 'it': it = 'c' && True
```

which is provoked by using a `Char` where an `Bool` is expected.

- Other type errors, such as

```
True + 4
```

provoke the error message

```
No instance for (Num Bool)
  arising from a use of '+' at <interactive>:1:0-7
Possible fix: add an instance declaration for (Num Bool)
In the expression: True + 4
In the definition of 'it': it = True + 4
```

This comes from the class mechanism: the system attempts to make `Bool` an instance of the class `Num` of numeric types over which '+' is defined. The error results since there is no such instance declaration making `Bool` belong to the class `Num`.

- As we said before, we can get type errors from syntax errors. For example, writing `abs -2` instead of `abs (-2)` gives the error message

```
No instance for (Num (a -> a))
  arising from a use of '-' at <interactive>:1:0-5
Possible fix: add an instance declaration for (Num (a -> a))
In the expression: abs - 2
In the definition of 'it': it = abs - 2
```

  because it is parsed as 2 subtracted from `abs::a->a`, and the operator '`-`' expects something in the class `Num`, rather than a function of type `a->a`. Other common type errors come from confusing the roles of '`:`' and '`++`' as in `2++[2]` and `[2]:[2]`.

- We always give type declarations for our definitions; one advantage of this is to spot when our definition does not conform to its declared type. For example,

```
myCheck :: Char -> Bool
myCheck n = toEnum n == 6
```

  gives the error message

```
Couldn't match expected type 'Int' against inferred type 'Char'
In the first argument of 'toEnum', namely 'n'
In the first argument of '(==)', namely 'toEnum n'
In the expression: toEnum n == 6
```

  Without the type declaration the definition would be accepted, only to give an error (presumably) when it is used.

- A definition like

```
asc x y
  | x <= y  = x y
```

  will give this error:

```
Occurs check: cannot construct the infinite type: a = a -> t
Probable cause: 'x' is applied to too many arguments
In the expression: x y
In the definition of 'asc': asc x y | x <= y = x y
```

  The problem here is that `x` is *compared with* `y` in the guard (type `a`), but *applied to* `y` in the body (type `a->t`): the unification required here produces an infinite type, which is not allowed.

- A final error related to types is given by definitions like

```
type Fred = (Fred,Int)                                    (Fred)
```

  a **recursive** type synonym; these are signalled by

```
Cycle in type synonym declarations:
  Errors.hs:9:0-21: type Fred = (Fred, Int)
```

  The effect of `(Fred)` can be modelled by the algebraic type definition

```
data Fred = Node Fred Int
```

  which introduces the **constructor** `Node` to identify objects of this type.

Program errors

Once we have written a syntactically and type correct script, and asked for the value of an expression which is itself acceptable, other errors can be produced during the **evaluation** of the expression.

- The first class of errors comes from missing cases in definitions. If we have written a definition like

  ```
  bat [] = 45
  ```

  and applied it to `[34]` we get the response

  ```
  *** Exception: Errors.hs:11:0-10: Non-exhaustive patterns
                 in function bat
  ```

  which shows the point at which evaluation can go no further, since there is no case in the definition of `bat` to cover a non-empty list. Similar errors come from built-in functions, such as `head`.
- Other errors happen because an **arithmetical constraint** has been broken. These include an out-of-range list index, division by zero, using a fraction where an integer is expected and floating-point calculations which go out of range; the error messages all have the same form. For example, suppose that we evaluate

  ```
  3 `div` 0
  ```

  then the error is

  ```
  *** Exception: divide by zero
  ```

- If we make a conformal definition, like

  ```
  [a,b] = [1 .. 10]
  ```

  this will fail with the message

  ```
  *** Exception: Errors.hs:13:0-16: Irrefutable pattern failed
                 for pattern [a, b]
  ```

  when either `a` or `b` is evaluated.

Module errors

The `module` and `import` statements can provoke a variety of error messages: files may not be present, or may contain errors; names may be included more than once, or an alias on inclusion may cause a name clash. The error messages for these and other errors are self-explanatory.

System messages

In response to some commands and interrupts, the system generates messages, including

- `^C ... Interrupted`
  signalling the interruption of the current task by typing `Ctrl-C`.

- The message

```
<interactive>: memory allocation failed (requested 2097152 bytes)
```

  which shows that the space consumption of the evaluation exceeds that available. One way around this is to increase the size of the heap.

A measure of the space complexity of a function, as described in Chapter 20, is given by the size of the smallest heap in which the evaluation can take place; how this 'residency' is measured is described in that chapter.

Appendix F
Project ideas

In this appendix we give some ideas for extended Haskell projects, building on what we have covered here. Most of the projects can be implemented using what you have learned in this text, but many would gain from using libraries on the Hackage site. The projects are also discussed in more detail in the online supplement to the text, which appears at `www.haskellcraft.com`.

Games and puzzles

Different games and puzzles give a whole lot of different challenges to the programmer. A game like noughts and crosses (tic-tac-toe) has an easy strategy which means that the first player can always avoid losing, whereas there is no machine implementation of Go that can compete with the best human players. Interacting with a game can be done textually – as people used to play chess by post – or through interactive graphics. Haskell has bindings to two fully-featured graphics/GUI libraries: `gtk` and `wx`, and these, as well as browser-based systems, can be used for implementations of various levels of sophistication.

We don't describe any of the games in any detail here: there are abundant web-based resources including, of course, Wikipedia, which cover the history and rules of the games as well as their many variants.

Sudoku

A first puzzle here is to *devise Sudoko problems*, which can then be printed and solved by hand. How do you find problems which fit the various levels of difficulty?

A second problem is to *solve a given problem*, as appears in many newspapers and online. As well as thinking of the algorithm which finds a solution efficiently, you'll need to think about how to input the problem and how to present the results. Finally, you could check whether the solution you find is unique: if not, can you enumerate or count the number of solutions? Conversely, are any seemingly consistent problems in fact unsolvable?

A third problem is to *provide assistance* to a human solver: can you give hints how they might make the next move when their solution has got to a certain point. Again you'll need to think about just how the interaction will take place. This could be part of an online solution assistant, or stand alone.

Minesweeper

The minesweeper program requires the player to uncover mines in a minefield without setting any of them off, when the player is given the count of mines on squares adjacent to each uncovered square. Many games are available online to provide examples.

One problem is to re-implement one of these *interactive games*: it could use 'text graphics', specifying the square to uncover by giving its coordinates, or could work interactively in a graphics system or browser.

Alternatively, you could implement an algorithm to *play the game automatically*: given a particular configuration, which is the optimal move to make next? How much information do you need to know to make this decision? Can you solve the problem in a deterministic way, or will your solution be probabilistic?

Noughts and crosses (tic-tac-toe)

When we looked at Rock – Paper – Scissors we saw that we could represent a *strategy* for a player as a function from their opponent's playing history to the player's next move. Try implementing noughts and crosses where two strategies play against each other: in doing this you will need to think about how to represent these strategies in a compact way, and also about how you might combine simple strategies together using *strategy combinators*.

A second problem is to implement an *interactive* version of the game in which a human player plays against a strategy. You will need to think about just how to present the interaction to the player, and in particular whether to us a textual, graphical or web browser-based representation of the game.

Web graphics

The web is moving towards a new standard, HTML5, which will directly render Scalable Vector Graphics (SVG) (SVG 2010) in web browsers. We have used SVG as a way of rendering our pictures, but in doing this we have barely scratched the surface of what is possible. The `gtk` package renders SVG using the Cairo system, but the aim of the projects discussed here is to use standard web technology to do the 'heavy lifting' of rendering.

Representing SVG

In this project you should devise a representation for a subset of SVG within Haskell. You might like to look at what has been done in `gtk`, and also at the different ways that XML is represented in HaXml and other Haskell libraries.

Rendering SVG

SVG can be rendered in place using a modern browser – in the case of Internet Explorer from version 9 upwards. Suppose that you want to generate SVG in a program, and have that data rendered in a browser: one mechanism for this is to run a local web server in which the page is created, and then served to a client on the local machine. Using the HTTP and web libraries for Haskell, build this local web server.

Web forms: calculator in a browser

Using the *web forms* standard in HTML5, you can build richer interactions, particularly using a local web server running in Haskell. In particular, you should be able to build a browser-embedded version of the calculator program, using Haskell to calculate the value of the input expression, and performing the interaction using a web form.

Alternatively, it is possible to build interactivity using JavaScript, either 'raw' or through the `jQuery` library.

Logic

Logic is a branch of mathematics closely linked to computer science (Huth and Ryan 2004), and implementing various logical procedures is a great way to understand precisely how logic works.

Truth tables

The first exercise is to *implement truth tables* for formulas of propositional logic. We can use these to decide whether a formula is a tautology (true in *all* interpretations) or satisfiable (true in *at least one* interpretation). The output could just be this classification, or it could be the full truth table, showing the value of the formula under each interpretation.

SAT solving

The practicalities of deciding whether or not a formula is satisfiable, the *SAT solving* problem, is an area where huge efforts are made to find efficient algorithms to solve the problem. How can these algorithms be implemented in Haskell?

Tableaux

Semantic tableaux give a decision procedure not just for propositional logic, but also for temporal and other modal logics. A tableau method is a *constructive* mechanism, which will find all the interpretations satisfying a particular formula. This project is to implement a *tableau procedure* for propositional logic; you can also look at tableaux for the temporal logics in (Huth and Ryan 2004).

Proof

Rather than using an automated mechanism to decide whether or not a formula is a tautology, it is possible to *prove* a formula in a formal system: this project is to implement an interactive proof system for propositional logic. You will need to decide how the interactions are to be modelled in the system, and how feedback can be provided to the user about their proof choices, as well as how advice can be given to them on request.

Voting systems

Depending on where you live, you will decide the form of your government in different ways. The simplest mechanism is perhaps *first past the post*, where the person gaining

the single largest number of votes is elected, but other systems include ways of getting a more *proportional* outcome than this.

How is it possible to explain these various mechanisms to a naïve voter? The aim of this exercise is to illustrate the effects of different systems, based on a simulated vote.

Voting data

It should be possible to get *actual voting data* from particular elections, and to use (processed?) variants of those data in different voting scenarios. Alternatively you should generate random data according to a scenario specified by a user. This could be done from scratch, or alternatively you could use the data generation facilities of QuickCheck to build a sample.

Visualization

'A picture is worth a thousand words', and in a situation it makes sense to provide *visualizations* of the various results. You could use the SVG facilities developed in an earlier suggestion, or investigate the capabilities of the graphical libraries provided within Hackage.

Tactical voting

What is the best way to get the result that you want? On the basis of historical data and the particular system, analyse the options for a voter who wants a particular outcome to happen. For instance, what is the best option for your second vote in an AV system?

Finite-state machines

One of the fundamental abstractions in computer science is the *finite-state machine* (FSM) (Aho, Lam, Sethi and Ullman 2006). We saw earlier that we can write recognizers for regular expressions, but more efficient implementations are given by deriving NFAs from regular expressions, and then (minimal) DFAs from those NFAs.

Conversion chain

As an exercise, implement the chain of conversions from regular expressions to NFAs, DFAs and finally optimal DFAs; there are also algorithms which convert directly from a regular expression to a DFA.

Inferring machines

There is a well-developed literature on deriving machines, regular expressions or grammars from sets of traces which are accepted (or rejected) by a particular machine. Investigate these further by implementing them.

Visualization

Develop mechanisms which provide a *visualization* of the operation of an FSM. and of the algorithms discussed earlier.

Domain-specific languages

One theme of this book has been domain-specific languages, and as a part of some of these projects you could build a domain-specific language. Examples include

- A language for describing games has been defined by Conway (Conway 2002; Berlekamp, Conway and Guy 2001): look at how you can build a DSL for these games; you could also look at a language for describing strategies to play these games.
- A language for describing different voting systems: your simulations and visualizations could then work with an arbitrary voting system, as described in the language.
- We saw in the body of the text that it is possible to write a simple DSL for patterns, namely regular expressions. Look at ways that this can be extended to make it more expressible, and also at the possibility of defining a DSL to describe different kinds of finite state machines.

These are just a few ideas of the kind of DSL that you could build: a general project is to use Haskell for building DSLs in a domain of your choice.