THINKING FUNCTIONALLY WITH HASKELL

Richard Bird is famed for the clarity and rigour of his writing. His new textbook, which introduces functional programming to students, emphasises fundamental techniques for reasoning mathematically about functional programs. By studying the underlying equational laws, the book enables students to apply calculational reasoning to their programs, both to understand their properties and to make them more efficient.

The book has been designed to fit a first- or second-year undergraduate course and is a thorough overhaul and replacement of his earlier textbooks. It features case studies in Sudoku and pretty-printing, and over 100 carefully selected exercises with solutions. This engaging text will be welcomed by students and teachers alike.
## Contents

**Preface**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 What is functional programming?</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Functions and types</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Functional composition</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Example: common words</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Example: numbers into words</td>
<td>7</td>
</tr>
<tr>
<td>1.5 The Haskell Platform</td>
<td>12</td>
</tr>
<tr>
<td>1.6 Exercises</td>
<td>14</td>
</tr>
<tr>
<td>1.7 Answers</td>
<td>17</td>
</tr>
<tr>
<td>1.8 Chapter notes</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 Expressions, types and values</td>
<td>22</td>
</tr>
<tr>
<td>2.1 A session with GHCi</td>
<td>22</td>
</tr>
<tr>
<td>2.2 Names and operators</td>
<td>25</td>
</tr>
<tr>
<td>2.3 Evaluation</td>
<td>27</td>
</tr>
<tr>
<td>2.4 Types and type classes</td>
<td>30</td>
</tr>
<tr>
<td>2.5 Printing values</td>
<td>33</td>
</tr>
<tr>
<td>2.6 Modules</td>
<td>35</td>
</tr>
<tr>
<td>2.7 Haskell layout</td>
<td>36</td>
</tr>
<tr>
<td>2.8 Exercises</td>
<td>37</td>
</tr>
<tr>
<td>2.9 Answers</td>
<td>42</td>
</tr>
<tr>
<td>2.10 Chapter notes</td>
<td>47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 Numbers</td>
<td>49</td>
</tr>
<tr>
<td>3.1 The type class Num</td>
<td>49</td>
</tr>
<tr>
<td>3.2 Other numeric type classes</td>
<td>50</td>
</tr>
<tr>
<td>3.3 Computing floors</td>
<td>52</td>
</tr>
<tr>
<td>3.4 Natural numbers</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>8 Pretty-printing</td>
</tr>
<tr>
<td>----</td>
<td>------------------</td>
</tr>
<tr>
<td>7.6</td>
<td>Tupling</td>
</tr>
<tr>
<td>7.7</td>
<td>Sorting</td>
</tr>
<tr>
<td>7.8</td>
<td>Exercises</td>
</tr>
<tr>
<td>7.9</td>
<td>Answers</td>
</tr>
<tr>
<td>7.10</td>
<td>Chapter notes</td>
</tr>
<tr>
<td>8</td>
<td>Setting the scene</td>
</tr>
<tr>
<td>8.1</td>
<td>Documents</td>
</tr>
<tr>
<td>8.2</td>
<td>A direct implementation</td>
</tr>
<tr>
<td>8.3</td>
<td>Examples</td>
</tr>
<tr>
<td>8.4</td>
<td>The best layout</td>
</tr>
<tr>
<td>8.5</td>
<td>A term representation</td>
</tr>
<tr>
<td>8.6</td>
<td>Exercises</td>
</tr>
<tr>
<td>8.7</td>
<td>Answers</td>
</tr>
<tr>
<td>8.8</td>
<td>Chapter notes</td>
</tr>
<tr>
<td>9</td>
<td>Infinite lists</td>
</tr>
<tr>
<td>9.1</td>
<td>Review</td>
</tr>
<tr>
<td>9.2</td>
<td>Cyclic lists</td>
</tr>
<tr>
<td>9.3</td>
<td>Infinite lists as limits</td>
</tr>
<tr>
<td>9.4</td>
<td>Paper–rock–scissors</td>
</tr>
<tr>
<td>9.5</td>
<td>Stream-based interaction</td>
</tr>
<tr>
<td>9.6</td>
<td>Doubly-linked lists</td>
</tr>
<tr>
<td>9.7</td>
<td>Exercises</td>
</tr>
<tr>
<td>9.8</td>
<td>Answers</td>
</tr>
<tr>
<td>9.9</td>
<td>Chapter notes</td>
</tr>
<tr>
<td>10</td>
<td>Imperative functional programming</td>
</tr>
<tr>
<td>10.1</td>
<td>The IO monad</td>
</tr>
<tr>
<td>10.2</td>
<td>More monads</td>
</tr>
<tr>
<td>10.3</td>
<td>The State monad</td>
</tr>
<tr>
<td>10.4</td>
<td>The ST monad</td>
</tr>
<tr>
<td>10.5</td>
<td>Mutable arrays</td>
</tr>
<tr>
<td>10.6</td>
<td>Immutable arrays</td>
</tr>
<tr>
<td>10.7</td>
<td>Exercises</td>
</tr>
<tr>
<td>10.8</td>
<td>Answers</td>
</tr>
<tr>
<td>10.9</td>
<td>Chapter notes</td>
</tr>
<tr>
<td>11</td>
<td>Parsing</td>
</tr>
<tr>
<td>11.1</td>
<td>Parsers as monads</td>
</tr>
<tr>
<td>11.2</td>
<td>Basic parsers</td>
</tr>
<tr>
<td>11.3</td>
<td>Choice and repetition</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>11.4</td>
<td>Grammars and expressions</td>
</tr>
<tr>
<td>11.5</td>
<td>Showing expressions</td>
</tr>
<tr>
<td>11.6</td>
<td>Exercises</td>
</tr>
<tr>
<td>11.7</td>
<td>Answers</td>
</tr>
<tr>
<td>11.8</td>
<td>Chapter notes</td>
</tr>
<tr>
<td>12</td>
<td>A simple equational calculator</td>
</tr>
<tr>
<td>12.1</td>
<td>Basic considerations</td>
</tr>
<tr>
<td>12.2</td>
<td>Expressions</td>
</tr>
<tr>
<td>12.3</td>
<td>Laws</td>
</tr>
<tr>
<td>12.4</td>
<td>Calculations</td>
</tr>
<tr>
<td>12.5</td>
<td>Rewrites</td>
</tr>
<tr>
<td>12.6</td>
<td>Matchings</td>
</tr>
<tr>
<td>12.7</td>
<td>Substitutions</td>
</tr>
<tr>
<td>12.8</td>
<td>Testing the calculator</td>
</tr>
<tr>
<td>12.9</td>
<td>Exercises</td>
</tr>
<tr>
<td>12.10</td>
<td>Answers</td>
</tr>
<tr>
<td>12.11</td>
<td>Chapter notes</td>
</tr>
<tr>
<td></td>
<td>Index</td>
</tr>
</tbody>
</table>
Preface

The present book is a completely rewritten version of the second edition of my Introduction to Functional Programming using Haskell (Prentice Hall). The main changes are: a reorganisation of some introductory material to reflect the needs of a one or two term lecture course; a fresh set of case studies; and a collection of over 100 exercises that now actually contain answers. As before, no knowledge of computers or programming is assumed, so the material is suitable as a first course in computing.

Every author has his or her own drum to beat when writing a textbook, and the present one is no different. While there are now numerous books, tutorials, articles and blogs devoted to Haskell, few of them emphasise what seems to me the main reason why functional programming is the best thing since sliced bread: the ability to think mathematically about functional programs. And the mathematics involved is neither new nor difficult. Any student who has come to grips with, say, high-school trigonometry and has applied simple trigonometric laws and identities to simplify expressions involving sines and cosines (a typical example: express $\sin 3\alpha$ in terms of $\sin \alpha$) will quickly appreciate that a similar activity is being proposed for programming problems. And the payoff is there at the terminal: faster computations. Even after 30 years I still get a great deal of pleasure from writing down a simple, obvious, but inefficient way to solve a problem, applying some well-known equational laws, and coming up with another solution that is ten times faster. Well, if I’m lucky.

If the message of the last paragraph turns you off, if you are perpetually running away from the Mordor of Mathematics, then the present book is probably not for you. Probably, but not necessarily so (nobody likes to lose customers). There is still pleasure to be gained in learning a novel and exciting way to write programs. Even programmers who for one reason or another do not or cannot use Haskell
in their daily work, and certainly do not have the time to spend calculating better answers to their problems, have still been inspired by the enjoyment of learning Haskell and are hugely appreciative of its ability to express computational ideas and methods simply and briefly. In fact, the ability to express programming ideas in a purely functional style has been slowly incorporated into mainstream imperative programming languages, such as Python, Visual Basic, and C#.

One final but important point: Haskell is a large language and this book by no means covers all of it. It is not a reference guide to Haskell. Although details of the language appear on almost every page, especially in the earlier chapters, my primary intention is to convey the essence of functional programming, the idea of thinking functionally about programs, not to dwell too much on the particulars of one specific language. But over the years Haskell has absorbed and codified most of the ideas of functional programming expressed in earlier functional languages, such as SASL, KRC, Miranda, Orwell and Gofer, and it is difficult to resist the temptation to explain everything in terms of this one super-cool language.

Most of the programs recorded in this book can be found on the website

www.cs.ox.ac.uk/publications/books/functional

It is hoped to add more exercises (and answers), suggestions for projects, and so on, in due course. For more information about Haskell, the site www.haskell.org should be your first port of call.

Acknowledgements

The present book arose out of lecture notes I prepared based on the second edition. It has benefited enormously from the comments and suggestions by tutors and students. Others have emailed me with constructive comments and criticisms, or simply to point out typos and silly mistakes. These include: Nils Andersen, Ani Calinescu, Franklin Chen, Sharon Curtis, Martin Filby, Simon Finn, Jeroen Fokker, Maarten Fokkinga, Jeremy Gibbons, Robert Giegerich, Kevin Hammond, Ralf Hinze, Gerard Huet, Michael Hinchey, Tony Hoare, Iain Houston, John Hughes, Graham Hutton, Cezar Ionescu, Stephen Jarvis, Geraint Jones, Mark Jones, John Launchbury, Paul Licameli, David Lester, Iain MacCullum, Ursula Martin, Lambert Meertens, Erik Meijer, Quentin Miller, Oege de Moor, Chris Okasaki, Oskar Permvall, Simon Peyton Jones, Mark Ramaer, Hamilton Richards, Dan Russell, Don Sannella, Antony Simmons, Deepak D’Souza, John Spanondakis, Mike Spivey, Joe Stoy, Bernard Sufrin, Masato Takeichi, Peter Thiemann, David Turner, Colin Watson, and Stephen Wilson. In particular, Jeremy Gibbons, Bernard Sufrin
and José Pedro Magalhães have read drafts of the manuscript and suggested a number of corrections.

I would also like to thank David Tranah, my editor at CUP, for continued advice and support. My status now is emeritus professor at the Department of Computer Science at Oxford, and I would like to thank the department and its head, Bill Roscoe, for continuing to make facilities available.

Richard Bird

Exercises

Exercise A

Express \( \sin 3\alpha \) in terms of \( \sin \alpha \).

Answers

Answer to Exercise A

\[
\begin{align*}
\sin 3\alpha \\
= & \quad \{ \text{arithmetic} \} \\
= & \quad \sin(2\alpha + \alpha) \\
= & \quad \{ \text{since } \sin(\alpha + \beta) = \sin \alpha \cos \beta + \cos \alpha \sin \beta \} \\
= & \quad \sin 2\alpha \cos \alpha + \cos 2\alpha \sin \alpha \\
= & \quad \{ \text{since } \sin 2\alpha = 2 \sin \alpha \cos \alpha \} \\
= & \quad 2 \sin \alpha \cos^2 \alpha + \cos 2\alpha \sin \alpha \\
= & \quad \{ \text{since } \cos 2\alpha = \cos^2 \alpha - \sin^2 \alpha \} \\
= & \quad 2 \sin \alpha \cos^2 \alpha + (\cos^2 \alpha - \sin^2 \alpha) \sin \alpha \\
= & \quad \{ \text{since } \sin^2 \alpha + \cos^2 \alpha = 1 \} \\
= & \quad \sin \alpha (3 - 4 \sin^2 \alpha)
\end{align*}
\]

The above proof format was, I believe, invented by Wim Feijen. It will be used throughout the book.
Chapter 1

What is functional programming?

In a nutshell:

- Functional programming is a method of program construction that emphasises functions and their application rather than commands and their execution.
- Functional programming uses simple mathematical notation that allows problems to be described clearly and concisely.
- Functional programming has a simple mathematical basis that supports equational reasoning about the properties of programs.

Our aim in this book is to illustrate these three key points, using a specific functional language called Haskell.

1.1 Functions and types

We will use the Haskell notation

\[ f : : X \rightarrow Y \]

to assert that \( f \) is a function taking arguments of type \( X \) and returning results of type \( Y \). For example,

- \( \text{sin} :: \text{Float} \rightarrow \text{Float} \)
- \( \text{age} :: \text{Person} \rightarrow \text{Int} \)
- \( \text{add} :: (\text{Integer},\text{Integer}) \rightarrow \text{Integer} \)
- \( \text{logBase} :: \text{Float} \rightarrow (\text{Float} \rightarrow \text{Float}) \)

Float is the type of floating-point numbers, things like 3.14159, and Int is the type of limited-precision integers, integers \( n \) that lie in a restricted range such as
What is functional programming?

\(-2^{29} \leq n < 2^{29}\). The restriction is lifted with the type \texttt{Integer}, which is the type of unlimited-precision integers. As we will see in Chapter 3, numbers in Haskell come in many flavours.

In mathematics one usually writes \(f(x)\) to denote the application of the function \(f\) to the argument \(x\). But we also write, for example, \(\sin \theta\) rather than \(\sin(\theta)\). In Haskell we can always write \(f \; x\) for the application of \(f\) to the argument \(x\). The operation of application can be denoted using a space. If there are no parentheses the space is necessary to avoid confusion with multi-letter names: \texttt{latex} is a name but \texttt{late \; x} denotes the application of a function \texttt{late} to an argument \(x\).

As examples, \(\sin \; 3.14\) or \(\sin \; (3.14)\) or \(\sin(3.14)\) are three legitimate ways of writing the application of the function \(\sin\) to the argument \(3.14\).

Similarly, \(\logBase \; 2 \; 10\) or \((\logBase \; 2) \; 10\) or \((\logBase \; 2)(10)\) are all legitimate ways of writing the logarithm to base 2 of the number 10. But the expression \(\logBase \; (2 \; 10)\) is incorrect. Parentheses are needed in writing \(\texttt{add} \; (3,4)\) for the sum of 3 and 4 because the argument of \(\texttt{add}\) is declared above as a pair of integers and pairs are expressed with parentheses and commas.

Look again at the type of \texttt{logBase}. It takes a floating point number as argument, and returns a function as result. At first sight that might seem strange, but at second sight it shouldn’t: the mathematical functions \(\log_2\) and \(\log_e\) are exactly what is provided by \texttt{logBase \; 2} and \texttt{logBase \; e}.

In mathematics one can encounter expressions like \(\log \sin \; x\). To the mathematician that means \(\log(\sin \; x)\), since the alternative \((\log \sin) \; x\) doesn’t make sense. But in Haskell one has to say what one means, and one has to write \(\log \; (\sin \; x)\) because \(\log \; \sin \; x\) is read by Haskell as \((\log \; \sin) \; x\). Functional application in Haskell \textit{associates} to the left in expressions and also has the highest \textit{binding power}. (By the way, \(\log\) is the Haskell abbreviation for \texttt{logBase \; e}.)

Here is another example. In trigonometry one can write

\[
\sin 2\theta = 2 \sin \theta \cos \theta.
\]

In Haskell one has to write

\[
\sin \; (2*\texttt{theta}) = 2 \; * \; \texttt{sin \; theta} \; * \; \texttt{cos \; theta}
\]

Not only do we have to make the multiplications explicit, we also have to put in parentheses to say exactly what we mean. We could have added a couple more and written

\[
\sin \; (2*\texttt{theta}) = 2 \; * \; (\texttt{sin \; theta}) \; * \; (\texttt{cos \; theta})
\]
1.2 Functional composition

Suppose \( f : : Y \rightarrow Z \) and \( g : : X \rightarrow Y \) are two given functions. We can combine them into a new function

\[ f \cdot g : : X \rightarrow Z \]

that first applies \( g \) to an argument of type \( X \), giving a result of type \( Y \), and then applies \( f \) to this result, giving a final result of type \( Z \). We always say that functions take \textit{arguments} and return \textit{results}. In fact we have

\[ (f \cdot g) x = f (g x) \]

The order of composition is from right to left because we write functions to the left of the arguments to which they are applied. In English we write ‘green pig’ and interpret adjectives such as ‘green’ as functions taking noun phrases to noun phrases. Of course, in French . . .

1.3 Example: common words

Let us illustrate the importance of functional composition by solving a problem. What are the 100 most common words in \textit{War and Peace}? What are the 50 most common words in \textit{Love’s Labours Lost}? We will write a functional program to find out. Well, perhaps we are not yet ready for a complete program, but we can construct enough of one to capture the essential spirit of functional programming.

What is given? Answer: a \textit{text}, which is a list of characters, containing visible characters like ‘B’ and ‘,’ and blank characters like spaces and newlines (‘ ’ and ‘\n’). Note that individual characters are denoted using single quotes. Thus ‘f’ is a character, while \( f \) is a name. The Haskell type \texttt{Char} is the type of characters, and the type of lists whose elements are of type \texttt{Char} is denoted by \texttt{[Char]}. This notation is not special to characters, so \texttt{[Int]} denotes a list of integers, and \texttt{[Float \rightarrow Float]} a list of functions.

What is wanted as output? Answer: something like

\[
\begin{align*}
\text{the: } & 154 \\
\text{of: } & 50
\end{align*}
\]
What is functional programming?

This display is also a list of characters, in fact it is the list

" the: 154\n of: 50\n a: 18\n and: 12\n in: 11\n"

Lists of characters are denoted using double quotes. More on this in the exercises.

So we want to design a function, commonWords say, with type

```haskell
commonWords :: Int -> [Char] -> [Char]
```

The function commonWords n takes a list of characters and returns a list of the n most common words in the list as a string (another name for a list of characters) in the form described above. The type of commonWords is written without parentheses, though we can put them in:

```haskell
commonWords :: Int -> ([Char] -> [Char])
```

Whenever two -> signs are adjacent in a type, the order of association is from right to left, exactly the opposite convention of functional application. So A -> B -> C means A -> (B -> C). If you want to describe the type (A -> B) -> C you have to put in the parentheses. More on this in the next chapter.

Having understood precisely what is given and what is wanted, different people come up with different ways of solving the problem, and express different worries about various parts of the problem. For example, what is a ‘word’ and how do you convert a list of characters into a list of words? Are the words "Hello", "hello" and "Hello!" distinct words or the same word? How do you count words? Do you count all the words or just the most common ones? And so on. Some find these details daunting and overwhelming. Most seem to agree that at some intermediate point in the computation we have to come up with a list of words and their frequencies, but how do we get from there to the final destination? Do we go through the list n times, extracting the word with the next highest frequency at each pass, or is there something better?

Let’s start with what a word is, and just assert that a word is a maximal sequence of characters not containing spaces or newline characters. That allows words like "Hello!", or "3*4" or "Thelma&Louise" but never mind. In a text a word is identified by being surrounded by blank characters, so "Thelma and Louise" contains three words.
We are not going to worry about how to split a text up into a list of its component words. Instead we just assume the existence of a function

\[
\text{words} :: \text{[Char]} \to \text{[[Char]]}
\]

that does the job. Types like \text{[[Char]]} can be difficult to comprehend, but in Haskell we can always introduce \textit{type synonyms}:

\[
\text{type Text} = \text{[Char]} \\
\text{type Word} = \text{[Char]}
\]

So now we have \text{words} :: \text{Text} \to \text{[Word]}, which is much easier on the brain. Of course, a text is different from a word in that the former can contain blank characters and the latter cannot, but type synonyms in Haskell do not support such subtle distinctions. In fact, \text{words} is a library function in Haskell, so we don’t have to define it ourselves.

There is still the issue of whether "The" and "the" denote the same or different words. They really should be the same word, and one way of achieving this is to convert all the letters in the text to lowercase, leaving everything else unchanged. To this end, we need a function \text{toLower} :: \text{Char} \to \text{Char} that converts uppercase letters to lowercase and leaves everything else unchanged. In order to apply this function to every character in the text we need a general function

\[
\text{map} :: (a \to b) \to \text{[a]} \to \text{[b]}
\]

such that \text{map f} applied to a list applies \(f\) to every element of the list. So, converting everything to lowercase is done by the function

\[
\text{map toLower} :: \text{Text} \to \text{Text}
\]

Good. At this point we have \text{words \ . map toLower} as the function which converts a text into a list of words in lowercase. The next task is to count the number of occurrences of each word. We could go through the list of words, checking to see whether the next word is new or has been seen before, and either starting a new count for a new word or incrementing the count for an existing word. But there is a conceptually simpler method, namely to \textit{sort} the list of words into alphabetical order, thereby bringing all duplicated words together in the list. Humans would not do it this way, but the idea of sorting a list to make information available is probably the single most important algorithmic idea in computing. So, let us assume the existence of a function

\[
\text{sortWords} :: \text{[Word]} \to \text{[Word]}
\]

that sorts the list of words into alphabetical order. For example,
What is functional programming?

```
sortWords = ["to","be","or","not","to","be"]
  = ["be","be","not","or","to","to"]
```

Now we want to count the runs of adjacent occurrences of each word in the sorted list. Suppose we have a function

```
countRuns :: [Word] -> [(Int,Word)]
```

that counts the words. For example,

```
countRuns = [(2,"be"),(1,"not"),(1,"or"),(2,"to")]
```

The result is a list of words and their counts in alphabetical order of the words.

Now comes the key idea: we want the information in the list to be ordered not by word, but by decreasing order of count. Rather than thinking of something more clever, we see that this is just another version of sorting. As we said above, sorting is a really useful method in programming. So suppose we have a function

```
sortRuns :: [(Int,Word)] -> [(Int,Word)]
```

that sorts the list of runs into descending order of count (the first component of each element). For example,

```
sortRuns = [(2,"be"),(2,"to"),(1,"not"),(1,"or")]
```

The next step is simply to take the first \( n \) elements of the result. For this we need a function

```
take :: Int -> [a] -> [a]
```

so that \( \text{take}\ n \) takes the first \( n \) elements of a list of things. As far as \( \text{take}\) is concerned it doesn’t matter what a ‘thing’ is, which is why there is an \( a \) in the type signature rather than \( (\text{Int},\text{Word}) \). We will explain this idea in the next chapter.

The final steps are just tidying up. We first need to convert each element into a string so that, for example, \( (2,"be") \) is replaced by "be 2\n". Call this function

```
showRun :: (Int,Word) -> String
```

The type \( \text{String} \) is a predeclared Haskell type synonym for \([\text{Char}]\). That means

```
map showRun :: [(Int,Word)] -> [String]
```

is a function that converts a list of runs into a list of strings.

The final step is to use a function
concat :: [[a]] -> [a]

that concatenates a list of lists of things together. Again, it doesn’t matter what the ‘thing’ is as far as concatenation is concerned, which is why there is an \( a \) in the type signature.

Now we can define

\[
\text{commonWords} :: \text{Int} \to \text{Text} \to \text{String} \\
\text{commonWords} \ n = \text{concat} \ . \ \text{map} \ \text{showRun} \ . \ \text{take} \ n \ . \\
\quad \text{sortRuns} \ . \ \text{countRuns} \ . \ \text{sortWords} \ . \\
\quad \text{words} \ . \ \text{map} \ \text{toLowerCase}
\]

The definition of \( \text{commonWords} \) is given as a pipeline of eight component functions glued together by functional composition. Not every problem can be decomposed into component tasks in quite such a straightforward manner, but when it can, the resulting program is simple, attractive and effective.

Notice how the process of decomposing the problem was governed by the declared types of the subsidiary functions. Lesson Two (Lesson One being the importance of functional composition) is that deciding on the type of a function is the very first step in finding a suitable definition of the function.

We said above that we were going to write a program for the common words problem. What we actually did was to write a functional definition of \( \text{commonWords} \), using subsidiary definitions that we either can construct ourselves or else import from a suitable Haskell library. A list of definitions is called a script, so what we constructed was a script. The order in which the functions are presented in a script is not important. We could place the definition of \( \text{commonWords} \) first, and then define the subsidiary functions, or else define all these functions first, and end up with the definition of the main function of interest. In other words we can tell the story of the script in any order we choose. We will see how to compute with scripts later on.

1.4 Example: numbers into words

Here is another example, one for which we will provide a complete solution. The example demonstrates another fundamental aspect of problem solving, namely that a good way to solve a tricky problem is to first simplify the problem and then see how to solve the simpler problem.

Sometimes we need to write numbers as words. For instance
What is functional programming?

```haskell
convert 308000 = "three hundred and eight thousand"
convert 369027 = "three hundred and sixty-nine thousand and twenty-seven"
convert 369401 = "three hundred and sixty-nine thousand four hundred and one"
```

Our aim is to design a function

```haskell
convert :: Int -> String
```

that, given a nonnegative number less than one million, returns a string that represents the number in words. As we said above, String is a predeclared type synonym in Haskell for [Char].

We will need the names of the component numbers. One way is to give these as three lists of strings:

```haskell
> units, teens, tens :: [String]
> units = ["zero","one","two","three","four","five",
>       "six","seven","eight","nine"]
> teens = ["ten","eleven","twelve","thirteen","fourteen",
>       "fifteen","sixteen","seventeen","eighteen",
>       "nineteen"]
> tens = ["twenty","thirty","forty","fifty","sixty",
>       "seventy","eighty","ninety"]
```

Oh, what is the > character doing at the beginning of each line above? The answer is that, in a script, it indicates a line of Haskell code, not a line of comment. In Haskell, a file ending with the suffix .lhs is called a Literate Haskell Script and the convention is that every line in such a script is interpreted as a comment unless it begins with a > sign, when it is interpreted as a line of program. Program lines are not allowed next to comments, so there has to be at least one blank line separating the two. In fact, the whole chapter you are now reading forms a legitimate .lhs file, one that can be loaded into a Haskell system and interacted with. We won’t carry on with this convention in subsequent chapters (apart from anything else, it would force us to use different names for each version of a function that we may want to define) but the present chapter does illustrate literate programming in which we can present and discuss the definitions of functions in any order we wish.

Returning to the task in hand, a good way to tackle tricky problems is to solve a simpler problem first. The simplest version of our problem is when the given number \( n \) contains only one digit, so \( 0 \leq n < 10 \). Let \( \text{convert1} \) deal with this version. We can immediately define
1.4 Example: numbers into words

> convert1 :: Int -> String
> convert1 n = units!!n

This definition uses the list-indexing operation (!!). Given a list xs and an index n, the expression xs!!n returns the element of xs at position n, counting from 0. In particular, units!!0 = "zero". And, yes, units!!10 is undefined because units contains just ten elements, indexed from 0 to 9. In general, the functions we define in a script are partial functions that may not return well-defined results for each argument.

The next simplest version of the problem is when the number n has up to two digits, so 0 ≤ n < 100. Let convert2 deal with this case. We will need to know what the digits are, so we first define

> digits2 :: Int -> (Int,Int)
> digits2 n = (div n 10, mod n 10)

The number div n k is the whole number of times k divides into n, and mod n k is the remainder. We can also write

    digits2 n = (n `div` 10, n `mod` 10)

The operators `div` and `mod` are infix versions of div and mod, that is, they come between their two arguments rather than before them. This device is useful for improving readability. For instance a mathematician would write \(x \div y\) and \(x \mod y\) for these expressions. Note that the back-quote symbol ` is different from the single quote symbol ' used for describing individual characters.

Now we can define

> convert2 :: Int -> String
> convert2 = combine2 . digits2

The definition of combine2 uses the Haskell syntax for guarded equations:

> combine2 :: (Int,Int) -> String
> combine2 (t,u)
> | t==0 = units!!u
> | t==1 = teens!!u
> | 2<=t && u==0 = tens!!(t-2)
> | 2<=t && u/=0 = tens!!(t-2) ++ "-" ++ units!!u

To understand this code you need to know that the Haskell symbols for equality and comparison tests are as follows:
What is functional programming?

==  (equals to)
/=  (not equals to)
<=  (less than or equal to)

These functions have well-defined types that we will give later on.

You also need to know that the conjunction of two tests is denoted by &&. Thus a && b returns the boolean value True if both a and b do, and False otherwise. In fact

(&&) :: Bool -> Bool -> Bool

The type Bool will be described in more detail in the following chapter.

Finally, (++) denotes the operation of concatenating two lists. It doesn’t matter what the type of the list elements is, so

(++) :: [a] -> [a] -> [a]

For example, in the equation

[sin,cos] ++ [tan] = [sin,cos,tan]

we are concatenating two lists of functions (each of type Float -> Float), while in

"sin cos" ++ " tan" = "sin cos tan"

we are concatenating two lists of characters.

The definition of combine2 is arrived at by carefully considering all the possible cases that can arise. A little reflection shows that there are three main cases, namely when the tens part $t$ is 0, 1 or greater than 1. In the first two cases we can give the answer immediately, but the third case has to be divided into two subcases, namely when the units part $u$ is 0 or not 0. The order in which we write the cases, that is, the order of the individual guarded equations, is unimportant as the guards are disjoint from one another (that is, no two guards can be true) and together they cover all cases.

We could also have written

combine2 :: (Int,Int) -> String
combine2 (t,u)
| t==0      = units!!u
| t==1      = teens!!u
| u==0      = tens!!(t-2)
| otherwise = tens!!(t-2) ++ "-" ++ units!!u
but now the order in which we write the equations is crucial. The guards are evaluated from top to bottom, taking the right-hand side corresponding to the first guard that evaluates to True. The identifier otherwise is just a synonym for True, so the last clause captures all the remaining cases.

There is yet another way of writing convert2:

```haskell
convert2 :: Int -> String
convert2 n
| t==0       = units!!u
| t==1       = teens!!u
| u==0       = tens!!(t-2)
| otherwise  = tens!!(t-2) ++ "-" ++ units!!u
where (t,u) = (n `div` 10, n `mod` 10)
```

This makes use of a where clause. Such a clause introduces a local definition or definitions whose context or scope is the whole of the right-hand side of the definition of convert2. Such clauses are very useful in structuring definitions and making them more readable. In the present example, the where clause obviates the need for an explicit definition of digits2.

That was reasonably easy, so now let us consider convert3 which takes a number n in the range 0 ≤ n < 1000, so n has up to three digits. The definition is

```haskell
> convert3 :: Int -> String
> convert3 n
> | h==0       = convert2 t
> | n==0       = units!!h ++ " hundred"
> | otherwise  = units!!h ++ " hundred and " ++ convert2 t
> where (h,t) = (n `div` 100, n `mod` 100)
```

We break up the number in this way because we can make use of convert2 for numbers that are less than 100.

Now suppose n lies in the range 0 ≤ n < 1,000,000, so n can have up to six digits. Following exactly the same pattern as before, we can define

```haskell
> convert6 :: Int -> String
> convert6 n
> | m==0       = convert3 h
> | h==0       = convert3 m ++ " thousand"
> | otherwise  = convert3 m ++ " thousand" ++ link h ++
>               convert3 h
> where (m,h) = (n `div` 1000,n `mod` 1000)
```
There will be a connecting word ‘and’ between the words for \( m \) and \( h \) just in the case that \( 0 < m \) and \( 0 < h < 100 \). Thus

\[
\text{link} :: \text{Int} \to \text{String} \\
\text{link} \ h = \text{if } h \ < \ 100 \ \text{then } " \text{and } " \ \text{else } " \ "
\]

This definition makes use of a conditional expression

\[
\text{if } \langle \text{test} \rangle \ \text{then } \langle \text{expr1} \rangle \ \text{else } \langle \text{expr2} \rangle
\]

We could also have used guarded equations:

\[
\text{link} \ h | h < 100 = " \text{and } " \\
| \text{otherwise} = " "
\]

Sometimes one is more readable, sometimes the other. The names if, then and else, along with some others, are reserved words in Haskell, which means that we cannot use them as names for things we want to define.

Notice how the definition of convert6 has been constructed in terms of the simpler function convert3, which in turn has been defined in terms of the even simpler function convert2. That is often the way with function definitions. In this example consideration of the simpler cases is not wasted because these simple cases can be used in the final definition.

One more thing: we have now named the function we are after as convert6, but we started off by saying the name should be convert. No problem:

\[
\text{convert} :: \text{Int} \to \text{String} \\
\text{convert} = \text{convert6}
\]

What we would like to do now is actually use the computer to apply convert to some arguments. How?

1.5 The Haskell Platform

If you visit the site \url{www.haskell.org}, you will see how to download The Haskell Platform. This is a large collection of tools and packages that can be used to run Haskell scripts. The platform comes in three versions, one for each of Windows, Mac and Linux. We deal only with the Windows version, the others being similar.

One of the tools is an interactive calculator, called GHCi. This is short for Glasgow Haskell Compiler Interpreter. The calculator is available as a Windows system called WinGHCi. If you open this window, you will get something like
1.5 The Haskell Platform

GHCi, version 7.6.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>

The prompt Prelude> means that the standard library of prelude functions, pre-declared types and other values is loaded. You can now use GHCi as a super-calculator:

Prelude> 3^5
243
Prelude> import Data.Char
Prelude Data.Char> map toLower "HELLO WORLD!"
"hello world!"
Prelude Data.Char>

The function toLower resides in the library Data.Char. After importing this library you have access to the functions defined in the library. Note that the prompt changes and now indicates the libraries that have been loaded. Such prompts can grow in size very quickly. But we can always change the prompt:

Prelude> :set prompt ghci>
ghci>

For brevity we will use this prompt throughout the book.

You can load a script, Numbers2Words.lhs say, that contains the definition of convert as follows:

ghci> :load "Numbers2Words.lhs"
[1 of 1] Compiling Main ( Numbers2Words.lhs, interpreted )
Ok, modules loaded: Main.
ghci>

We will explain what modules are in the next chapter. Now you can type, for example,

ghci> convert 301123
"three hundred and one thousand one hundred and twenty-three"
ghci>

We end the chapter with some exercises. These contain additional points of interest
What is functional programming?

and should be regarded as an integral part of the text. The same is true for all subsequent chapters, so please read the questions even if you do not answer them. The answers are given afterwards.

1.6 Exercises

Exercise A

Consider the function

```haskell
double :: Integer -> Integer
double x = 2*x
```

that doubles an integer. What are the values of the following expressions?

```haskell
map double [1,4,4,3]
map (double . double) [1,4,4,3]
map double []
```

Suppose `sum :: [Integer] -> Integer` is a function that sums a list of integers. Which of the following assertions are true and why?

```haskell
sum . map double = double . sum
sum . map sum = sum . concat
sum . sort = sum
```

You will need to recall what the function `concat` does. The function `sort` sorts a list of numbers into ascending order.

Exercise B

In Haskell, functional application takes precedence over every other operator, so `double 3+4` means `(double 3)+4`, not `double (3+4)`. Which of the following expressions is a rendering of \( \sin^2 \theta \) into Haskell?

```haskell
sin^2 theta sin theta^2 (sin theta)^2
```

(Exponentiation is denoted by (^).) How would you express \( \sin 2\theta / 2\pi \) as a well-formed Haskell expression?

Exercise C

As we said in the text, a character, i.e. an element of `Char`, is denoted using single quotes, and a string is denoted using double quotes. In particular the string "Hello World!" is just a much shorter way of writing the list
General lists can be written with brackets and commas. (By the way, parentheses are round, brackets are square, and braces are curly.) The expressions 'H' and "H" therefore have different types. What are they? What is the difference between 2001 and "2001"?

The operation ++ concatenates two lists. Simplify

\[
[1,2,3] \text{ ++ } [3,2,1] \\
"Hello" \text{ ++ } " World!"
\]

Exercise D

In the common words example we started off by converting every letter in the text to lowercase, and then we computed the words in the text. An alternative is to do things the other way round, first computing the words and then converting each letter in each word to lowercase. The first method is expressed by \texttt{words . map \text{ toLower}}. Give a similar expression for the second method.

Exercise E

An operator \( \oplus \) is said to be \textit{associative} if \( x \oplus (y \oplus z) = (x \oplus y) \oplus z \). Is numerical addition associative? Is list concatenation associative? Is functional composition associative? Give an example of an operator on numbers that is not associative.

An element \( e \) is said to be an \textit{identity element} of \( \oplus \) if \( x \oplus e = e \oplus x = x \) for all \( x \). What are the identity elements of addition, concatenation and functional composition?

Exercise F

My wife has a book with the title

\textit{EHT CDOORRSSW AAAGMNR ACDIINORTY}.

It contains lists of entries like this:

\texttt{6-letter words}

\texttt{-------------------}

\texttt{...}

\texttt{eginor: ignore, region}

\texttt{eginrr: ringer}

\texttt{eginrs: resign, signer, singer}

\texttt{...}
What is functional programming?

Yes, it is an anagram dictionary. The letters of the anagrams are sorted and the results are stored in dictionary order. Associated with each anagram are the English words with the same letters. Describe how you would go about designing a function

\[
\text{anagrams :: Int} \rightarrow \text{[Word]} \rightarrow \text{String}
\]

so that \(\text{anagrams} \, n\) takes a list of English words in alphabetical order, extracts just the \(n\)-letter words and produces a string that, when displayed, gives a list of the anagram entries for the \(n\)-letter words. You are not expected to be able to define the various functions; just give suitable names and types and describe what each of them is supposed to do.

**Exercise G**

Let’s end with a song:

One man went to mow
Went to mow a meadow
One man and his dog
Went to mow a meadow

Two men went to mow
Went to mow a meadow
Two men, one man and his dog
Went to mow a meadow

Three men went to mow
Went to mow a meadow
Three men, two men, one man and his dog
Went to mow a meadow

Write a Haskell function \(\text{song :: Int} \rightarrow \text{String}\) so that \(\text{song} \, n\) is the song when there are \(n\) men. Assume \(n < 10\).

To print the song, type for example

```
ghci> putStrLn (song 5)
```

The function \(\text{putStrLn}\) will be explained in the following chapter. I suggest starting with

\[
\text{song n} = \begin{cases} 
"\n" & \text{if } n = 0 \\
\text{else } \text{song} \,(n-1) \, + \, "\n" \, + \, \text{verse n}
\end{cases}
\]

\[
\text{verse n} = \text{line1 n} \, + \, \text{line2 n} \, + \, \text{line3 n} \, + \, \text{line4 n}
\]
This defines song recursively.

1.7 Answers

Answer to Exercise A

map double [1,4,4,3] = [2,8,8,6]
map (double . double) [1,4,4,3] = [4,16,16,12]
map double [] = []

You will gather from this that [] denotes the empty list.

All the following equations hold:

\[ \text{sum} \cdot \text{map double} = \text{double} \cdot \text{sum} \]
\[ \text{sum} \cdot \text{map sum} = \text{sum} \cdot \text{concat} \]
\[ \text{sum} \cdot \text{sort} = \text{sum} \]

In fact, each of these three equations are consequences of the three simpler laws:

\[ a \cdot (x + y) = a \cdot x + a \cdot y \]
\[ x + (y + z) = (x + y) + z \]
\[ x + y = y + x \]

Of course, we don’t know yet how to prove that the equations hold. (By the way, to avoid fuss we will often use a typewriter = sign to denote the equality of two Haskell expressions written in typewriter font. But a mathematical = sign is used in equations such as \( \sin 2\theta = 2\sin \theta \cos \theta \).)

Answer to Exercise B

Both \( \sin^2 \theta \) and \( (\sin \theta)^2 \) are okay, but not \( \sin^2 \theta \).

Here is the rendering of \( \sin 2\theta / 2\pi \) in Haskell:

\[ \sin (2*\theta) / (2*\pi) \]

Note that

\[ \sin (2*\theta) / 2 \times \pi = (\sin (2*\theta) / 2) \times \pi \]

which is not what we want. The reason is that operators such as / and \* at the same level of precedence associate to the left in expressions. More on this in the next chapter.
What is functional programming?

Answer to Exercise C

<table>
<thead>
<tr>
<th>Character</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'H'</td>
<td>Char</td>
</tr>
<tr>
<td>&quot;H&quot;</td>
<td>[Char]</td>
</tr>
<tr>
<td>2001</td>
<td>Integer</td>
</tr>
<tr>
<td>&quot;2001&quot;</td>
<td>[Char]</td>
</tr>
</tbody>
</table>

By the way, '\' is used as an escape character, so '\n' is the newline character, and '\t' is the tab character. Also, '\\' is the backslash character, and "\\n" is a list of two characters, a backslash and the letter n. As a consequence, the file path C:\firefox\stuff is written as the Haskell string "C:\firefox\stuff".

\[\begin{align*}
[1,2,3] ++ [3,2,1] &= [1,2,3,3,2,1] \\
"Hello" ++ " World!" &= "Hello World!" \\
[1,2,3] ++ [] &= [1,2,3] \\
"Hello" ++ " " ++"World!" &= "HelloWorld!"
\end{align*}\]

If you got the last two right, you will have appreciated that [] is an empty list of anything, but "" is an empty list of characters.

Answer to Exercise D

The clue is in the phrase ‘converting each letter in each word to lowercase’. Converting each letter in a single word is expressed by \texttt{map toLower}, so the answer is \texttt{map (map toLower) \ . \ words}. That means the following equation holds:

\[\text{words \ . \ map toLower} = \text{map (map toLower) \ . \ words}\]

Answer to Exercise E

Numerical addition, list concatenation and functional composition are all associative. But of course, numerical subtraction isn’t. Nor is exponentiation. The identity element of addition is 0, the identity element of concatenation is the empty list, and the identity element of functional composition is the identity function:

\[\text{id} :: a \rightarrow a\]
\[\text{id} \ x = x\]

Answer to Exercise F

This exercise follows Section 1.3 quite closely. One way of computing the function \texttt{anagrams n} is as follows:

1. Extract the words of length \(n\), using a function

\[\text{getWords} :: \text{Int} \rightarrow \text{[Word]} \rightarrow \text{[Word]}\]
2. Take each word and add a label to it. The label consists of the characters of the word, sorted into alphabetical order. For example, word is turned into the pair ("dorw", "word") This labelling is achieved by the function

\[
\text{addLabel} :: \text{Word} \rightarrow (\text{Label}, \text{Word})
\]

where

\[
\text{type Label} = [\text{Char}]
\]

3. Sort the list of labelled words into alphabetical order of label, using the function

\[
\text{sortLabels} :: [(\text{Label}, \text{Word})] \rightarrow [(\text{Label}, \text{Word})]
\]

4. Replace each group of adjacent labelled words with the same label with a single entry consisting of a pair in which the first component is the common label and the second component is a list of words with that label. This uses a function

\[
\text{groupByLabel} :: [(\text{Label}, \text{Word})] \rightarrow [(\text{Label}, [\text{Word}])]\]

5. Replace each entry by a string using a function

\[
\text{showEntry} :: [(\text{Label}, [\text{Word}])] \rightarrow \text{String}
\]

and concatenate the results.

That gives

\[
\text{anagrams} n = \text{concat} \ . \ \text{map} \ \text{showEntry} \ . \ \text{groupByLabel} \ . \ \text{sortLabels} \ . \ \text{map} \ \text{addLabel} \ . \ \text{getWords} n
\]

\textbf{Answer to Exercise G}

One possible solution:

\[
\text{song} n = \text{if } n==0 \ \text{then ""} \\text{else} \ \text{song (n-1)} ++ "\n" ++ \text{verse n}
\]

\[
\text{verse n} = \text{line1 n} ++ \text{line2 n} ++ \text{line3 n} ++ \text{line4 n}
\]

\[
\text{line1 n} = \text{if } n==1 \ \text{then} "One man went to mow\n" \text{else} \ \text{numbers!!(n-2)} ++ " men went to mow\n"
\]

\[
\text{line2 n} = "Went to mow a meadow\n"
\]

\[
\text{line3 n} = \text{if } n==1 \ \text{then} "One man and his dog\n" \text{else}
\]

\[
\text{line4 n} = "One man went to mow\n"
\]
What is functional programming?

```haskell
numbers!!(n-2) ++ " men, " ++ count (n-2)
++ "one man and his dog\n"
line4 n = "Went to mow a meadow\n\n"

count n = if n==0 then ""
  else
  nums!!(n-1) ++ " men, " ++ count (n-1)

numbers = ["Two", "Three", "Four", "Five", "Six",
  "Seven", "Eight", "Nine"]
nums = ["two", "three", "four", "five", "six",
  "seven", "eight"]
```

Notice that we have omitted to declare the types of the component functions and values in this script. Although Haskell will infer the correct types, it is usually a good idea to put them in for all functions and other values, however simple the types may be. Scripts with explicit type signatures are clearer to read and provide a useful check on the validity of definitions.

1.8 Chapter notes

If you are interested in the origins of Haskell, you should definitely read The History of Haskell, a copy of which is obtainable at

research.microsoft.com/~simonpj/papers/history-of-haskell

One of the abiding strengths of Haskell is that it wasn’t designed to be a closed language, and researchers were encouraged to implement novel programming ideas and techniques by building language extensions or libraries. Consequently, Haskell is a large language and there are numerous books, tutorials and papers devoted to various aspects of the subject, including the recent Parallel and Concurrent Programming in Haskell by Simon Marlow (O’Reilly, 2013). Pointers to much of the material can be found at www.haskell.org. But three books in particular were open on my desk while writing this text. The first is Haskell 98, Languages and Libraries, The Revised Report (Cambridge University Press, 2003), edited by Simon Peyton Jones. This is an indispensable aid in understanding the nitty-gritty of the first standard version of Haskell, called Haskell 98. An online version of the report is available at

www.haskell.org/onlinereport
The present book mostly follows this standard, though it does not cover the whole language by any means.

Since then a new standard, Haskell 2010, has been released; see

haskell.org/onlinereport/haskell2010/

One change is that module names are now hierarchical, so we write Data.List rather than just List for the library of list utilities.

The second two are textbooks: Real World Haskell (O’Reilly, 2009) by Bryan O’Sullivan, John Goerzen and Don Stewart; and Programming in Haskell (Cambridge, 2007) by Graham Hutton. As its name implies, the former deals mostly with highly practical applications, while the latter is another introductory text. Graham Hutton did suggest to me, albeit with a grin, that my book should be called Ivory Tower Haskell.

There is a fascinating history concerning the common words problem. Jon Bentley invited one programmer, Don Knuth, to write a literate WEB program for the problem, and another programmer, Doug McIlroy, to write a literary review of it. The result was published in Bentley’s Programming Pearls column in Communications of the ACM, vol. 29, no. 6 (June 1986).
In Haskell every well-formed expression has, by definition, a well-formed type. Each well-formed expression has, by definition, a value. Given an expression for evaluation,

- GHCi checks that the expression is syntactically correct, that is, it conforms to the rules of syntax laid down by Haskell.
- If it is, GHCi infers a type for the expression, or checks that the type supplied by the programmer is correct.
- Provided the expression is well-typed, GHCi evaluates the expression by reducing it to its simplest possible form to produce a value. Provided the value is printable, GHCi then prints it at the terminal.

In this chapter we continue the study of Haskell by taking a closer look at these processes.

2.1 A session with GHCi

One way of finding out whether or not an expression is well-formed is of course to use GHCi. There is a command :type expr which, provided expr is well-formed, will return its type. Here is a session with GHCi (with some of GHCi’s responses abbreviated):

ghci> 3 +4)
<interactive>:1:5: parse error on input `)'

GHCi is complaining that on line 1 the character ')' at position 5 is unexpected; in other words, the expression is not syntactically correct.
2.1 A session with GHCi

ghci> :type 3+4
3+4 :: Num a => a

GHCi is asserting that the type of 3+4 is a number. More on this below.

ghci> :type if 1==0 then 'a' else "a"
<interactive>:1:23:
Couldn't match expected type `Char' with actual type `[Char]'
In the expression: "a"
In the expression: if 1 == 0 then 'a' else "a"

GHCi expects the types of expr1 and expr2 in a conditional expression

    if test then expr1 else expr2

to be the same. But a character is not a list of characters so the conditional expression, though conforming to the rules of Haskell syntax, is not well-formed.

ghci> sin sin 0.5
<interactive>:1:1:
No instance for (Floating (a0 -> a0))
arising from a use of `sin'
Possible fix: add an instance declaration for
   (Floating (a0 -> a0))
In the expression: sin sin 0.5
In an equation for `it': it = sin sin 0.5

GHCi gives a rather opaque error message, complaining that the expression is not well-formed.

ghci> sin (sin 0.5)
0.4612695550331807

Ah, GHCi is happy with this one.

ghci> :type map
map :: (a -> b) -> [a] -> [b]

GHCi returns the type of the function map.

ghci> map
<interactive>:1:1:
No instance for (Show ((a0 -> b0) -> [a0] -> [b0]))
arising from a use of `print'
Possible fix:
add an instance declaration for
Expressions, types and values

(Show ((a0 -> b0) -> [a0] -> [b0]))

In a stmt of an interactive GHCi command: print it

GHCi is saying that it doesn’t know how to print a function.

ghci> :type 1 `div` 0
1 `div` 0 :: Integral a => a

GHCi is asserting that the type of 1 `div` 0 is an integral number. The expression 1 `div` 0 is therefore well-formed and possesses a value.

ghci> 1 `div` 0
*** Exception: divide by zero

GHCi returns an error message. So what is the value of 1 `div` 0? The answer is that it is a special value, written mathematically as $\perp$ and pronounced ‘bottom’. In fact, Haskell provides a predeclared name for this value, except that it is called undefined, not bottom.

ghci> :type undefined
undefined :: a
ghci> undefined
*** Exception: Prelude.undefined

Haskell is not expected to produce the value $\perp$. It may return with an error message, or remain perpetually silent, computing an infinite loop, until we interrupt the computation. It may even cause GHCi to crash. Oh, yes.

ghci> x*x where x = 3
<interactive>:1:5: parse error on input `where'

ghci> let x = 3 in x*x
9

A where clause does not qualify an expression in Haskell, but the whole of the right-hand side of a definition. Thus the first example is not a well-formed expression. On the other hand, a let expression

let <defs> in <expr>

is well-formed, at least assuming the definitions in <defs> are and the expression <expr> is. Let-expressions appear infrequently in what follows, but occasionally they can be useful.
As we have seen, a script is a collection of names and their definitions. Names for functions and values begin with a lowercase letter, except for data constructors (see later on) which begin with an uppercase letter. Types (e.g. \texttt{Int}), type classes (e.g. \texttt{Num}) and modules (e.g. \texttt{Prelude} or \texttt{Data.Char}) also begin with an uppercase letter.

An operator is a special kind of function name that appears between its (two) arguments, such as the $+$ in $x + y$ or the $++$ in $xs ++ ys$. Operator names begin with a symbol. Any (non-symbolic) function of two arguments can be converted into an operator by enclosing it in back quotes, and any operator can be converted to a prefix name by enclosing it in parentheses. For example,

\begin{verbatim}
3 + 4  is the same as  (+) 3 4
\end{verbatim}

\begin{verbatim}
div 3 4  is the same as  3 `div` 4
\end{verbatim}

Operators have different levels of precedence (binding power). For example,

\begin{verbatim}
3 * 4 + 2  means  (3 * 4) + 2
xs ++ yss !! 3  means  xs ++ (yss !! 3)
\end{verbatim}

If in any doubt, add parentheses to remove possible ambiguity. By the way, we can use any names we like for lists, including \texttt{x}, \texttt{y}, \texttt{goodylist}, and so on. But a simple aid to memory is to use \texttt{x} for things, \texttt{xs} for lists of things, and \texttt{xss} for lists of lists of things. That explains why we wrote \texttt{yss} in the expression \texttt{yss !! 3} in the last line above.

Operators with the same level of precedence normally have an order of association, either to the left or right. For example, the usual arithmetic operators associate to the left:

\begin{verbatim}
3 - 4 - 2  means  (3 - 4) - 2
3 - 4 + 2  means  (3 - 4) + 2
3 / 4 * 5  means  (3 / 4) * 5
\end{verbatim}

Functional application, which has higher precedence than any other operator, also associates to the left:

\begin{verbatim}
eee bah gum  means  (eee bah) gum
eee bah gum*2  means  ((eee bah) gum)*2
\end{verbatim}

Some operators associate to the right:
Expressions, types and values

(a → b) → [a] → [b] means (a → b) → ([a] → [b])

x ^ y ^ z means x ^ (y ^ z)

eee . bah . gum means eee . (bah . gum)

Of course, if an operator, such as functional composition, is associative the order has no effect on meaning (i.e. the value is the same). Again, one can always add parentheses to remove possible ambiguity.

We can declare new operators; for example:

(+++) :: Int → Int → Int

x +++ y = if even x then y else x + y

The conditional expression has low binding power, so the expression above means

if even x then y else (x + y)

not (if even x then y else x) + y. Again, one can always use parentheses to group differently.

If we like we can declare a precedence level and an order of association for (+++), but we won’t spell out how.

Sections and lambda expressions

It is a matter of style, but in the main we prefer to write scripts in which all the little helper functions are named explicitly. Thus if we need a function that adds 1 to a number, or doubles a number, then we might choose to name such functions explicitly:

succ, double :: Integer → Integer

succ n = n+1

double n = 2*n

However, Haskell provides alternative ways of naming these two functions, namely (+1) and (2*). The device is called a section. In a section one of the arguments of an operator is included along with the operator. Thus

(+1) n = n+1

(0<) n = 0<n

(<0) n = n<0

(1/) x = 1/x
Sections are certainly attractive ways of naming simple helper functions and we henceforth accept them onto our list of Good Things to Use in Moderation.

There is one important caveat about sections: although (+1) is the section that adds 1 to a number, (−1) is \textit{not} the section that subtracts 1. Instead (−1) is just the number −1. Haskell uses the minus sign both as the binary operation of subtraction and as a prefix to denote negative numbers.

Now suppose we want a function that doubles a number and then adds 1 to the answer. This function is captured by the composition ( +1 ) . ( *2 ) of two sections. But the result is unsatisfying because it looks a little abstruse; anyone reading it would have to pause for a moment to see what it meant. The alternative seems to be to give the function a name, but what would be a suitable name? Nothing helpful really comes to mind.

The alternative is to use a lambda expression \( \lambda n \rightarrow 2n+1 \). It is called a lambda expression because mathematically the function would be written as \( \lambda n.2n+1 \). Read the expression as ‘that function of \( n \) which returns \( 2n+1 \)’. For example,

\[
\text{ghci> } \text{map (\( \lambda n \rightarrow 2n+1 \)) \{1..5\}}
\]
\[
[3,5,7,9,11]
\]

Once in a while a lambda expression seems the best way to describe some function, but only once in a while and we will take them out of the box only on rare occasions.

2.3 Evaluation

Haskell evaluates an expression by reducing it to its simplest possible form and printing the result. For example, suppose we have defined

\[
\text{sqr :: Integer -> Integer}
\]
\[
\text{sqr x = x*x}
\]

There are basically two ways to reduce the expression \( \text{sqr (3+4)} \) to its simplest possible form, namely 49. Either we can evaluate \( 3+4 \) first, or else apply the definition of \( \text{sqr} \) first:

\[
\begin{align*}
\text{sqr (3+4)} &= \text{sqr 7} \\
&= \text{let } x = 3+4 \text{ in } x*x \\
&= 7*7 \\
&= 49
\end{align*}
\]

\[
\begin{align*}
\text{sqr (3+4)} &= \text{sqr (3+4)} \\
&= \text{let } x = 3+4 \text{ in } x*x \\
&= 7*7 \\
&= 49
\end{align*}
\]
The number of reduction steps is the same in each case, but the order of the reduction steps is slightly different. The method on the left is called *innermost reduction* and also *eager evaluation*; the one on the right is called *outermost reduction* or *lazy evaluation*. With eager evaluation arguments are always evaluated before a function is applied. With lazy evaluation the definition of a function is installed at once and only when they are needed are the arguments to the function evaluated.

Doesn’t seem much of a difference, does it? But consider the following (slightly abbreviated) evaluation sequences concerning the function \( \text{fst} \) that returns the first element of a pair, so \( \text{fst} (x,y) = x \):

\[
\begin{align*}
\text{fst} (\text{sqr} \ 1, \text{sqr} \ 2) &= \text{fst} (\text{sqr} \ 1, \text{sqr} \ 2) \\
= \text{fst} (1*1, \text{sqr} \ 2) &= \text{let} \ p = (\text{sqr} \ 1, \text{sqr} \ 2) \\
= \text{fst} (1, \text{sqr} \ 2) &= \text{in} \ \text{fst} \ p \\
= \text{fst} (1,2*2) &= \text{sqr} \ 1 \\
= \text{fst} (1,4) &= 1*1 \\
= 1 &= 1
\end{align*}
\]

The point here is that under eager evaluation the value \( \text{sqr} \ 2 \) is computed, while under lazy evaluation that value is not needed and is not computed.

Now suppose we add the definitions

\[
\begin{align*}
\text{infinity} :: \text{Integer} \\
\text{infinity} &= 1 + \text{infinity} \\
\text{three} :: \text{Integer} \rightarrow \text{Integer} \\
\text{three} \ x &= 3
\end{align*}
\]

Evaluating \( \text{infinity} \) will cause GHCi to go into a long, silent think trying to compute \( 1 + (1 + (1 + (1 + (1 + \ldots \text{until eventually it runs out of space and returns an error message. The value of infinity is } \bot.\)

Again there are two ways to evaluate \( \text{three} \ \text{infinity} \):

\[
\begin{align*}
\text{three} \ \text{infinity} &= \text{let} \ x = \text{infinity} \ \text{in} \ 3 \\
= \text{three} (1+\text{infinity}) &= 3 \\
= \text{three} (1+(1+\text{infinity})) &= \ldots
\end{align*}
\]

Here eager evaluation gets stuck in a loop trying to evaluate \( \text{infinity} \), while lazy evaluation returns the answer 3 at once. We don’t need to evaluate the argument of \( \text{three} \) in order to return 3.

One more definition, a version of the factorial function:
2.3 Evaluation

```haskell
factorial :: Integer -> Integer
factorial n = fact (n,1)

fact :: (Integer,Integer) -> Integer
fact (x,y) = if x==0 then y else fact (x-1,x*y)
```

This is another example of a recursive definition (the definition of infinity was also recursive, and so was the function song in the previous chapter). Expressions involving recursive functions are evaluated like any other definition.

Here the two evaluation schemes result in the following sequence of reduction steps (we hide the steps involving simplification of the conditional expression to make another point):

```
factorial 3 = fact (3,1) = fact (3-1,3*1) = fact (2,3) = fact (2-1,2*3) = fact (1,6) = fact (1-1,1*6) = fact (0,6) = 6
```

The point to appreciate is that, while the number of reduction steps is basically the same, lazy evaluation requires much more space to achieve the answer. The expression `1*(2*(3*1))` is built up in memory before being evaluated.

The pros and cons of lazy evaluation are briefly as follows. On the plus side, lazy evaluation terminates whenever any reduction order terminates; it never takes more steps than eager evaluation, and sometimes infinitely fewer. On the minus side, it can require a lot more space and it is more difficult to understand the precise order in which things happen.

Haskell uses lazy evaluation. ML (another popular functional language) uses eager evaluation. Exercise D explores why lazy evaluation is a Good Thing. Lazy evaluation is considered further in Chapter 7.

A Haskell function \( f \) is said to be strict if \( f \ undefined = undefined \), and non-strict otherwise. The function `three` is non-strict, while `(+)` is strict in both arguments. Because Haskell uses lazy evaluation we can define non-strict functions. That is why Haskell is referred to as a non-strict functional language.
Haskell has built-in (or primitive) types such as Int, Float and Char. The type Bool of boolean values is defined in the standard prelude:

\[
\text{data Bool = False | True}
\]

This is an example of a \textit{data declaration}. The type Bool is declared to have two data \textit{constructors}, False and True. The type Bool has three values, not two: False, True and undefined :: Bool. Why do we need that last value? Well, consider the function

\[
to :: \text{Bool} \rightarrow \text{Bool}
\]

\[
to \ b = \text{not} \ (\text{to} \ b)
\]

The prelude definition of \textit{not} is

\[
\text{not :: Bool} \rightarrow \text{Bool}
\]

\[
\text{not True} = \text{False}
\]

\[
\text{not False} = \text{True}
\]

The definition of \textit{to} is perfectly well-formed, but evaluating \textit{to True} causes GHCi to go into an infinite loop, so its value is $\perp$ of type \text{Bool}. We will have much more to say about data declarations in future chapters.

Haskell has built-in compound types, such as

\[
[\text{Int}] \quad \text{a list of elements, all of type Int}
\]

\[
(\text{Int}, \text{Char}) \quad \text{a pair consisting of an Int and a Char}
\]

\[
(\text{Int}, \text{Char}, \text{Bool}) \quad \text{a triple}
\]

\[
() \quad \text{an empty tuple}
\]

\[
\text{Int} \rightarrow \text{Int} \quad \text{a function from Int to Int}
\]

The sole inhabitant of the type () is also denoted by (). Actually, there is a second member of (), namely undefined :: (). Now we can appreciate that there is a value $\perp$ for every type.

As we have already said, when defining values or functions it is always a good idea to include the type signature as part of the definition.

Consider next the function \textit{take n} that takes the first \textit{n} elements of a list. This function made its appearance in the previous chapter. For example,

\[
\text{take 3 \ [1,2,3,4,5]} = [1,2,3]
\]

\[
\text{take 3 "category"} = "cat"
\]

\[
\text{take 3 [sin,cos]} = [sin,cos]
\]
What type should we assign to \texttt{take}? It doesn’t matter what the type of the elements of the list is, so \texttt{take} is what is called a \textit{polymorphic} function and we denote its type by

\[
\texttt{take :: Int} \to \, [\texttt{a}] \to \, [\texttt{a}]
\]

The \texttt{a} is a \textit{type variable}. Type variables begin with a lowercase letter. Type variables can be instantiated to any type.

Similarly,

\[
\texttt{(++) :: [a] -> [a] -> [a]}
\]

\[
\texttt{map :: (a -> b) -> [a] -> [b]}
\]

\[
\texttt{(.) :: (b -> c) -> (a -> b) -> (a -> c)}
\]

The last line declares the polymorphic type of functional composition.

Next, what is the type of \texttt{(+)}? Here are some suggestions:

\[
\texttt{(+) :: Int} \to \, \texttt{Int} \to \, \texttt{Int}
\]

\[
\texttt{(+) :: Float} \to \, \texttt{Float} \to \, \texttt{Float}
\]

\[
\texttt{( + ) : : a- > a- > a}
\]

The first two types seem too specific, while the last seems too general: we can’t add two functions or two characters or two booleans, at least not in any obvious way.

The answer is to introduce \textit{type classes}:

\[
\texttt{(+) :: Num a => a} \to \, a \to \, a
\]

This declaration asserts that \texttt{( + )} is of type \texttt{a} \to \texttt{a} \to \texttt{a} for any \textit{number type} \texttt{a}. A type class, such as \texttt{Num}, has a collection of named methods, such as \texttt{( + )}, which can be defined differently for each instance of the type class. Type classes therefore provide for \textit{overloaded} functions, functions with the same name but different definitions. Overloading is another kind of polymorphism.

Numbers are rather complicated, and are explained in more detail in the following chapter, so we illustrate type classes with a simpler type class

\[
\texttt{class Eq a where}
\]

\[
\texttt{==),(/=) :: a} \to \, a \to \, \texttt{Bool}
\]

\[
\texttt{x /= y = not (x == y)}
\]

This introduces the Equality type class, members of which can use one and the same equality test \texttt{(==)} and inequality test \texttt{(=/=)}. There is a \textit{default} definition of \texttt{(=/=)} as part of the class, so we only have to provide a definition of \texttt{(==)}. 
To become a member of the `Eq` club we have to define an instance. For example,

```haskell
instance Eq Bool where
  x == y = if x then y else not y

instance Eq Person where
  x == y = (pin x == pin y)
```

If `pin :: Person -> Pin` then we need `Eq Pin` for the last instance to be correct. Of course, we don’t have to make `Person` a member of the Equality club; we can always define

```haskell
samePerson :: Person -> Person -> Bool
samePerson x y = (pin x == pin y)
```

But we can’t use `(==)` instead of `samePerson` unless we make an instance declaration.

Here are simplified versions of two other type classes, `Ord` and `Show`:

```haskell
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  x < y = not (x >= y)
  x <= y = x == y || x < y
  x >= y = x == y || x > y
  x > y = not (x <= y)

class Show a where
  show :: a -> String
```

The boolean operator `(||)` denotes disjunction: `a || b` is true only if at least one of `a` and `b` is true. We can define this operator by

```haskell
(||) :: Bool -> Bool -> Bool
  a || b = if a then True else b
```

The default definitions of the `Ord` methods are mutually dependent, so one has to provide a specific definition of at least one of them in any instance to break the dependency (unlike `Eq` where only `(=/=)` was given a default definition). The type class `Ord` needs `Eq` as a superclass because it makes use of `(==)` in the default definitions of the four comparison operations.

The type class `Show` is used for displaying results. Haskell cannot display the result of a computation unless the type of the result is a member of `Show`. Let us explain this in a little more detail.
We begin with a mystery:

ghci> "Hello ++"\n++ "young" ++"\n++ "lovers"
"Hello\nyoung\nlovers"

Oh. What we wanted was

Hello
young
lovers

Why didn’t Haskell print that?

The reason is that after evaluating a well-formed expression to produce a value, Haskell applies `show` to the value to produce a string that can be printed at the terminal. Applying `show` to a value `v` produces a string that when printed looks exactly like `v`: Thus,

\[
\begin{align*}
    \text{show } 42 &= "42" \\
    \text{show } 42.3 &= "42.3" \\
    \text{show } 'a' &= "'a'" \\
    \text{show } "hello\n" &= "\"hello\n\n"
\end{align*}
\]

Printing the result involves the use of a Haskell command

\[
\text{putStrLn} :: \text{String} \to \text{IO ()}
\]

The type `IO a` is a special type, the type of input–output computations that when executed have some interaction with the outside world and return a value of type `a`. If the return value is uninteresting, as with `putStrLn`, we use the null-tuple value `()`. 

So, Haskell uniformly applies a show-and-put strategy to print values. Since the greeting above is already a string, we really want to miss out the show step and go straight to the put:

ghci> putStrLn ("Hello ++"\n++ "young" ++"\n++ "lovers")
Hello
young
lovers

Haskell provides many more commands for input–output, for reading and writing to files, for displaying graphics, and so on. Such commands have to be sequenced
correctly, and for this Haskell provides a special notation, called do-notation. Com-
mands are the subject of Chapter 10, and what follows is simply a foretaste of 
things to come.

To see an example, consider the common words problem of the previous chapter. 
There we defined a function

\[
\text{commonWords :: Int} \to \text{String} \to \text{String}
\]
such that \text{commonWords} \ n took a text string and returned a string giving a table of 
the \ n \ most common words in the text. The following program reads the text from 
a file, and writes the output to a file. The type \text{FilePath} \ is another synonym for a 
list of characters:

\[
\text{cwords :: Int} \to \text{FilePath} \to \text{FilePath} \to \text{IO()}
\]
\[
cwords \ n \ \text{infile} \ \text{outfile} \\
= \text{do \ {text} <- \text{readFile} \ \text{infile};} \\
\quad \text{writeFile} \ \text{outfile} \ (\text{commonWords} \ n \ \text{text}); \\
\quad \text{putStrLn} \ \text{"cwords done!"}
\]

Evaluating, for example

\text{ghci} \ cwords \ 100 \ "c:\WarAndPeace" \ "c:\Results"

on a Windows platform will cause the file \text{c:\WarAndPeace} to be read, and the 
results printed to \text{c:\Results}. The program also prints a message to the terminal. 
The two component functions of the definition above have types

\[
\text{readFile :: FilePath} \to \text{IO String}
\]
\[
\text{writeFile :: FilePath} \to \text{String} \to \text{IO ()}
\]

Suppose that we didn’t want to call \text{cwords} from within an interactive session, but 
to use it as a stand-alone program. Here is one way. We need to define a value for 
an identifier \text{main} \ of type \text{IO ()}. Here is such a program:

\[
\text{main} \\
= \text{do \ {putStrLn} \ \text{"Take text from where:"};} \\
\quad \text{infile} <- \text{getLine}; \\
\quad \text{putStrLn} \ \text{"How many words:"}; \\
\quad \text{n} <- \text{getLine}; \\
\quad \text{putStrLn} \ \text{"Put results where:"}; \\
\quad \text{outfile} <- \text{getLine}; \\
\quad \text{text} <- \text{readFile} \ \text{infile}; \\
\quad \text{writeFile} \ \text{outfile} \ (\text{commonWords} \ (\text{read} \ \text{n}) \ \text{text}); \\
\quad \text{putStrLn} \ \text{"cwords done!"} 
\]
2.6 Modules

For an explanation of read see Exercise H. Suppose the common words script is stored in the file `cwords.lhs`. We can compile it with GHC, the Glasgow Haskell Compiler:

$ ghc cwords.lhs

The compiled program will be stored in the file `cwords.exe`. To run the program under Windows, type

$ cwords

and follow the instructions.

2.6 Modules

Suppose we thought that the function `commonWords` was sufficiently useful that we wanted to incorporate it into other scripts. The way to do this is to turn the common words script into a module. First, we rewrite the script in the following way:

```haskell
module CommonWords (commonWords) where
  import Data.Char (toLower)
  import Data.List (sort, words)
  ...
  commonWords :: Int -> String -> String
  ...
```

The module declaration is followed by the name of the module, which must begin with a capital letter. Furthermore, the script has to be stored in a file called `CommonWords.lhs` to enable Haskell to find the module (at least, if you are using literate scripts; otherwise it would be `CommonWords.hs`). Following the name of the module is a list of exports, the functions, types and other values you want to be able to export to other scripts. The list of exports has to be enclosed in parentheses. Here we just export one function, `commonWords`. The exports are the only things defined in the module that are visible in other modules. Omitting the export list, and the surrounding parentheses, means that everything in the module is exported.

We can then compile the module using GHC and then import it into other scripts with the declaration

```haskell
import CommonWords (commonWords)
```

There are two major advantages of Haskell modules. One is we can structure our scripts into bite-sized chunks, separating out little groups of related functions into
Expressions, types and values

separate modules. The other advantage is that the functions in a compiled module are much faster to evaluate because their definitions are compiled into machine-specific code, leading to a much slicker reduction process. GHCi is an interpreter rather than a compiler; it evaluates internal forms of expression that are much closer to the source language of Haskell.

2.7 Haskell layout

The examples of do-notation used braces ({ and }) and semicolons; these are examples of explicit layout. Braces and semicolons are used only to control layout and have no meaning as part of the language of Haskell expressions. We can use them in other places too:

```haskell
roots :: (Float,Float,Float) -> (Float,Float)
roots (a,b,c) |
  | a == 0 = error "not quadratic"
  | disc < 0 = error "complex roots"
  | otherwise = ((-b-r)/e, (-b+r)/e)
where {disc = b*b - 4*a*c; r = sqrt d; e = 2*a}
```

Here the where clause uses explicit braces and semicolons rather than appealing to Haskell’s layout rules. Instead, we could have written

```haskell
where disc = b*b - 4*a*c
      r    = sqrt d
      e    = 2*a
```

But we couldn’t have written

```haskell
where disc = b*b - 4*a*c
      r = sqrt d
      e = 2*a
```

The layout (or offside) rule takes effect whenever the opening brace is omitted after the keyword where or do (and also after let). When this happens the indentation of the next item, whether or not on a new line, is remembered. For each subsequent line, if it is indented more, then the previous line is continued; if it is indented the same amount, then a new item begins; and if it is indented less, then the layout list is ended. At least, that’s roughly the offside rule.

The offside rule explains why there is an indentation in the declarations of type classes and instances:
2.8 Exercises

class Foo a where
    I am part of the class declaration.
    So am I.
    Now the class declaration has ended.

You can always put in braces and semicolons if in any doubt. Actually the offside rule can still cause confusion when used with do-notation. So the recommendation is belts, braces and semicolons.

And you thought the football offside rule was complicated.

2.8 Exercises

Exercise A

On the subject of precedence, this question comes from Chris Maslanka’s puzzle page in the *Guardian* newspaper:

‘Is a half of two plus two equal to two or three?’

Exercise B

Some of the following expressions are not syntactically correct, while others are syntactically correct but do not have sensible types. Some are well-formed. Which is which? In the case of a well-formed expression, give a suitable type. Assume double :: Int -> Int. I suggest you don’t use a computer to check your answers, but if you do, be prepared for some strange error messages.

The expressions are:

- [0,1)
- double -3
- double (-3)
- double double 0
- if 1==0 then 2==1
- "++" == "+" ++ "+"
- [(+),(-)]
- [[],[],[[]]]
- concat ["tea","for","2"]
- concat ["tea","for","2"]
Exercise C

In the good old days, one could write papers with titles such as

‘The morphology of prex – an essay in meta-algorithmics’

These days, journals seem to want all words capitalised:

‘The Morphology Of Prex – An Essay In Meta-algorithmics’

Write a function modernise :: String -> String which ensures that paper titles are capitalised as above. Here are some helpful questions to answer first:

1. The function toLower :: Char -> Char converts a letter to lowercase. What do you think is the name of the prelude function that converts a letter to uppercase?

2. The function words :: String -> [Word] was used in the previous chapter. What do you think the prelude function unwords :: [Word] -> String does? Hint: which, if either, of the following equations should hold?

   words . unwords = id
   unwords . words = id

3. The function head :: [a] -> a returns the head of a nonempty list, and tail :: [a] -> [a] returns the list that remains when the head is removed. Suppose a list has head x and tail xs. How would you reconstruct the list?

Exercise D

Beaver is an eager evaluator, while Susan is a lazy one. How many times would Beaver evaluate \( f \) in computing head (map \( f \) \( xs \)) when \( xs \) is a list of length \( n \)?

How many times would Susan? What alternative to head . map \( f \) would Beaver prefer?

The function filter \( p \) filters a list, retaining only those elements that satisfy the boolean test \( p \). The type of filter is

\[
\text{filter} :: (a \to \text{Bool}) \to [a] \to [a]
\]

Susan would happily use head . filter \( p \) for a function that finds the first element of a list satisfying \( p \). Why would Beaver not use the same expression?

Instead, Beaver would probably define something like

1 If you don’t know, google ‘lazy susan’ to discover what a lazy susan is.
first :: (a -> Bool) -> [a] -> a
first p xs | null xs  = error "Empty list"
  | p x  = ...
  | otherwise = ...
where x = head xs

The function `null` returns `True` on an empty list, and `False` otherwise. When evaluated, the expression `error message` stops execution and prints the string `message` at the terminal, so its value is `⊥`. Complete the right-hand side of Beaver’s definition.

What alternative might Beaver prefer to `head . filter p . map f`?

**Exercise E**

The type `Maybe` is declared in the standard prelude as follows:

```haskell
data Maybe a = Nothing | Just a
  deriving (Eq, Ord)
```

This declaration uses a `deriving` clause. Haskell can automatically generate instances of some standard type classes for some data declarations. In the present case the `deriving` clause means that we don’t have to go through the tedium of writing

```haskell
instance (Eq a) => Eq (Maybe a)
  Nothing == Nothing = True
  Nothing == Just y  = False
  Just x == Nothing  = False
  Just x == Just y   = (x == y)

instance (Ord a) => Ord (Maybe a)
  Nothing <= Nothing = True
  Nothing <= Just y  = True
  Just x <= Nothing  = False
  Just x <= Just y   = (x <= y)
```

The reason why `Nothing` is declared to be less than `Just y` is simply because the constructor `Nothing` comes before the constructor `Just` in the data declaration for `Maybe`.

The reason why the `Maybe` type is useful is that it provides a systematic way of handling failure. Consider again the function
Expressions, types and values

first p = head . filter p

of the previous exercise. Both Eager Beaver and Lazy Susan produced versions of this function that stopped execution and returned an error message when first \( p \) was applied to the empty list. That’s not very satisfactory. Much better is to define

\[
\text{first :: (a -> Bool) -> [a] -> Maybe a}
\]

Now failure is handled gracefully by returning Nothing if there is no element of the list that satisfies the test.

Give a suitable definition of this version of first.

Finally, count the number of functions with type \( \text{Maybe a -> Maybe a} \).

\textbf{Exercise F}

Here is a function for computing \( x \) to the power \( n \), where \( n \geq 0 \):

\[
\text{exp :: Integer -> Integer -> Integer}
\]

\[
\text{exp x n | if } n == 0 \text{ then 1 else if } n == 1 \text{ then } x \text{ else } x*exp x (n-1)
\]

How many multiplications does it take to evaluate \( \text{exp } x \ n \)?

Dick, a clever programmer, claims he can compute \( \text{exp } x \ n \) with far fewer multiplications:

\[
\text{exp x n | if } n == 0 \text{ then 1 else if even n then ... else odd n = ...}
\]

Fill in the dots and say how many multiplications it takes to evaluate the expression \( \text{exp } x \ n \) by Dick’s method, assuming \( 2^p \leq n < 2^{p+1} \).

\textbf{Exercise G}

Suppose a date is represented by three integers \((\text{day, month, year})\). Define a function \( \text{showDate :: Date -> String} \) so that, for example,

\[
\text{showDate (10,12,2013) = "10th December, 2013"}
\]

\[
\text{showDate (21,11,2020) = "21st November, 2020"}
\]

You need to know that \texttt{Int} is a member of the type class \texttt{Show}, so that \texttt{show n} produces a string that is the decimal representation of the integer \( n \).
Exercise H

The credit card company Foxy issues cards with ten-digit card-identification numbers (CINs). The first eight digits are arbitrary but the number formed from the last two digits is a checksum equal to the sum of the first eight digits. For example, “6324513428” is a valid CIN because the sum of the first eight digits is 28.

Construct a function \( \text{addSum} :: \text{CIN} \rightarrow \text{CIN} \) that takes a string consisting of eight digits and returns a string of ten digits that includes the checksum. Thus \text{CIN} is a type synonym for \text{String}, though restricted to strings of digits. (Note that Haskell type synonyms cannot enforce type constraints such as this.) You will need to convert between a digit character and the corresponding number. One direction is easy: just use \text{show}. The other direction is also fairly easy:

\[
\begin{align*}
\text{getDigit} & : \text{Char} \rightarrow \text{Int} \\
\text{getDigit} \ c & = \text{read} \ [c]
\end{align*}
\]

The function \text{read} is a method of the type class \text{Read} and has type

\[
\text{read} :: \text{Read a} \Rightarrow \text{String} \rightarrow a
\]

The type class \text{Read} is dual to \text{Show} and \text{read} is dual to \text{show}. For example,

\[
\begin{align*}
\text{ghci}\>&\text{> read "123" :: Int} \\
&123 \\
\text{ghci}\>&\text{> read "123" :: Float} \\
&123.0
\end{align*}
\]

The function \text{read} has to be supplied with the type of the result. One can always add \textit{type annotations} to expressions in this way.

Now construct a function \( \text{valid} :: \text{CIN} \rightarrow \text{Bool} \) that checks whether an identification number is valid. The function \text{take} might prove useful.

Exercise I

By definition a \textit{palindrome} is a string that, ignoring punctuation symbols, blank characters and whether or not a letter is in lowercase or uppercase, reads the same forwards and backwards. Write an interactive program

\[
\text{palindrome} :: \text{IO ()}
\]

which, when run, conducts an interactive session, such as

\[
\text{ghci}\>&\text{> palindrome} \\
\text{Enter a string:}
\]
Madam, I'm Adam
Yes!

ghci> palindrome
Enter a string:
A Man, a plan, a canal – Suez!
No!

ghci> palindrome
Enter a string:
Doc, note I dissent. A fast never prevents a fatness.
I diet on cod.
Yes!

The function isAlpha :: Char -> Bool tests whether a character is a letter, and reverse :: [a] -> [a] reverses a list. The function reverse is provided in the standard prelude and isAlpha can be imported from the library Data.Char.

2.9 Answers

Answer to Exercise A

The answer to Maslanka’s puzzle is ‘Yes!’ This little puzzle has fooled a number of distinguished computer scientists.

Answer to Exercise B

My GHCi session produced (with explanations added):

ghci> :type [0,1)
<interactive>:1:5: parse error on input `)'

GHCi knows that ')' is wrong, though it is not smart enough to suggest ']' .

ghci> :type double -3
<interactive>:1:9:
No instance for (Num (Int -> Int))
arising from the literal `3'
Possible fix: add an instance declaration for
   (Num (Int -> Int))
In the second argument of `(-)’, namely `3’
In the expression: double - 3

The explanation of the error message is that numerical subtraction ( - ) has type
Num a => a -> a. For double - 3 to be well-formed (yes, it was typed as
double -3 but the spaces are not significant here), double has to be a number, so
the class instance Num (Int -> Int) is required. But there isn’t one: you cannot
sensibly subtract a number from a function.

ghci> double (-3)
-6
ghci> double double 0
<interactive>:1:1:
The function `double' is applied to two arguments,
but its type `Int -> Int' has only one
In the expression: double double 0
In an equation for `it': it = double double 0

Most of GHCi’s error message is clear.

ghci> if 1==0 then 2==1

<interactive>:1:18:
pars e error (possibly incorrect indentation)
Conditional expressions are incomplete without an ‘else’ clause.

ghci> "++" == "+" ++ "+"
True

Both sides are well-formed and denote the same list.

ghci> [(+),(-)]
<interactive>:1:1:
No instance for (Show (a0 -> a0 -> a0))
arising from a use of `print'
Possible fix:
add an instance declaration for
  (Show (a0 -> a0 -> a0))
In a stmt of an interactive GHCi command: print it

To display the value [(+),(-)] we have to be able to show its elements. But no
way of showing functions has been provided.

ghci> :type [[]],[[]],[[[[]]]]
Expressions, types and values

\[[],[],[[]]] :: [[[a]]]

To explain, let the main list have type \[b\]. The first element is a list, so \(b = [c]\). The second element is a list of lists, so \(c = [d]\). The third element is a list of lists of lists, so \(d = [a]\).

ghci> concat ["tea","for","2"]
<interactive>:1:21:
Couldn't match expected type `[Char]' with actual type `Char'
In the expression: '2'
In the first argument of `concat',
namely `"tea", "for", '2']'
In the expression: concat ["tea", "for", '2']

The first two elements of the list have type `[Char]`, but the last has type Char and that is not allowed.

ghci> concat ["tea","for","2"]
"teafor2"

**Answer to Exercise C**

1. `toUpperCase`, of course.

2. Concatenates the words, putting a single space between them. We have

```haskell
words . unwords = id
```

but not `unwords . words = id`.

3. \([x] ++ xs\).

```haskell
modernise :: String -> String
modernise = unwords . map capitalise . words

capitalise :: Word -> Word
capitalise xs = [toUpper (head xs)] ++ tail xs
```

We will see another way of writing `capitalise` in Chapter 4.

**Answer to Exercise D**

Computing `head (map f xs)` takes \(n\) evaluations of \(f\) under eager evaluation, but only one under lazy evaluation. Beaver would have to exploit the identity

```haskell
head . map f = f . head.
```
Instead of defining \( \text{first } p = \text{head} \cdot \text{filter } p \), Beaver might define

\[
\text{first} :: (a -> \text{Bool}) -> [a] -> a \\
\text{first } p \, \text{xs} \mid \text{null } \text{xs} = \text{error } "\text{Empty list}" \mid \text{p } x = x \mid \text{otherwise } = \text{first } p \, (\text{tail } \text{xs}) \\
\text{where } x = \text{head } \text{xs}
\]

Instead of defining \( \text{first } p \, f = \text{head} \cdot \text{filter } p \cdot \text{map } f \), Beaver might define

\[
\text{first} :: (b -> \text{Bool}) -> (a -> b) -> [a] -> b \\
\text{first } p \, f \, \text{xs} \mid \text{null } \text{xs} = \text{error } "\text{Empty list}" \mid \text{p } x = x \mid \text{otherwise } = \text{first } p \, f \, (\text{tail } \text{xs}) \\
\text{where } x = f \, (\text{head } \text{xs})
\]

The point is that with eager evaluation most functions have to be defined using explicit recursion, not in terms of useful component functions like map and filter.

**Answer to Exercise E**

Lazy Susan would probably write

\[
\text{first } p \, \text{xs} = \text{if } \text{null } \text{ys} \text{ then Nothing } \text{ else Just } (\text{head } \text{ys}) \\
\text{where } y = \text{filter } p \, \text{xs}
\]

As to the number of functions of type \( \text{Maybe } a -> \text{Maybe } a \), there are just six. Applied to Nothing the function can only return Nothing or undefined. Applied to Just \( x \) the function can only return Nothing or Just \( x \) or undefined. The point is that we know absolutely nothing about the underlying type, so no new values can be invented. That makes six possible functions in all.

**Answer to Exercise F**

It takes \( n-1 \) multiplications to evaluate \( \text{exp } x \, n \). Dick’s method is to exploit the identities \( x^{2m} = (x^2)^m \) and \( x^{2m+1} = x(x^2)^m \) to obtain a recursive definition:

\[
\text{exp } x \, n \mid n == 0 = 1 \mid n == 1 = x \mid \text{even } n = \text{exp } (x \times x) \, m \mid \text{odd } n = x \times \text{exp } (x \times x) \, (n-1) \\
\text{where } m = n \div 2
\]
Expressions, types and values

This is an example of a divide and conquer algorithm. Dick’s program takes \( p \) multiplications, where \( 2^p \leq n < 2^{p+1} \). Thus \( p = \lfloor \log n \rfloor \), where \( \lfloor x \rfloor \) returns the floor of a number, the greatest integer no bigger than the number. We will consider the floor function in more detail in the following chapter.

Answer to Exercise G

\[
\text{showDate :: Date} \rightarrow \text{String}\\
\text{showDate (d,m,y) = show d ++ suffix d ++ " " ++ months !! (m-1) ++ ", " ++ show y}
\]

The function suffix computes the right suffix:

\[
suffix d = \text{if } d==1 \, || \, d==21 \, || \, d==31 \, \text{then } "st" \, \text{else }\\
\quad \text{if } d==2 \, || \, d==22 \, \text{then } "nd" \, \text{else }\\
\quad \text{if } d==3 \, || \, d==23 \, \text{then } "rd" \, \text{else }\\
\quad "th"
\]

\[
\text{months} = \text{["January",\ldots\ldots]}
\]

If you indulged in clever arithmetic to compute suffix, then you should realise that Sometimes a Simple Solution is Best.

Answer to Exercise H

One solution is as follows:

\[
\text{addSum :: CIN} \rightarrow \text{CIN}\\
\text{addSum cin = }\\
\quad \text{cin} ++ \text{show (n `div` 10)} ++ \text{show (n `mod` 10)}\\
\quad \text{where } n = \text{sum (map fromDigit cin)}
\]

\[
\text{valid :: CIN} \rightarrow \text{Bool}\\
\text{valid cin = cin == addSum (take 8 cin)}
\]

\[
\text{fromDigit :: Char} \rightarrow \text{Int}\\
\text{fromDigit c = read [c]}
\]

The function fromDigit will return a numerical digit given a digit character.

Answer to Exercise I

Here is one solution:
import Data.Char (toLower, isAlpha)

palindrome :: IO()
palindrome
  = do {putStrLn "Enter a string:";
        xs <- getLine;
        if isPalindrome xs then putStrLn "Yes!"
                      else putStrLn "No!"}

isPalindrome :: String -> Bool
isPalindrome xs = (ys == reverse ys)
  where ys = map toLower (filter isAlpha xs)

2.10 Chapter notes

The chapter has referred a number of times to the Haskell ‘standard prelude’. This
is a collection of basic types, type classes, functions and other values that are indis-
pensible in many programming tasks. For a complete description of the standard
prelude, see Chapter 8 of the Haskell report; alternatively, visit

www.haskell.org/onlinereport/standard-prelude.html

See www.haskell.org for more information on the implementation of functional
languages and of Haskell in particular. An older book, The Implementation of Func-
tional Programming Languages (Prentice Hall, 1987) by Simon Peyton Jones, is
no longer in print, but an online version can be found at


Apart from GHC there are other maintained compilers for Haskell, including UHC,
the Utrecht Haskell Compiler. See the home page cs.uu.nl/wiki/UHC.

On the eager-versus-lazy evaluation debate, read Bob Harper’s blog article The
point of laziness, which can be found at

existentialtype.wordpress.com/2011/04/24/

In the blog Harper enumerates some of the reasons why he prefers a strict lan-
guage. But also read Lennart Augustsson’s reply to the post. Augustsson’s main
point, emphasised in Exercise D, is that under strict evaluation you are forced for
efficiency reasons to define most functions by explicit recursion, and therefore lose
the ability to build definitions out of simple standard functions. That undercuts our
ability to reason about functions by applying general laws about their component functions.

Bob Harper is one of the authors of *The Definition of Standard ML (Revised)* (MIT Press, 1989). ML is a strict functional language. You can find an introduction to ML at


Another increasingly popular language is Agda, which is both a dependently-typed functional language and also a proof assistant; see the Agda home page

wiki.portal.chalmers.se/agda/pmwiki.php

Chris Maslanka writes a regular column in the Saturday edition of the *Guardian* newspaper.
Numbers in Haskell are complicated because in the Haskell world there are many different kinds of number, including:

- **Int**: limited-precision integers in at least the range $[−2^{29}, 2^{29})$. Integer overflow is not detected.
- **Integer**: arbitrary-precision integers
- **Rational**: arbitrary-precision rational numbers
- **Float**: single-precision floating-point numbers
- **Double**: double-precision floating-point numbers
- **Complex**: complex numbers (defined in Data.Complex)

Most programs make use of numbers in one way or another, so we have to get at least a working idea of what Haskell offers us and how to convert between the different kinds. That is what the present chapter is about.

### 3.1 The type class Num

In Haskell all numbers are instances of the type class `Num`:

```haskell
class (Eq a, Show a) => Num a where
  (+),(-),(*) :: a -> a -> a
  negate     :: a -> a
  abs, signum :: a -> a
  fromInteger :: Integer -> a
```

The class `Num` is a subclass of both `Eq` and `Show`. That means every number can be printed and any two numbers can be compared for equality. Any number can be added to, subtracted from or multiplied by another number. Any number can be
negated. Haskell allows $-x$ to denote negate $x$; this is the only prefix operator in Haskell.

The functions abs and signum return the absolute value of a number and its sign. If ordering operations were allowed in Num (and they aren’t because, for example, complex numbers cannot be ordered), we could define

\[
\begin{align*}
\text{abs } x &= \text{if } x < 0 \text{ then } -x \text{ else } x \\
\text{signum } x &| x < 0 = -1 \\
&| x == 0 = 0 \\
&| x > 0 = 1
\end{align*}
\]

The function fromInteger is a conversion function. An integer literal such as 42 represents the application of fromInteger to the appropriate value of type Integer, so such literals have type Num a => a. This choice is explained further below after we have considered some other classes of number and the conversion functions between them.

### 3.2 Other numeric type classes

The Num class has two subclasses, the real numbers and the fractional numbers:

```haskell
class (Num a, Ord a) => Real a where
toRational :: a -> Rational

class (Num a) => Fractional a where
(/) :: a -> a -> a
fromRational :: Rational -> a
```

Real numbers can be ordered. The only new method in the class Real, apart from the comparison operations which are inherited from the superclass Ord, is a conversion function from elements in the class to elements of Rational. The type Rational is essentially a synonym for pairs of integers. The real number $\pi$ is not rational, so toRational can only convert to an approximate rational number:

```haskell
ghci> toRational pi
884279719003555 % 281474976710656
```

Not quite as memorable as $22 \% 7$, but more accurate. The symbol % is used to separate the numerator and denominator of a rational number.
The fractional numbers are those on which division is defined. A complex number cannot be real but it can be fractional. A floating-point literal such as \(3.149\) represents the application of \(\text{fromRational}\) to an appropriate rational number. Thus

\[
3.149 :: \text{Fractional } a \Rightarrow a
\]

This type and the earlier type \(\text{Num } a \Rightarrow a\) for 42 explains why we can form a legitimate expression such as \(42 + 3.149\), adding an integer to a floating-point number. Both types are members of the \(\text{Num}\) class and all numbers can be added. Consideration of

\[
\text{ghci} > \text{:type } 42 + 3.149
\]

\[
42 + 3.149 :: \text{Fractional } a \Rightarrow a
\]

shows that the result of the addition is also a fractional number.

One of the subclasses of the real numbers is the integral numbers. A simplified version of this class is:

\[
\text{class (Real } a, \text{Enum } a) \Rightarrow \text{Integral } a \text{ where}
\]

\[
\text{divMod} :: a \rightarrow a \rightarrow (a,a)
\]

\[
\text{toInteger} :: a \rightarrow \text{Integer}
\]

The class \(\text{Integral}\) is a subclass of \(\text{Enum}\), those types whose elements can be enumerated in sequence. Every integral number can be converted into an \(\text{Integer}\) through the conversion function \(\text{toInteger}\). That means we can convert an integral number into any other type of number in two steps:

\[
\text{fromIntegral} :: (\text{Integral } a, \text{Num } b) \Rightarrow a \rightarrow b
\]

\[
\text{fromIntegral} = \text{fromInteger} \cdot \text{toInteger}
\]

Application of \(\text{divMod}\) returns two values:

\[
x \div y = \text{fst} (x \divMod y)
\]

\[
x \mod y = \text{snd} (x \divMod y)
\]

The standard prelude functions \(\text{fst}\) and \(\text{snd}\) return the first and second components of a pair:

\[
\text{fst} :: (a,b) \Rightarrow a
\]

\[
\text{fst} (x,y) = x
\]

\[
\text{snd} :: (a,b) \Rightarrow b
\]

\[
\text{snd} (x,y) = y
\]
Mathematically, $x \div y = [x/y]$. We will see how to compute $[x]$ in the following section. And $x \mod y$ is defined by

$$x = (x \div y) \times y + x \mod y$$

For positive $x$ and $y$ we have $0 \leq x \mod y < x$.

Recall the function $\text{digits2}$ from the first chapter, where we defined

$$\text{digits2} n = (n \ `\div` 10, n \ `\mod` 10)$$

It is more efficient to say $\text{digits2} n = n \ `\divMod` 10$ because then only one invocation of $\divMod$ is required. Even more briefly, we can use a section and write $\text{digits2} = (\ `\divMod` 10)$.

There are also other numeric classes, including the subclass $\text{Floating}$ of the class $\text{Fractional}$ that contains, among others, the logarithmic and trigonometric functions. But enough is enough.

### 3.3 Computing floors

The value $[x]$, the floor of $x$, is defined to be the largest integer $m$ such that $m \leq x$. We define a function $\text{floor} :: \text{Float} \to \text{Integer}$ for computing floors. Haskell provides such a function in the standard prelude, but it is instructive to consider our own version.

One student, call him Clever Dick, to whom this task was given came up with the following solution:

```
floor :: Float -> Integer
floor = read . takeWhile (/= '.') . show
```

In words, the number is shown as a string, the string is truncated by taking only the digits up to the decimal point, and the result is read again as an integer. We haven’t met `takeWhile` yet, though Clever Dick evidently had. Clever Dick’s solution is wrong on a number of counts, and Exercise D asks you to list them.

Instead we will find the floor of a number with the help of an explicit search, and for that we will need a loop:

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f x = if p x then x else until p f (f x)
```

The function `until` is also provided in the standard prelude. Here is an example:
3.3 Computing floors

ghci>  until (>100) (*7) 1
343

Essentially until \( f \ p \ x \) computes the first element \( y \) in the infinite list

\[
[x, f \ x, f (f \ x), f (f (f \ x)), \ldots]
\]

for which \( p \ y = \text{True} \). See the following chapter where this interpretation of\( \text{until} \) is made precise.

Thinking now about the design of floor it is tempting to start off with a case analysis, distinguishing between the cases \( x < 0 \) and \( x \geq 0 \). In the case \( x < 0 \) we have to find the first number \( m \) in the sequence \(-1, -2, \ldots\) for which \( m \leq x \). That leads to – in the case of a negative argument –

```haskell
floor x = until (`leq` x) (subtract 1) (-1)
  where m `leq` x = fromInteger m <= x
```

There are a number of instructive points about this definition. Firstly, note the use of the prelude function `subtract` whose definition is

```haskell
subtract x y = y - x
```

We have to use `subtract 1` because \((-1)\) is \textit{not} a section but the number \(-1\) (look at the third argument of `until`).

Secondly, why have we used `\( \text{\_leq}\)` when the alternative `(<=)` seems perfectly adequate? The answer is that `(<=)` has the type

\[
(\leq) :: \text{Num a} \Rightarrow \text{a} \rightarrow \text{a} \rightarrow \text{Bool}
\]

In particular the two arguments of \((\leq)\) have to have the same type. But we want

\[
(\leq) :: \text{Integer} \rightarrow \text{Float} \rightarrow \text{Bool}
\]

and the two arguments have different numeric types. We therefore need to convert integers to floats using `fromInteger`. Appreciation of the need for conversion functions in some situations is one of the key points to understand about Haskell arithmetic.

Finally, note that `(\text{\_leq} \ x)` is not the same as `(\text{leq} \ x)`:  

```haskell
(leq x) y = leq x y
(\text{\_leq} \ x) y = y \text{\_leq} x = leq y x
```

It is easy to make this mistake.

If you don’t like the subsidiary definition, you can always write
floor x = until ((<=x) . fromInteger) (subtract 1) (-1)

In this version we have *inlined* the definition of (`leq` x).

We still have to deal with the case \( x \geq 0 \). In this case we have to look for the first integer \( n \) such that \( x < n+1 \). We can do this by finding the first integer \( n \) such that \( x < n \) and subtracting 1 from the answer. That leads to

\[
\text{floor } x = \text{until } (x \ `lt` \ ) \ (+1) \ 1 - 1
\]

where \( x \ `lt` \ n = x < \text{fromInteger } n \)

Putting the two pieces together, we obtain

\[
\text{floor } x = \begin{cases} 
\text{if } x < 0 \\
\text{then until } (\ `leq` \ x \) \ (\text{subtract } 1) \ (-1) \\
\text{else until } (x \ `lt` \ ) \ (+1) \ 1 - 1
\end{cases}
\]

(Question: why do we not have to write \( x < \text{fromInteger } 0 \) in the first line?)

The real problem with this definition, apart from the general ugliness of a case distinction and the asymmetry of the two cases, is that it is very slow: it takes about \(|x|\) steps (\(|x|\) is the mathematician’s way of writing \(\text{abs } x\)) to deliver the result.

---

**Binary search**

A better method for computing \(\text{floor}\) is to first find integers \(m\) and \(n\) such that \(m \leq x < n\) and then shrink the interval \((m,n)\) to a unit interval (one with \(m+1 = n\)) that contains \(x\). Then the left-hand bound of the interval can be returned as the result. That leads to

\[
\text{floor} :: \text{Float -> Integer}
\]

\[
\text{floor } x = \text{fst (until unit (shrink x) (bound x))}
\]

where \(\text{unit } (m,n) = (m+1 == n)\)

The value \(\text{bound } x\) is some pair \((m,n)\) of integers such that \(m \leq x < n\). If \((m,n)\) is not a unit interval, then \(\text{shrink } x \ (m,n)\) returns a new interval of strictly smaller size that still bounds \(x\).

Let us first consider how to shrink a non-unit interval \((m,n)\) containing \(x\), so \(m \leq x < n\). Suppose \(p\) is any integer that satisfies \(m < p < n\). Such a \(p\) exists since \((m,n)\) is not a unit interval. Then we can define

\[
\text{type Interval } = (\text{Integer, Integer})
\]

\[
\text{shrink} :: \text{Float -> Interval -> Interval}
\]


3.3 Computing floors

\[
\text{shrink } x \ (m, n) = \begin{cases} 
    p \ `\leq` \ x & \text{then } (p, n) \\
    (m, p) & \text{else}
\end{cases}
\]

where \( p = \text{choose } (m, n) \)

How should we define choose?

Two possible choices are \( \text{choose } (m, n) = m+1 \) or \( \text{choose } (m, n) = n-1 \) for both reduce the size of an interval. But a better choice is

\[
\text{choose } :: \text{Interval} \rightarrow \text{Integer} \\
\text{choose } (m, n) = (m+n) \ `\div` 2
\]

With this choice the size of the interval is halved at each step rather than reduced by 1.

However, we need to check that \( m < (m+n) \ `\div` 2 < n \) in the case \( m+1 \neq n \). The reasoning is:

\[
m < (m+n) \ `\div` 2 < n
\]
\[\equiv\ \{\text{ordering on integers}\}\]
\[
m + 1 \leq (m+n) \ `\div` 2 < n
\]
\[\equiv\ \{\text{since } (m+n) \ `\div` 2 = [(m+n)/2]\}\]
\[
m + 1 \leq (m+n)/2 < n
\]
\[\equiv\ \{\text{arithmetic}\}\]
\[
m + 2 \leq n \land m < n
\]
\[\equiv\ \{\text{arithmetic}\}\]
\[
m + 1 < n
\]

Finally, how should we define bound? We can start off by defining

\[
\text{bound } :: \text{Float} \rightarrow \text{Interval} \\
\text{bound } x = (\text{lower } x, \text{upper } x)
\]

The value \( \text{lower } x \) is some integer less than or equal to \( x \), and \( \text{upper } x \) some integer greater than \( x \). Instead of using linear search to discover these values, it is better to use

\[
\text{lower } :: \text{Float} \rightarrow \text{Integer} \\
\text{lower } x = \text{until } (\ `\leq` \ x) \ (*2) \ (-1)
\]

\[
\text{upper } :: \text{Float} \rightarrow \text{Integer} \\
\text{upper } x = \text{until } (x \ `\lt`) \ (*2) \ 1
\]
For a fast version of bound it is better to double at each step rather than increase or decrease by 1. For example, with \( x = 17.3 \) it takes only seven comparisons to compute the surrounding interval \((-1,32)\), which is then reduced to \((17,18)\) in a further five steps. In fact, evaluating both the upper and lower bounds takes time proportional to \(\log |x|\) steps, and the whole algorithm takes at most twice this time. An algorithm that takes logarithmic time is much faster than one that takes linear time.

The standard prelude defines \texttt{floor} in the following way:

\[
\texttt{floor } x = \text{ if } r < 0 \text{ then } n-1 \text{ else } n
\]
where \((n,r) = \texttt{properFraction } x\)

The function \texttt{properFraction} is a method in the \texttt{RealFrac} type class (a class we haven’t discussed and whose methods deal with truncating and rounding numbers). It splits a number \(x\) into its integer part \(n\) and its fractional part \(r\), so \(x = n + r\). Now you know.

3.4 Natural numbers

Haskell does not provide a type for the natural numbers, that is, the nonnegative integers. But we can always define such a type ourselves:

\[
\texttt{data Nat = Zero | Succ Nat}
\]

This is an example of a \texttt{data declaration}. The declaration says that \texttt{Zero} is a value of \texttt{Nat} and that \texttt{Succ n} is also a value of \texttt{Nat} whenever \(n\) is. Both \texttt{Zero} and \texttt{Succ} are called \texttt{data constructors} and begin with a capital letter. The type of \texttt{Zero} is \texttt{Nat} and the type of \texttt{Succ} is \texttt{Nat -> Nat}. Thus each of

\[
\texttt{Zero, Succ Zero, Succ (Succ Zero), Succ (Succ (Succ Zero))}
\]
is an element of \texttt{Nat}.

Let us see how to program the basic arithmetical operations by making \texttt{Nat} a fully paid-up member of the \texttt{Num} class. First, we have to make \texttt{Nat} an instance of \texttt{Eq} and \texttt{Show}:

\[
\texttt{instance Eq Nat where}
\]
\[
\texttt{Zero == Zero \ } = \text{ True}
\texttt{Zero == Succ n} \ = \text{ False}
\texttt{Succ m == Zero} \ = \text{ False}
\texttt{Succ m == Succ n = (m == n)}
\]
instance Show Nat where
  show Zero = "Zero"
  show (Succ Zero) = "Succ Zero"
  show (Succ (Succ n)) = "Succ (" ++ show (Succ n) ++ ")"

These definitions make use of pattern matching. In particular, the definition of show makes use of three patterns, Zero, Succ Zero and Succ (Succ n). These patterns are different from one another and together cover all the elements of Nat apart from ⊥.

Alternatively, we could have declared

    data Nat = Zero | Succ Nat deriving (Eq,Ord,Show)

As we said in Exercise E of the previous chapter, Haskell is smart enough to construct automatically instances of some standard classes, including Eq, Ord and Show.

Now we can install Nat as a numeric type:

instance Num Nat where
  m + Zero = m
  m + Succ n = Succ (m+n)

  m * Zero = Zero
  m * (Succ n) = m * n + m

  abs n = n
  signum Zero = Zero
  signum (Succ n) = Succ Zero

  m - Zero = m
  Zero - Succ n = Zero
  Succ m - Succ n = m - n

  fromInteger x
  | x <= 0 = Zero
  | otherwise = Succ (fromInteger (x-1))

We have defined subtraction as a total operation: \( m - n = 0 \) if \( m \leq n \). Of course, the arithmetic operations on Nat are horribly slow. And each number takes up a lot of space.
**Partial numbers**

We have said that there is a value $\bot$ of every type. Thus undefined :: a for all types a. Since Succ is, by definition, a non-strict function, the values

    undefined, Succ undefined, Succ (Succ undefined), ...

are all different and all members of Nat. To be honest, these partial numbers are not very useful, but they are there. You can think of Succ undefined as being a number about which we know only that it is at least 1:

```
ghci> Zero == Succ undefined
False
ghci> Succ Zero == Succ undefined
*** Exception: Prelude.undefined
```

There is also one further number in Nat:

    infinity :: Nat
    infinity = Succ infinity

Thus

```
ghci> Zero == infinity
False
ghci> Succ Zero == infinity
False
```

and so on.

In summary, the elements of Nat consist of the finite numbers, the partial numbers and the infinite numbers (of which there is only one). We shall see that this is true of other data types: there are the finite elements of the type, the partial elements and the infinite elements.

We could have chosen to make the constructor Succ strict. This is achieved by declaring

    data Nat = Zero | Succ !Nat

The annotation ! is known as *strictness flag*. With such a declaration, we have for example

```
ghci> Zero == Succ undefined
*** Exception: Prelude.undefined
```
This time, evaluating the equality test forces the evaluation of both sides, and the evaluation of \texttt{Succ undefined} raises an error message. Making \texttt{Succ} strict collapses the natural numbers into just the finite numbers and one undefined number.

### 3.5 Exercises

**Exercise A**

Which of the following expressions denote 1?

\[-2 + 3, 3 + -2, 3 + (-2), \text{subtract} \; 2 \; 3, 2 + \text{subtract} \; 3\]

In the standard prelude there is a function \texttt{flip} defined by

\[
\text{flip } f \; x \; y = f \; y \; x
\]

Express \texttt{subtract} using \texttt{flip}.

**Exercise B**

Haskell provides no fewer than three ways to define exponentiation:

\[
\begin{align*}
(\wedge) &:: (\text{Num a}, \text{Integral b}) \Rightarrow a \rightarrow b \rightarrow a \\
(\wedge\wedge) &:: (\text{Fractional a}, \text{Integral b}) \Rightarrow a \rightarrow b \rightarrow a \\
(\star\star) &:: (\text{Floating a}) \Rightarrow a \rightarrow a \rightarrow a
\end{align*}
\]

The operation \texttt{(\wedge)} raises any number to a nonnegative integral power; \texttt{(\wedge\wedge)} raises any number to any integral power (including negative integers); and \texttt{(\star\star)} takes two fractional arguments. The definition of \texttt{(\wedge)} basically follows Dick’s method of the previous chapter (see Exercise E). How would you define \texttt{(\wedge\wedge)}?

**Exercise C**

Could you define \texttt{div} in the following way?

\[
\text{div} :: \text{Integral a} \Rightarrow a \rightarrow a \rightarrow a \\
\text{div} \; x \; y = \text{floor} \; (x/y)
\]

**Exercise D**

Consider again Clever Dick’s solution for computing \texttt{floor}:

\[
\begin{align*}
\text{floor} &:: \text{Float} \rightarrow \text{Integer} \\
\text{floor} &\equiv \text{read} \; . \; \text{(takeWhile (\neq \text{'.')}) \; . \; \text{show}}
\end{align*}
\]
Why doesn’t it work?

Consider the following mini-interaction with GHCi:

```
ghci> 12345678.0 :: Float
1.2345678e7
```

Haskell allows the use of so-called *scientific notation*, also called *exponent notation*, to describe certain floating-point numbers. For example the number above denotes $1.2345678 \times 10^7$. When the number of digits of a floating-point number is sufficiently large, the number is printed in this notation. Now give another reason why Clever Dick’s solution doesn’t work.

**Exercise E**

The function\( \textit{isqrt} : \text{Float} \to \text{Integer} \) returns the floor of the square root of a (nonnegative) number. Following the strategy of Section 3.3, construct an implementation of \( \textit{isqrt} \ x \) that takes time proportional to \( \log x \) steps.

**Exercise F**

Haskell provides a function\( \textit{sqrt} : \text{Floating} \ a \Rightarrow a \to a \) that gives a reasonable approximation to the square root of a (nonnegative) number. But, let’s define our own version. If \( y \) is an approximation to \( \sqrt{x} \), then so is \( x/y \). Moreover, either \( y \leq \sqrt{x} \leq x/y \) or \( x/y \leq \sqrt{x} \leq y \). What is a better approximation to \( \sqrt{x} \) than either \( y \) or \( x/y \)? (Yes, you have just rediscovered Newton’s method for finding square roots.)

The only remaining problem is to decide when an approximation \( y \) is good enough. One possible test is \( |y^2 - x| < \varepsilon \), where \( |x| \) returns the absolute value of \( x \) and \( \varepsilon \) is a suitably small number. This test guarantees an absolute error of at most \( \varepsilon \). Another test is \( |y^2 - x| < \varepsilon \times x \), which guarantees a relative error of at most \( \varepsilon \). Assuming that numbers of type \textit{Float} are accurate only to six significant figures, which of these two is the more sensible test, and what is a sensible value for \( \varepsilon \)?

Hence construct a definition of \( \textit{sqrt} \).

**Exercise G**

Give an explicit instance of \textit{Nat} as a member of the type class \textit{Ord}. Hence construct a definition of

\[
\text{divMod} :: \text{Nat} \to \text{Nat} \to (\text{Nat},\text{Nat})
\]
Answer to Exercise A

All except 2 + -3 and 2 + subtract 3, neither of which are well-formed. We have subtract = flip (-).

Answer to Exercise B

\[ x \ ^n = \begin{cases} \text{if } \ 0 \leq n \ \text{then} \ x \ ^n \ \text{else} \ 1/(x \ ^{\text{negate } n}) \end{cases} \]

Answer to Exercise C

No. You would have to write

\[ \text{div} :: \text{Integral } a \Rightarrow a \rightarrow a \rightarrow a \]
\[ \text{div } x \ y = \text{floor (fromInteger } x / \text{fromInteger } y) \]

Answer to Exercise D

Clever Dick’s function gives \( \text{floor (-3.1)} = -3 \) when the answer should be -4. And if you tried to repair his solution by subtracting 1 if the solution was negative, you would have \( \text{floor (-3.0)} = -4 \) when the answer should be -3. Ugh!

Also, Clever Dick’s solution has \( \text{floor 12345678.0} = 1 \) because the argument is shown as \( 1.2345678e7 \).

Answer to Exercise E

\[ \text{isqrt} :: \text{Float } \rightarrow \text{Integer} \]
\[ \text{isqrt } x = \text{fst (until unit (shrink } x) (\text{bound } x)) \]
\[ \text{where unit } (m,n) = (m+1 == n) \]

\[ \text{shrink} :: \text{Float } \rightarrow \text{Interval } \rightarrow \text{Interval} \]
\[ \text{shrink } x (m,n) = \text{if } (p*p) \ `\text{leq}` \ x \ \text{then} \ (p,n) \ \text{else} \ (m,p) \]
\[ \text{where } p = (m+n) \ `\text{div}` \ 2 \]

\[ \text{bound} :: \text{Float } \rightarrow \text{Interval} \]
\[ \text{bound } x = (0,\text{until above } (*2) \ 1) \]
\[ \text{where } \text{above } n = x \ `\text{lt}` \ (n*n) \]

The functions `\text{leq}` and `\text{lt}` were defined in Section 3.3. Note the parentheses in the expressions \( (p*p) \ `\text{leq}` \ x \) and \( x \ `\text{lt}` \ (n*n) \). We didn’t state an order of association for `\text{leq}` and `\text{lt}`, so without parentheses these two expressions
would have been interpreted as the ill-formed expressions \( p \ast (p \ `\leq` \ x) \) and \((x \ `\lt` \ n) \ast n\). (I made just this mistake when first typing in the solution.)

**Answer to Exercise F**

A better approximation to \( \sqrt{x} \) than either \( y \) or \( \frac{x}{y} \) is \( (y + \frac{x}{y})/2 \). The relative-error test is the more sensible one, and the program is

```haskell
sqrt :: Float -> Float
sqrt x = until goodenough improve x
    where goodenough y = abs (y*y-x) < eps*x
          improve y = (y+x/y)/2
          eps = 0.000001
```

**Answer to Exercise G**

It is sufficient to define \(<\) :

```haskell
instance Ord Nat where
    Zero < Zero   = False
    Zero < Succ n = True
    Succ m < Zero = False
    Succ m < Succ n = (m < n)
```

Now we can define

```haskell
divMod :: Nat -> Nat -> (Nat,Nat)
divMod x y = if x < y then (Zero,x)
             else (Succ q,r)
    where (q,r) = divMod (x-y) y
```

### 3.7 Chapter notes

Chapter 4

Lists

Lists are the workhorses of functional programming. They can be used to fetch and carry data from one function to another; they can be taken apart, rearranged and combined with other lists to make new lists. Lists of numbers can be summed and multiplied; lists of characters can be read and printed; and so on. The list of useful operations on lists is a long one. This chapter describes some of the operations that occur most frequently, though one particularly important class will be introduced only in Chapter 6.

4.1 List notation

As we have seen, the type \([a]\) denotes lists of elements of type \(a\). The empty list is denoted by \([],\). We can have lists over any type but we cannot mix different types in the same list. As examples,

\[
\begin{align*}
[undefined,undefined] & : [a] \\
[sin,cos,tan] & : \text{Floating } a \Rightarrow [a \rightarrow a] \\
[[1,2,3],[4,5]] & : \text{Num } a \Rightarrow [[a]] \\
["tea","for",2] & \text{ not valid}
\end{align*}
\]

List notation, such as \([1,2,3]\), is in fact an abbreviation for a more basic form

\[1:2:3:[]\]

The operator \((::) : : a \rightarrow [a] \rightarrow [a]\), pronounced ‘cons’, is a constructor for lists. It associates to the right so there is no need for parentheses in the above expression. It has no associated definition, which is why it is a constructor. In other words, there are no rules for simplifying an expression such as \(1:2:[]\). The
operator (:) is non-strict in both arguments – more precisely, it is non-strict and returns a non-strict function. The expression

```
undefined : undefined
```

may not be very interesting, but we do know it is not the empty list. In fact, that is the only thing we do know about it. Note that the two occurrences of undefined have different types in this expression.

The empty list [] is also a constructor. Lists can be introduced as a Haskell data type with the declaration

```
data List a = Nil | Cons a (List a)
```

The only difference is that List a is written [a], Nil is written [] and Cons is written (:).

According to this declaration, every list of type [a] takes one of three forms:

- The undefined list `undefined :: [a];`
- The empty list `[] :: [a];`
- A list of the form `x:xs` where `x :: a` and `xs :: [a].`

As a result there are three kinds of list:

- A **finite** list, which is built from (:) and []; for example, `1:2:3:[]`
- A **partial** list, which is built from (:) and undefined; for example, the list `filter (<4) [1..]` is the partial list `1:2:3:undefined`. We know there is no integer after 3 that is less than 4, but Haskell is an evaluator, not a theorem prover, so it ploughs away without success looking for more answers.
- An **infinite** list, which is built from (:) alone; for example, `[1..]` is the infinite list of the nonnegative integers.

All three kinds of list arise in everyday programming. Chapter 9 is devoted to exploring the world of infinite lists and their uses. For example, the prelude function `iterate` returns an infinite list:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x:iterate f (f x)
```

In particular, `iterate (+1) 1` is an infinite list of the positive integers, a value we can also write as `[1..]` (see the following section).

As another example,
4.2 Enumerations

Haskell provides useful notation for enumerating lists of integers. When \( m \) and \( n \) are integers we can write

\[
[m..n] \quad \text{for the list } [m, m+1, \ldots, n] \\
[m..] \quad \text{for the infinite list } [m, m+1, m+2, \ldots] \\
[m,n..p] \quad \text{for the list } [m, m+(n-m), m+2(n-m), \ldots, p] \\
[m,n..] \quad \text{for the infinite list } [m, m+(n-m), m+2(n-m), \ldots]
\]

The first two notations crop up frequently in practice, the second two less so. As examples,

ghci> [0,2..11]
[0,2,4,6,8,10]
ghci> [1,3..]
[1,3,5,7,9,11 {Interrupted}]

In the first example the enumeration stops at 10 because 11 isn’t even. In the second example we quickly interrupted the evaluation of an infinite list.

As a matter of fact, enumerations are not restricted to integers, but to members of yet another type class \texttt{Enum}. We won’t elaborate more on this class, except to say that \texttt{Char} is also a member:

ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
4.3 List comprehensions

Haskell provides another useful and very attractive piece of notation, called *list comprehensions*, for constructing lists out of other lists. We illustrate with a few examples:

```
ghci> [x*x | x <- [1..5]]
[1,4,9,16,25]
ghci> [x*x | x <- [1..5], isPrime x]
[4,9,25]
ghci> [(i,j) | i <- [1..5], even i, j <- [i..5]]
[(2,2),(2,3),(2,4),(2,5),(4,4),(4,5)]
ghci> [x | xs <- [[(3,4)],[5,4),(3,2)], (3,x) <- xs]
[4,2]
```

Here is another example. Suppose we wanted to generate all Pythagorean triads in a given range. These are triples of numbers \((x,y,z)\) such that \(x^2 + y^2 = z^2\) and \(1 \leq x,y,z \leq n\) for some given \(n\). We can define

```
triads :: Int -> [(Int,Int,Int)]
triads n = [(x,y,z) | x <- [1..n], y <- [1..n],
                    z <- [1..n], x*x+y*y==z*z]
```

Hence

```
ghci> triads 15
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),
  (8,6,10),(9,12,15),(12,5,13),(12,9,15)]
```

That’s probably not what we want: each essentially distinct triad is generated in two different ways. Moreover, the list contains redundant triads consisting of multiples of basic triads.

To improve the definition of \(\text{triad}\) we can restrict \(x\) and \(y\) so that \(x < y\) and \(x\) and \(y\) are coprime, meaning they have no divisors in common. As mathematicians we know that \(2x^2\) cannot be the square of an integer, so the first restriction is valid. The divisors of a number can be computed by

```
divisors x = [d | d <- [2..x-1], x `mod` d == 0]
```

Hence

```
coprime x y = disjoint (divisors x) (divisors y)
```

We will leave the definition of \(\text{disjoint}\) as an exercise.
That means we can define

\[
\text{triads } n = \{(x,y,z) \mid x \leftarrow [1..n], \ y \leftarrow [x+1..n], \ \text{coprime } x \ y, \ z \leftarrow [y+1..n], \ x^2 + y^2 = z^2\}
\]

This definition is better than before, but let us try to make it a little faster, mainly to illustrate an important point. Since \(2x^2 < x^2 + y^2 = z^2 \leq n^2\) we see that \(x < n/\sqrt{2}\). So \(x \leq \lfloor n/\sqrt{2} \rfloor\). That suggests we can write

\[
\text{triads } n = \{(x,y,z) \mid x \leftarrow [1..m], \ y \leftarrow [x+1..n], \ \text{coprime } x \ y, \ z \leftarrow [y+1..n], \ x^2 + y^2 = z^2\}
\]

where \(m = \text{floor } (n / \sqrt{2})\). But the expression for \(m\) is incorrect: \(n\) is an \text{Int} and we cannot divide integers. We need an explicit conversion function, and the one to use is \text{fromIntegral} (not \text{fromInteger} because \(n\) is an \text{Int} not an \text{Integer}). We need to replace the definition of \(m\) by \(m = \text{floor } (\text{fromIntegral } n / \sqrt{2})\). Once again we have to be careful about what kinds of number we are dealing with and aware of the available conversion functions between them.

List comprehensions can be used to define some common functions on lists. For example,

\[
\begin{align*}
\text{map } f \ x s &= [f x \mid x \leftarrow x s] \\
\text{filter } p \ x s &= [x \mid x \leftarrow x s, \ p x] \\
\text{concat } x s s &= [x \mid x s \leftarrow x s s, \ x \leftarrow x s]
\end{align*}
\]

Actually, in Haskell it is the other way around: list comprehensions are translated into equivalent definitions in terms of \text{map}, and \text{concat}. The translation rules are:

\[
\begin{align*}
[e \mid \text{True}] &= [e] \\
[e \mid q] &= [e \mid q, \text{True}] \\
[e \mid b, Q] &= \text{if } b \text{ then } [e \mid Q] \text{ else } [] \\
[e \mid p \leftarrow x s, Q] &= \text{let } \text{ok } p = [e \mid Q] \\
&\quad \text{ok } _ = [] \\
&\quad \text{in } \text{concat } (\text{map } \text{ok } x s)
\end{align*}
\]

The definition of \text{ok} in the fourth rule uses a \textit{don't care} pattern, also called a \textit{wild card}. The \(p\) in the fourth rule is a pattern, and the definition of \text{ok} says that the empty list is returned on any argument that doesn’t match the pattern \(p\).

Another useful rule is
Lists

\[ [e \mid Q1, Q2] = \text{concat} \ [ [e \mid Q2] \mid Q1] \]

4.4 Some basic operations

We can define functions over lists by pattern matching. For example,

\[
\text{null} :: [a] \to \text{Bool} \\
\text{null} \ [\] = \text{True} \\
\text{null} \ (x:x:xs) = \text{False}
\]

The patterns \([\] \) and \(x:xs\) are disjoint and exhaustive, so we can write the two equations for \text{null} in either order. The function \text{null} is strict because Haskell has to know which equation to apply and that requires evaluation of the argument, at least to the extent of discovering whether it is the empty list or not. (A question: why not simply define \text{null} = (==[])?) We could also have written

\[
\text{null} \ [\] = \text{True} \\
\text{null} \ _ = \text{False}
\]

This definition uses a don’t care pattern.

Here are two other definitions using pattern matching:

\[
\text{head} :: [a] \to a \\
\text{head} \ (x:x:xs) = x
\]

\[
\text{tail} :: [a] \to [a] \\
\text{tail} \ (x:x:xs) = xs
\]

There is no equation for the pattern \([\] \), so Haskell reports an error if we try to evaluate \text{head} \ [\] or \text{tail} \ [\].

We can use \([x]\) as shorthand for \(x:[]\) in a pattern:

\[
\text{last} :: [a] \to a \\
\text{last} \ [x] = x \\
\text{last} \ (x:y:ys) = \text{last} \ (y:ys)
\]

The first equation has a pattern that matches a singleton list; the second has a pattern that matches a list that contains at least two elements. The standard prelude definition of \text{last} is slightly different:

\[
\text{last} \ [x] = x \\
\text{last} \ (_:xs) = \text{last} \ xs
\]
This definition uses a don’t care pattern. The two equations have to be written in this order because \( x \cdot [] \) matches both patterns.

4.5 Concatenation

Here is the definition of \((++):= [a] \rightarrow [a] \rightarrow [a]\):

\[
\begin{align*}
[] ++ ys &= ys \\
(x:xs) ++ ys &= x:(xs ++ ys)
\end{align*}
\]

The definition uses pattern matching on the first argument but not on the second. The second equation for \((++)\) is very succinct and requires some thought, but once you have got it, you have understood a lot about how lists work in functional programming. Here is a simple evaluation sequence:

\[
\begin{align*}
[1,2] ++ [3,4,5] &= \{\text{notation}\} \\
(1:(2:[])) ++ (3:(4:(5:[]))) &= \{\text{second equation for ++}\} \\
1:(((2:[]) ++ (3:(4:(5:[])))) &= \{\text{and again}\} \\
1:((2:[] ++ (3:(4:(5:[])))) &= \{\text{first equation for ++}\} \\
1:(2:(3:(4:(5:[])))) &= \{\text{notation}\} \\
[1,2,3,4,5]
\end{align*}
\]

As this example suggests, the cost of evaluating \(xs++ys\) is proportional to the length of \(xs\), where

\[
\begin{align*}
\text{length} &:= [a] \rightarrow \text{Int} \\
\text{length} [\mathbf{\_}] &= 0 \\
\text{length} (x:xs) &= 1 + \text{length} xs
\end{align*}
\]

Note also that

\[
\begin{align*}
\text{undefined} ++ [1,2] &= \text{undefined} \\
[1,2] ++ \text{undefined} &= 1:2:\text{undefined}
\end{align*}
\]
Lists

We know nothing about the first list, but we do know that the second list begins with 1 followed by 2.

Concatenation is an associative operation. Thus

\[(xs ++ ys) ++ zs = xs ++ (ys ++ zs)\]

for all lists \(xs\), \(ys\) and \(zs\). We will see how to prove assertions like these in Chapter 6.

4.6 concat, map and filter

Three very useful list operations that we have met already are \texttt{concat}, \texttt{map} and \texttt{filter}. Here are their definitions using pattern matching:

\[
\begin{align*}
\texttt{concat} & : \ [\ [a] ] \to [a] \\
\texttt{concat} [ ] & = [ ] \\
\texttt{concat} (xs:xss) & = xs ++ \texttt{concat} xss \\
\texttt{map} & : (a \to b) \to [a] \to [b] \\
\texttt{map} f [ ] & = [ ] \\
\texttt{map} f (x:xs) & = f \ x : \texttt{map} f \ xs \\
\texttt{filter} & : (a \to \text{Bool}) \to [a] \to [a] \\
\texttt{filter} p [ ] & = [ ] \\
\texttt{filter} p (x:xs) & = \text{if} \ p \ x \ \text{then} \ x : \texttt{filter} p \ xs \\
& \quad \text{else} \ \texttt{filter} p x x
\end{align*}
\]

There is a common theme underlying these definitions that we will identify and exploit in Chapter 6. An alternative definition of \texttt{filter} is

\[
\begin{align*}
\texttt{filter} p & = \texttt{concat} . \texttt{map} (\text{test} \ p) \\
\texttt{test} p x & = \text{if} \ p \ x \ \text{then} \ [x] \ \text{else} \ [ ]
\end{align*}
\]

With this definition, \texttt{filter} \(p\) is implemented by converting each element of the list into a singleton list if it satisfies \(p\), and the empty list otherwise. The results are then concatenated.

Two basic facts about \texttt{map} are that

\[
\begin{align*}
\texttt{map} \ \text{id} & = \text{id} \\
\texttt{map} \ (f \ . \ g) & = \texttt{map} f \ . \texttt{map} g
\end{align*}
\]
The first equation says that applying the identity function to each element of a list leaves the list unchanged. The two occurrence of id in this law have different types: on the left it is \( a \rightarrow a \) and on the right it is \([a] \rightarrow [a]\). The second equation says that applying \( g \) to every element of a list, and then applying \( f \) to every element of the result, gives the same list as applying \( f \circ g \) to every element. Read from right to left, the equation says that two traversals of a list can be replaced by one, with a corresponding gain in efficiency.

The two facts have a name: they are called the functor laws of map. The name is borrowed from a branch of mathematics called Category Theory. In fact, Haskell provides a type class \textbf{Functor}, whose definition is

\[
\text{class Functor } f \text{ where} \\
fmap :: (a \rightarrow b) \rightarrow f a \rightarrow f b
\]

The method \textbf{fmap} is expected to satisfy exactly the same laws as \textbf{map}. The reason for this type class is that the idea of mapping a function over a list can be generalised to one of mapping a function over an arbitrary data structure, such as trees of various kinds. For example, consider the type

\[
data \text{Tree } a = \text{Tip } a \mid \text{Fork } (\text{Tree } a) (\text{Tree } a)
\]

of binary trees with labels in their tips. Tree-structured data arise in a number of places, for example with the syntax of expressions of various kinds. We can define a mapping function over trees, but rather than calling it \textbf{mapTree} we can call it \textbf{fmap} by making trees a member of the \textbf{Functor} class:

\[
\text{instance Functor Tree where} \\
fmap f (\text{Tip } x) = \text{Tip } (f x) \\
fmap f (\text{Fork } u v) = \text{Fork } (fmap f u) (fmap f v)
\]

In fact \textbf{map} is just a synonym for the instance \textbf{fmap} for lists:

\[
\text{ghci}> \text{fmap } (+1) \ [2,3,4] \\
[3,4,5]
\]

We mention the \textbf{Functor} type class here primarily to show that if ever you think some function on lists can be usefully generalised to other kinds of data structure, the chances are good that the designers of Haskell have already spotted it and introduced an appropriate type class. As we will see later on, and especially in Chapter 12, the functor laws of \textbf{map} appear in many calculations.

There is another group of laws that involve \textbf{map}, all of which have a common theme. Consider the equations
The first equation holds only if $f$ is a strict function, but the others hold for arbitrary $f$. If we apply both sides of the equation to the empty list, we get

$$f(\text{head}[]) = \text{head}(\text{map} f[]) = \text{head}[]$$

Since the head of an empty list is undefined, we require $f$ to be strict to make the equation true.

Each of the laws has a simple interpretation. In each case you can apply the operation ($\text{head}$, $\text{tail}$, and so on) to a list and then change each element, or you can change each element first and then apply the operation. The common theme lies in the types of the operations involved:

$$\text{head} :: [a] \rightarrow a$$
$$\text{tail} :: [a] \rightarrow [a]$$
$$\text{concat} :: [[[a]]] \rightarrow [a]$$

The point about the operations is that they do not depend in any way on the nature of the list elements; they are simply functions that shuffle, discard or extract elements from lists. That is why they have polymorphic types. And functions with polymorphic types all satisfy some law that says you can change values before or after applying the function. In mathematics such functions are called natural transformations and the associated laws, naturality laws.

As another example, since $\text{reverse} :: [a] \rightarrow [a]$ we would expect that

$$\text{map} f . \text{reverse} = \text{reverse} . \text{map} f$$

Indeed this is the case. Of course, this naturality law still has to be proved.

Another law is

$$\text{concat} . \text{map} \text{concat} = \text{concat} . \text{concat}$$

The two sides assert that two ways of concatenating a list of lists of lists (either do the inner concatenations first, or do the outer concatenations first) give the same result.

Finally, here is just one property of $\text{filter}$:

$$\text{filter} p . \text{map} f = \text{map} f . \text{filter} (p . f)$$
We can prove this law by simple equational reasoning:

\[
\begin{align*}
\text{filter } p \cdot \text{map } f &= \{\text{second definition of } \text{filter}\} \\
\text{concat } \cdot \text{map } (\text{test } p) \cdot \text{map } f &= \{\text{functor property of } \text{map}\} \\
\text{concat } \cdot \text{map } (\text{map } f \cdot \text{test } (p \cdot f)) &= \{\text{functor property of } \text{map}\} \\
\text{concat } \cdot \text{map } (\text{map } f) \cdot \text{map } (\text{test } (p \cdot f)) &= \{\text{naturality of } \text{concat}\} \\
\text{map } f \cdot \text{concat } \cdot \text{map } (\text{test } (p \cdot f)) &= \{\text{second definition of } \text{filter}\} \\
\text{map } f \cdot \text{filter } (p \cdot f)
\end{align*}
\]

Laws like those above are not just of academic interest, but are deployed in finding new and better ways of expressing definitions. That’s why functional programming is the best thing since sliced bread.

### 4.7 zip and zipWith

Finally, to complete a simple toolbox of useful operations, we consider the functions `zip` and `zipWith`. The definitions in the standard prelude are:

\[
\begin{align*}
\text{zip} &: [a] \rightarrow [b] \rightarrow [(a,b)] \\
\text{zip } (x:xs) \ (y:ys) &= (x,y) \cdot \text{zip } xs \ ys \\
\text{zip } _ & _ &= [] \\
\text{zipWith} &: (a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c] \\
\text{zipWith } f \ (x:xs) \ (y:ys) &= f \ x \ y \cdot \text{zipWith } f \ xs \ ys \\
\text{zipWith } f \ _ & _ &= []
\end{align*}
\]

A caring programmer (one who doesn’t like ‘don’t care’ patterns) would have written

\[
\begin{align*}
\text{zip } [] \ ys &= [] \\
\text{zip } (x:xs) \ [] &= []
\end{align*}
\]
zip \( (x:xs) \ (y:ys) \) = \((x,y):\text{zip} \ xs \ ys\)

Both definitions use pattern matching on both arguments. You have to know that
pattern matching is applied from top to bottom and from left to right. Thus

zip \([\ ]\) undefined = \([\ ]\)
zip undefined \([\ ]\) = undefined

The definition of zip can be given another way:

zip = zipWith (,)

The operation \((,\)\) is a constructor for pairs: \((,\)\) a b = (a,b).

Here is one example of the use of zipWith. Suppose we want to determine whether
a list is in nondecreasing order. A direct definition would have:

\[
\text{nondec} :: (\text{Ord } a) \Rightarrow [a] \rightarrow \text{Bool}
\]
\[
\text{nondec} [\] = \text{True}
\]
\[
\text{nondec} [x] = \text{True}
\]
\[
\text{nondec} (x:y:xs) = (x \leq y) \&\& \text{nondec} (y:xs)
\]

But another, equivalent and shorter definition is

\[
\text{nondec} \ xs = \text{and} \ (\text{zipWith} \ (\leq) \ xs \ (\text{tail} \ xs))
\]

The function and is yet another useful function in the standard prelude. It takes a
list of booleans and returns True if all the elements are True, and False otherwise:

\[
\text{and} :: [\text{Bool}] \rightarrow \text{Bool}
\]
\[
\text{and} \ ([\]) = \text{True}
\]
\[
\text{and} \ (x:xs) = x \&\& \text{and} \ xs
\]

One final example. Consider the task of building a function position that takes a
value x and a finite list xs and returns the first position in xs (counting positions
from 0) at which x occurs. If x does not occur in the list, then \(-1\) is returned. We
can define

\[
\text{position} :: (\text{Eq } a) \Rightarrow a \rightarrow [a] \rightarrow \text{Int}
\]
\[
\text{position} \ x \ xs
\]
\[
= \text{head} \ ([j | (j,y) \leftarrow \text{zip} \ [0..] \ xs, y==x] \ ++ \ [-1])
\]

The expression \text{zip} [0..] \ xs pairs each element of \(\)xs with its position in \(\)xs. Although the first argument of \text{zip} is an infinite list, the result is a finite list whenever \(\)xs is. Observe that the problem is solved by first computing the list of all
positions at which x is found, and then taking the first element. Under lazy evaluation it is not necessary to construct the value of every element of the list in order
4.8 Common words, completed

Let's now return to Section 1.3 and complete the definition of \texttt{commonWords}. Recall that we finished with

\[
\texttt{commonWords} :: \texttt{Int} \to \texttt{[Char]} \to \texttt{[Char]}
\]

\[
\texttt{commonWords} \ n = \texttt{concat} \ . \ \texttt{map showRun} \ . \ \texttt{take} \ n \ .
\]

\[
\texttt{sortRuns} \ . \ \texttt{countRuns} \ . \ \texttt{sortWords} \ .
\]

\[
\texttt{words} \ . \ \texttt{map toLower}
\]

The only functions we have still to give definitions for are

\[
\texttt{showRun} \ \texttt{countRuns} \ \texttt{sortRuns} \ \texttt{sortWords}
\]

All the others, including \texttt{words}, are provided in the standard Haskell libraries.

The first one is easy:

\[
\texttt{showRun} :: (\texttt{Int},\texttt{Word}) \to \texttt{[Char]}
\]

\[
\texttt{showRun} \ (n,w) = w ++ ":" \ ++ \ \texttt{show} \ n \ ++ \ "\n"
\]

The second one can be defined by

\[
\texttt{countRuns} :: \texttt{[Word]} \to \texttt{[(Int,Word)]}
\]

\[
\texttt{countRuns} \ [] = []
\]

\[
\texttt{countRuns} \ (w:ws) = (1+\texttt{length} \ \texttt{us},w):\texttt{countRuns} \ \texttt{vs}
\]

\[
\texttt{where} \ (\texttt{us},\texttt{vs}) = \texttt{span} \ (==w) \ \texttt{ws}
\]

The prelude function \texttt{span} \ p \ splits a list into two, the first being the longest prefix of the list all of whose elements satisfy the test \texttt{p}, and the second being the suffix that remains. Here is the definition:

\[
\texttt{span} :: (\texttt{a} \to \texttt{Bool}) \to [\texttt{a}] \to ([\texttt{a}],[\texttt{a}])
\]

\[
\texttt{span} \ \texttt{p} \ [] = ([],[])
\]

\[
\texttt{span} \ \texttt{p} \ (x:xs) = \texttt{if} \ \texttt{p} \ x \ \texttt{then} \ (x:ys,zs)
\]

\[
\texttt{else} \ ([],x:xs)
\]

\[
\texttt{where} \ (ys,zs) = \texttt{span} \ \texttt{p} \ \texttt{xs}
\]

That leaves \texttt{sortRuns} and \texttt{sortWords}. We can import the function \texttt{sort} from \texttt{Data.List} by the command
import Data.List (sort)

Since \(\text{sort} : (\text{Ord } a) \Rightarrow [a] \rightarrow [a]\) we can then define

\[
\text{sortWords} :: [\text{Word}] \rightarrow [\text{Word}]
\]
\[
\text{sortWords} = \text{sort}
\]

\[
\text{sortRuns} :: [(\text{Int,Word})] \rightarrow [(\text{Int,Word})]
\]
\[
\text{sortRuns} = \text{reverse} \cdot \text{sort}
\]

To understand the second definition you have to know that Haskell automatically defines the comparison operation \((\leq)\) on pairs by

\[
(x_1, y_1) \leq (x_2, y_2) = (x_1 < x_2) \lor (x_1 = x_2 \land y_1 \leq y_2)
\]

You also have to know that \(\text{sort}\) sorts into ascending order. Since we want the codes in descending order of count, we just sort into ascending order and reverse the result. That, by the way, is why we defined frequency counts by having the count before the word rather than afterwards.

Instead of relying on the library function for sorting, let us end by programming a sorting function ourselves. One good way to sort is to use a divide and conquer strategy: if the list has length at most one then it is already sorted; otherwise we can divide the list into two equal halves, sort each half by using the sorting algorithm recursively, and then merge the two sorted halves together. That leads to

\[
\text{sort} :: (\text{Ord } a) \Rightarrow [a] \rightarrow [a]
\]
\[
\text{sort} [] = []
\]
\[
\text{sort} [x] = [x]
\]
\[
\text{sort} \; x\!s = \text{merge} \; (\text{sort} \; y\!s) \; (\text{sort} \; z\!s)
\]
\[
\quad \text{where} \; (y\!s,z\!s) = \text{halve} \; x\!s
\]

\[
\text{halve} \; x\!s = (\text{take} \; n \; x\!s, \text{drop} \; n \; x\!s)
\]
\[
\quad \text{where} \; n = \text{length} \; x\!s \; \div \; 2
\]

That leaves us with the definition of \(\text{merge}\), which merges two sorted lists together into one sorted list:

\[
\text{merge} :: (\text{Ord } a) \Rightarrow [a] \rightarrow [a] \rightarrow [a]
\]
\[
\text{merge} \; [] \; y\!s = y\!s
\]
\[
\text{merge} \; x\!s \; [] = x\!s
\]
\[
\text{merge} \; (x:x\!s) \; (y:y\!s)
\]
\[
\quad | \; x \leq y \quad = x : \text{merge} \; x\!s \; (y:y\!s)
\]
\[
\quad | \; \text{otherwise} \quad = y : \text{merge} \; (x:x\!s) \; y\!s
\]
In fact, many Haskell programmers wouldn’t write the last clause of \texttt{merge} in quite this way. Instead they would write

\[
\text{merge } xs'@(x:xs) \ ys'@(y:ys) \\
\mid x \leq y = \text{x:merge } xs \ ys' \\
\mid \text{otherwise } = \text{y:merge } xs' \ ys
\]

This definition uses an \emph{as-pattern}. You can see the point: rather than deconstructing a list and then reconstructing it again (a cheap but not free operation), it is better to reuse the value that we matched with. True, but it does obscure a simple mathematical equation, and we will use such patterns only very sparingly in this book.

Both \texttt{sort} and \texttt{merge} are defined recursively and it is worthwhile pointing out why the two recursions terminate. In the case of \texttt{merge} you have to see that one or other of the two arguments of \texttt{merge} decreases in size at each recursive call. Hence one of the base cases will eventually be reached. In the case of \texttt{sort} the critical observation is that if \(xs\) has length at least two, then both \(ys\) and \(zs\) have length strictly less than \(xs\), and the same argument applies. But see what happens if we had omitted the clause \(\text{sort } [x] = [x]\). Since \(1 \div 2 = 0\) we would have,

\[
\text{sort } [x] = \text{merge } (\text{sort } []) \ (\text{sort } [x])
\]

That means evaluation of \(\text{sort } [x]\) requires evaluation of \(\text{sort } [x]\), and the whole definition of \texttt{sort} spins off into an infinite loop for nonempty arguments. Checking that you have all the necessary base cases is one of the most important parts of constructing a recursive function.

\section*{4.9 Exercises}

\subsection*{Exercise A}

Which of the following equations are true for all \(xs\) and which are false?

\[
[] : xs = xs \\
[] : xs = [ [], xs] \\
x : [] = xs \\
x : [] = [xs] \\
x : xs = [xs, xs] \\
[[]] ++ xs = xs \\
[[]] ++ xs = [ [], xs] \\
[[]] ++ [xs] = [ [], xs]
\]
By the way, why didn’t we define \texttt{null} = \texttt{(==[])}?

\textbf{Exercise B}

You want to produce an infinite list of all distinct pairs \((x,y)\) of natural numbers. It doesn’t matter in which order the pairs are enumerated, as long as they all are there. Say whether or not the definition
\begin{verbatim}
allPairs = [(x,y) | x <- [0..], y <- [0..]]
\end{verbatim}
does the job. If you think it doesn’t, can you give a version that does?

\textbf{Exercise C}

Give a definition of the function
\begin{verbatim}
disjoint :: (Ord a) => [a] -> [a] -> Bool
\end{verbatim}
that takes two lists in ascending order, and determines whether or not they have an element in common.

\textbf{Exercise D}

Under what conditions do the following two list comprehensions deliver the same result?
\begin{verbatim}
[e | x <- xs, p x, y <- ys]
[e | x <- xs, y <- ys, p x]
\end{verbatim}

Compare the costs of evaluating the two expressions.

\textbf{Exercise E}

When the great Indian mathematician Srinivasan Ramanujan was ill in a London hospital, he was visited by the English mathematician G.H. Hardy. Trying to find a subject of conversation, Hardy remarked that he had arrived in a taxi with the number 1729, a rather boring number it seemed to him. Not at all, Ramanujan instantly replied, it is the first number that can be expressed as two cubes in essentially different ways: \(1^3 + 12^3 = 9^3 + 10^3 = 1729\). Write a program to find the second such number.

In fact, define a function that returns a list of all essentially different quadruples \((a,b,c,d)\) in the range \(0 < a, b, c, d \leq n\) such that \(a^3 + b^3 = c^3 + d^3\). I suggest using a list comprehension, but only after thinking carefully about what it means to say
two quadruples are essentially different. After all, \( a^3 + b^3 = c^3 + d^3 \) can be written in eight different ways.

**Exercise F**

The dual view of lists is to construct them by adding elements to the end of the list:

```haskell
data List a = Nil | Snoc (List a) a
```

Snoc is, of course, Cons backwards. With this view of lists \([1, 2, 3]\) would be represented by

```
Snoc (Snoc (Snoc Nil 1) 2) 3
```

Exactly the same information is provided by the two views but it is organised differently. Give the definitions of `head` and `last` for the snoc-view of lists, and define two functions

```haskell
toList :: [a] -> List a
fromList :: List a -> [a]
```

for converting efficiently from one view of lists to the other. (Hint: `reverse` is efficient, taking linear time to reverse a list.)

**Exercise G**

How much space is required to evaluate `length xs`? Consider the following alternative definition of `length`:

```haskell
length :: [a] -> Int
length xs = loop (0, xs)
where loop (n, []) = n
      loop (n, x:xs) = loop (n+1, xs)
```

Does the space requirement change? Does it change if we switched to eager evaluation? These questions are taken up in much more detail in Chapter 7.

**Exercise H**

The prelude function `take n` takes the first `n` elements of a list, while `drop n` drops the first `n` elements. Give recursive definitions for these functions. What are the values of

```haskell
take 0 undefined      take undefined []
```

according to your definition? A more tricky question: can you find a definition in which both the above expressions have the value `[]`? If not, why not?
Which of the following equations are valid for all integers $m$ and $n$? You don’t have to justify your answers, just try to understand what they claim to say.

$$
take n \; xs \; ++ \; drop \; n \; xs \; = \; xs
$$
$$
take \; m \; . \; drop \; n \; = \; drop \; n \; . \; take \; (m+n)
$$
$$
take \; m \; . \; take \; n \; = \; take \; (m \; \text{`min`} \; n)
$$
$$
drop \; m \; . \; drop \; n \; = \; drop \; (m+n)
$$

The standard prelude function \texttt{splitAt n} can be defined by

$$
\texttt{splitAt n \; xs} = (\texttt{take \; n \; xs} , \texttt{drop \; n \; xs})
$$

Though clear, the above definition is maybe a little inefficient as it involves processing \texttt{xs} twice. Give a definition of \texttt{splitAt} that traverses the list only once.

\textbf{Exercise I}

Which of the following statements about the equation

$$
\text{map \; (f \; . \; g \; \; xs} \; = \; \text{map \; f \; (map \; g \; xs)}
$$
do you agree with, and which do you disagree with (again, no justification is required)?

1. It’s not true for all \texttt{xs}; it depends on whether \texttt{xs} is a finite list or not.

2. It’s not true for all \texttt{f} and \texttt{g}; it depends on whether \texttt{f} and \texttt{g} are strict functions or not.

3. It’s true for all lists \texttt{xs}, finite, partial or infinite, and for all \texttt{f} and \texttt{g} of the appropriate type. In fact \texttt{map \; (f \; . \; g)} = \texttt{map \; f \; \; map \; g} is a much neater alternative.

4. It looks true, but it has to be proved so from the definition of \texttt{map} and the definition of functional composition.

5. Used right-to-left, it expresses a program optimisation: two traversals of a list are replaced by one.

6. It’s not an optimisation under lazy evaluation because \texttt{map \; g \; xs} is not computed in its entirety before evaluation of \texttt{map \; f} on the result begins.

7. Whether or not it is computed in pieces or as a whole, the right-hand side does produce an intermediate list, while the left-hand side doesn’t. It is a rule for optimising a program even under lazy evaluation.
Exercise J

Here are some equations; at least one of them is false. Which are the true ones, and which are false? Once again, you do not have to provide any justification for your answers, the aim is just to look at some equations and appreciate what they are saying.

\[
\begin{align*}
\text{map } f \ . \ \text{take } n & = \text{take } n \ . \ \text{map } f \\
\text{map } f \ . \ \text{reverse} & = \text{reverse} \ . \ \text{map } f \\
\text{map } f \ . \ \text{sort} & = \text{sort} \ . \ \text{map } f \\
\text{map } f \ . \ \text{filter } p & = \text{map } \text{fst} \ . \ \text{filter } \text{snd} \ . \ \text{map } (\text{fork} \ (f,p)) \\
\text{filter } (p \ . \ g) & = \text{map } (\text{invertg}) \ . \ \text{filter } p \ . \ \text{map } g \\
\text{reverse} \ . \ \text{concat} & = \text{concat} \ . \ \text{reverse} \ . \ \text{map } \text{reverse} \\
\text{filter } p \ . \ \text{concat} & = \text{concat} \ . \ \text{map } (\text{filter } p)
\end{align*}
\]

In the fifth equation assume \( \text{invertg} \) satisfies \( \text{invertg} \ . \ g = \text{id} \). The function \( \text{fork} \) in the fourth equation is defined by

\[
\text{fork} :: (a \to b,a \to c) \to a \to (b,c)
\]

\[
\text{fork} \ (f,g) \ x = (f \ x, \ g \ x)
\]

Exercise K

Define unzip and cross by

\[
\begin{align*}
\text{unzip} & = \text{fork} \ (\text{map } \text{fst}, \ \text{map } \text{snd}) \\
\text{cross} (f,g) & = \text{fork} \ (f \ . \ \text{fst}, \ g \ . \ \text{snd})
\end{align*}
\]

What are the types of these functions?

Prove by simple equational reasoning that

\[
\text{cross} \ (\text{map } f, \ \text{map } g) \ . \ \text{unzip} = \text{unzip} \ . \ \text{map} \ (\text{cross} \ (f,g))
\]

You can use the functor laws of \text{map} and the following rules:

\[
\begin{align*}
\text{cross} \ (f,g) \ . \ \text{fork} \ (h,k) & = \text{fork} \ (f \ . \ h,g \ . \ k) \\
\text{fork} \ (f,g) \ . \ h & = \text{fork} \ (f \ . \ h,g \ . \ h) \\
\text{fst} \ . \ \text{cross} \ (f,g) & = f \ . \ \text{fst} \\
\text{snd} \ . \ \text{cross} \ (f,g) & = g \ . \ \text{snd}
\end{align*}
\]

Exercise L

Continuing from the previous exercise, prove that

\[
\text{cross} \ (f,g) \ . \ \text{cross} \ (h,k) = \text{cross} \ (f \ . \ h,g \ . \ k)
\]
We also have \( \text{cross} \ (\text{id}, \text{id}) = \text{id} \) (Why?). So it looks like \( \text{cross} \) has functor-like properties, except that it takes a pair of functions. Yes, it's a \textit{bifunctor}. That suggests a generalisation:

\[
\text{class Bifunctor } p \text{ where }
\]
\[
\text{bimap} :: (a \to b) \to (c \to d) \to p \ a \ c \to p \ b \ d
\]

The arguments to \text{bimap} are given one by one rather than paired. Express \( \text{cross} \) in terms of \text{bimap} for the instance \text{Pair} of \text{Bifunctor}, where

\[
\text{type Pair } a \ b = (a,b)
\]

Now consider the data type

\[
\text{data Either } a \ b = \text{Left } a \mid \text{Right } b
\]

Construct the instance \text{Either} of \text{Bifunctor}.

### 4.10 Answers

**Answer to Exercise A**

Only the following three equations are true:

\[
\begin{align*}
\text{xs:[]} &= [\text{xs}] \\
[[]] ++ [\text{xs}] &= [[], \text{xs}] \\
[\text{xs}] ++ [] &= [\text{xs}]
\end{align*}
\]

If we defined \text{null} by \text{null} = (==[]), then its type would have to be the more restrictive

\[
\text{null} :: (\text{Eq } a) \Rightarrow [a] \to \text{Bool}
\]

That means you can only use an equality test on lists if the list elements can be compared for equality. Of course, the empty list contains no elements, so (==) is not needed.

**Answer to Exercise B**

No, \text{allPairs} produces the infinite list

\[
\text{allPairs} = [(0, y) \mid y \leftarrow [0..]]
\]

One alternative, which lists the pairs in ascending order of their sum, is

\[
\text{allPairs} = [(x, d-x) \mid d \leftarrow [0..], x \leftarrow [0..d]]
\]
Answer to Exercise C

The definition is

\[
\text{disjoint } xs \; [] = \text{True} \\
\text{disjoint } [] \; ys = \text{True} \\
\text{disjoint } xs'@(x:xs) \; ys'@(y:ys) \\
    \mid x < y \; = \text{disjoint } xs \; ys' \\
    \mid x == y \; = \text{False} \\
    \mid x > y \; = \text{disjoint } xs' \; ys
\]

We used an as-pattern, just to be clever.

Answer to Exercise D

They deliver the same result only if \(ys\) is a finite list:

\[
\text{ghci} \triangleright [1 \mid x \leftarrow [1,3], \; \text{even } x, \; y \leftarrow \text{undefined}] \\
[]
\]

\[
\text{ghci} \triangleright [1 \mid x \leftarrow [1,3], \; y \leftarrow \text{undefined}, \; \text{even } x]
\]

*** Exception: Prelude.undefined

\[
\text{ghci} \triangleright [1 \mid x \leftarrow [1,3], \; \text{even } x, \; y \leftarrow [1..]]
\]

[]

\[
\text{Prelude} \triangleright [1 \mid x \leftarrow [1,3], \; y \leftarrow [1..], \; \text{even } x]
\]

{Interrupted}

When they do deliver the same result, the former is more efficient.

Answer to Exercise E

One way of generating essentially different quadruples is to restrict the quadruple \((a, b, c, d)\) to values satisfying \(a \leq b\) and \(c \leq d\) and \(a < c\). Hence

\[
\text{quads } n = [(a,b,c,d) \mid a \leftarrow [1..n], \; b \leftarrow [a..n], \\
\quad c \leftarrow [a+1..n], d \leftarrow [c..n], \\
\quad a^3 + b^3 == c^3 + d^3]
\]

The second such number is \(4104 = 2^3 + 16^3 = 9^3 + 15^3\).

Answer to Exercise F

\[
\begin{align*}
\text{head} & : \text{List } a \rightarrow a \\
\text{head} (\text{Snoc } \text{Nil } x) & = x \\
\text{head} (\text{Snoc } xs \; x) & = \text{head } xs
\end{align*}
\]
last :: List a -> a
last (Snoc xs x) = x

toList :: [a] -> List a
toList = convert . reverse
  where convert [] = Nil
        convert (x:xs) = Snoc (convert xs) x

fromList :: List a -> [a]
fromList = reverse . convert
  where convert Nil = []
        convert (Snoc xs x) = x:convert xs

Answer to Exercise G

It requires a linear amount of space since the expression

1 + (1 + (1 + ... (1 + 0)))

is built up in memory. The space requirement for the second definition of length does not change under lazy evaluation since the expression

loop (((0 + 1) + 1) + 1 ... +1),[])

is built up in memory. But under eager evaluation the length of a list can be computed using constant extra space.

Answer to Exercise H

take, drop :: Int -> [a] -> [a]
take n [] = []
take n (x:xs) = if n==0 then [] else x:take (n-1) xs
drop n [] = []
drop n (x:xs) = if n==0 then x:xs else drop (n-1) xs

With this definition of take we have

take undefined [] = []
take 0 undefined = undefined

With the alternative

take n xs | n==0 = []
          | null xs = []
          | otherwise = head xs: take (n-1) (tail xs)

we have
\[ \text{take undefined } [] = \text{undefined} \quad \text{take 0 undefined } = [] \]

The answer to the tricky question is: no. Either argument \( n \) or argument \( xs \) has to be examined and, whichever happens first, \( \bot \) is the result if \( \bot \) is the value of that argument.

All four equations are valid for all lists \( xs \) and for all \( m, n \neq \bot \), under either definition.

The function \( \text{splitAt } n \) can be defined by

\[
\text{splitAt} :: \text{Int} \to \text{[a]} \to ([\text{a}], [\text{a}])
\]
\[
\text{splitAt} \ n \ [] = ([],[])
\]
\[
\text{splitAt} \ n \ (x:xs) = \text{if } n==0 \text{ then } ([],x:xs) \text{ else } (x:ys,zs)
\]
\[
\text{where } (ys,zs) = \text{splitAt} \ (n-1) \ xs
\]

**Answer to Exercise I**

I would agree with (3), (4), (5) and (7).

**Answer to Exercise J**

The only false equation is \( \text{map } f . \text{sort} = \text{sort} . \text{map } f \) which is true only if \( f \) is order-preserving, i.e. \( x \leq y \equiv f(x) \leq f(y) \).

**Answer to Exercise K**

\[
\text{unzip} :: \text{[(a,b)]} \to ([\text{a}],[\text{b}])
\]
\[
\text{cross} :: (a \to b, c \to d) \to (a,c) \to (b,d)
\]

The calculation is

\[
\text{cross} \ (\text{map } f, \text{map } g) . \text{unzip}
\]
\[
= \ \{\text{definition of unzip}\}
\]
\[
\text{cross} \ (\text{map } f, \text{map } g) . \text{fork} \ (\text{map } \text{fst}, \text{map } \text{snd})
\]
\[
= \ \{\text{law of cross and fork}\}
\]
\[
\text{fork} \ (\text{map } f \ . \text{map } \text{fst}, \text{map } g \ . \text{map } \text{snd})
\]
\[
= \ \{\text{law of map}\}
\]
\[
\text{fork} \ (\text{map } (f \ . \text{fst}), \text{map } (g \ . \text{snd}))
\]
We seem to be stuck, as no law applies. Try the right-hand side:

\[
\begin{align*}
\text{unzip} \cdot \text{map} \ (\text{cross} \ (f,g)) \\
= & \quad \{\text{definition of unzip}\} \\
\text{fork} \ (\text{map} \ \text{fst}, \ \text{map} \ \text{snd}) \ . \ \text{map} \ (\text{cross} \ (f,g)) \\
= & \quad \{\text{law of fork}\} \\
\text{fork} \ (\text{map} \ \text{fst} \ . \ \text{map} \ (\text{cross} \ (f,g)), \\
\text{map} \ \text{snd} \ . \ \text{map} \ (\text{cross} \ (f,g))) \\
= & \quad \{\text{law of map}\} \\
\text{fork} \ (\text{map} \ (\text{fst} \ . \ \text{cross} \ (f,g)), \\
\text{map} \ (\text{snd} \ . \ \text{cross} \ (f,g))) \\
= & \quad \{\text{laws of fst and snd}\} \\
\text{fork} \ (\text{map} \ (f \ . \ \text{fst}), \ \text{map} \ (g \ . \ \text{snd}))
\end{align*}
\]

Phew. Both sides have reduced to the same expression. That is often the way with calculations: one side doesn’t always lead easily to the other, but both sides reduce to the same result.

The calculations we have seen so far have all been carried out at the function level. Such a style of definition and proof is called \textit{point-free} (and also \textit{pointless} by some jokers). Point-free proofs are what the automatic calculator of Chapter 12 produces. The point-free style is very slick, but it does necessitate the use of various \textit{plumbing combinators}, such as \textit{fork} and \textit{cross}, to pass arguments to functions. Plumbing combinators push values around, duplicate them and even eliminate them. As an example of the last kind,

\[
\begin{align*}
\text{const} :: a \rightarrow b \rightarrow a \\
\text{const} \ x \ y &= x
\end{align*}
\]

This little combinator is in the standard prelude and can be quite useful on occasion.

Two more plumbing combinators, also defined in the standard prelude, are \textit{curry} and \textit{uncurry}:

\[
\begin{align*}
\text{curry} :: ((a, b) \rightarrow c) \rightarrow a \rightarrow b \rightarrow c \\
\text{curry} \ f \ x \ y &= f \ (x,y)
\end{align*}
\]

\[
\begin{align*}
\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow (a,b) \rightarrow c \\
\text{uncurry} \ f \ (x,y) &= f \ x \ y
\end{align*}
\]

A \textit{curried} function is a function that takes its arguments one at a time, while a non-curried function takes a single, tupled argument. The key advantage of curried
functions is that they can be *partially applied*. For instance, \( \text{take } n \) is a perfectly valid function in its own right, and so is \( \text{map } f \). That is why we have used curried functions from the start.

By the way, curried functions are named after Haskell B. Curry, an American logician. And, yes, that is where Haskell got its name.

**Answer to Exercise L**

\[
\begin{align*}
\text{cross } (f,g) . \text{cross } (h,k) \\
= & \quad \{\text{definition of cross}\} \\
\text{cross } (f,g) . \text{fork } (h . \text{fst}, k . \text{snd}) \\
= & \quad \{\text{law of cross and fork}\} \\
\text{fork } (f . h . \text{fst}, g . k . \text{snd}) \\
= & \quad \{\text{definition of cross}\} \\
\text{cross } (f . h, g . k)
\end{align*}
\]

We have \( \text{cross } = \text{uncurry } \text{bimap} \), where \text{uncurry} was defined in the previous answer.

Here is the instance of Either:

```haskell
instance Bifunctor Either where
  bimap f g (Left x) = Left (f x)
  bimap f g (Right y) = Right (g y)
```

Most of the functions introduced in this chapter can be found in the Haskell standard prelude. Functors, bifunctors, and natural transformations are explained in books about Category Theory. Two such are *Basic Category Theory for Computer Scientists* (MIT Press, 1991) by Benjamin Pierce, and *The Algebra of Programming* (Prentice Hall, 1997) by Richard Bird and Oege de Moor.

Also on the subject of laws, read Phil Wadler’s influential article *Theorems for free!* which can be found at

\[\text{homepages.inf.ed.ac.uk/wadler/papers/free/}\]
In mathematics, the so-called taxicab number $\text{taxicab}(n)$ is the smallest number that can be expressed as the sum of two positive cubes in $n$ distinct ways. So $1729 = \text{taxicab}(2)$. Google ‘taxicab numbers’ for more information.
Chapter 5

A simple Sudoku solver

HOW TO PLAY: Fill in the grid so that every row, every column and every 3 × 3 box contains the digits 1–9. There’s no maths involved. You solve the puzzle with reasoning and logic.

Advice on how to play Sudoku, the Independent

This chapter is devoted to an extended exercise in the use of lists to solve problems, and in the use of equational reasoning to reason about them and to improve efficiency.

The game of Sudoku is played on a 9 by 9 grid, though other sizes are also possible. Given a matrix, such as that in Figure 5.1, the idea is to fill in the empty cells with the digits 1 to 9 so that each row, column and 3 × 3 box contains the numbers 1 to 9. In general there may be any number of solutions, though in a good Sudoku puzzle there should always be a unique solution. Our aim is to construct a program to solve Sudoku puzzles. Specifically, we will define a function solve for computing a list of all the ways a given grid may be completed. If only one solution is wanted, then we can take the head of the list. Lazy evaluation means that only the first result will then be computed.

We begin with a specification, then use equational reasoning to calculate a more efficient version. There’s no maths involved, just reasoning and logic!

5.1 Specification

Here are the basic data types of interest, starting with matrices:

```haskell
type Matrix a = [Row a]
```
A simple Sudoku solver

![A Sudoku grid](image)

**Figure 5.1** A Sudoku grid

```haskell
type Row a = [a]

The two type synonyms say nothing more than that Matrix a is a synonym for \[[a]\]. But the way it is said emphasises that a matrix is a list of *rows*; more precisely, a $m \times n$ matrix is a list of $m$ rows in which each row is a list with the same length $n$. Haskell type synonyms cannot enforce such constraints, though there are languages, called *dependently-typed* languages, that can.

A grid is a $9 \times 9$ matrix of digits:

```haskell
  type Grid = Matrix Digit
  type Digit = Char
```

The valid digits are 1 to 9 with 0 standing for a blank:

```haskell
digits :: [Char]
digits = ['1' .. '9']

  blank :: Digit -> Bool
  blank = (== '0')
```

Recall that Char is also an instance of the type class Enum, so \(['1' .. '9']\) is a valid expression and does indeed return the list of nonzero digits.

We will suppose for simplicity that the input grid contains only digits and blanks, so we do not have to check for the input being well-formed. But should we also insist that no non-blank digit is repeated in any row, column or box? If there were such repetitions there would be no solution. We postpone this decision until after we see how the rest of the algorithm pans out.
Now for the specification. The aim is to write down the simplest and clearest specification without regard to how efficient the result might be. That’s a key difference between functional programming and other forms of program construction: we can always begin with a clear and simple, though possibly extremely inefficient definition of `solve`, and then use the laws of functional programming to massage the computation into one that takes acceptable time and space.

One possibility is first to construct a list of all possible correctly filled grids, a vastly long but still finite list, and then to test the given grid against each of them to identify those whose entries match the given non-blank ones. Certainly that approach takes the idea of an inefficient specification to the extreme. Another reasonable alternative is to start with the given grid and to complete it by filling in every possible choice for the blank entries. The result will be a list of filled grids. Then we can filter this list for those that don’t contain duplicates in any row, box or column. This specification is implemented by

```haskell
solve :: Grid -> [Grid]
solve = filter valid . completions
```

where the subsidiary functions have types

```haskell
completions :: Grid -> [Grid]
valid :: Grid -> Bool
```

Let us work on `completions` first and consider `valid` afterwards. One way of defining `completions` is by a two-step process:

```haskell
completions = expand . choices
```

where

```haskell
choices :: Grid -> Matrix [Digit]
expand :: Matrix [Digit] -> [Grid]
```

The function `choices` installs the available digits for each cell:

```haskell
choices = map (map choice)
choice d = if blank d then digits else [d]
```

If the cell is blank, then all digits are installed as possible choices; otherwise there is only one choice and a singleton is returned. If we want to apply `f` to every element of a matrix, then `map (map f)` is the function to use because, after all, a matrix is just a list of lists.

After applying `choices` we obtain a matrix each of whose entries is a list of digits. What we want to do next is to define `expand` to convert this matrix into a list of
A simple Sudoku solver

grids by installing all the choices in all possible ways. That seems a little difficult
to think about, so let’s consider a simpler problem first, namely when instead of a
9×9 matrix we have a list of length 3. Suppose we want to convert

\[
[[1,2,3],[2],[1,3]]
\]

into the list

\[
[[1,2,1],[1,2,3],[2,2,1],[2,2,3],[3,2,1],[3,2,3]]
\]

The second list of lists arises by taking, in all possible ways, one element from
the first list, one element from the second list and one element from the third list.
Let us call the function that does this \(cp\) (short for ‘cartesian product’, which is exactly what a mathematician would call it). There doesn’t seem to be any clever
way of computing \(cp\) in terms of other functions, so we adopt the default strategy
of defining this function by breaking up its argument list into two possibilities, the
empty list \([\,]\) and a nonempty list \(xs:xss\). You might guess the definition of \(cp\) \([\,]\)
but you would probably be wrong; the better alternative is to think about the second
case first. Suppose we assume

\[cp\ (xs:xss) = [[2,1],[2,3]]\]

How can we extend this definition to one for \(cp\ (\ [1,2,3]: [[2],[1,3]] )\)? The
answer is to prefix 1 to every element of \(cp\ (xs:xss)\), then to prefix 2 to every
element of the same list, and finally to prefix 3 to every element. That process can
be expressed neatly using a list comprehension:

\[cp\ (xs:xss) = [x:ys | x ← xs, ys ← cp\ xss]\]

In words, prefix every element of \(xs\) to every element of \(cp\ xss\) in all possible
ways.

If your nose is good at sniffing out inefficiencies, you might suspect that this one-
liner is not the best possible, and you would be right. We will return to this point
in Section 7.3, but let’s just say that a more efficient definition is

\[cp\ (xs:xss) = [x:ys | x ← xs, ys ← yss]\]
\[where\ yss = cp\ xss\]

This version guarantees that \(cp\ xss\) is computed just once.

Now, what is \(cp\ ([\,])\)? The answer is not \([\,]\) but \([\ [\,]\ ]\). To see why the first is wrong,
consider a little calculation:

\[cp\ [xs] = cp\ (xs:\ [\,])\]
\[= [x:ys | x ← xs, ys ← cp\ [\,]]\]
In fact with \( \text{cp} \ [\] = [\] \) we can show that \( \text{cp} \ xss = [\] \) for all lists \( xss \). So that definition is clearly wrong. You can check that the second alternative, \([[]]\), does give what is wanted.

Summarising, we can define \( \text{cp} \) by

\[
\text{cp} :: [[a]] \rightarrow [[a]] \\
\text{cp} [\] = [[]] \\
\text{cp} (xs:xss) = [x:ys \mid x \leftarrow xs, ys \leftarrow yss] \\
\text{where} \ yss = \text{cp} xss
\]

For example,

ghci> cp [[1],[2],[3]]
[[1,2,3]]

ghci> cp [[1,2],[[]],[4,5]]
[]

In the second example there is no possible choice from the middle list, so the empty list is returned.

But what about matrices and \( \text{expand} \), which does the same thing on matrices as \( \text{cp} \) does on lists? You will have to think a bit before seeing that what is wanted is

\[
\text{expand} :: \text{Matrix} \ [\text{Digit}] \rightarrow \text{[Grid]} \\
\text{expand} = \text{cp} \ . \ \text{map} \ \text{cp}
\]

That looks a little cryptic, but \( \text{map} \ \text{cp} \) returns a list of all possible choices for each row, and so applying \( \text{cp} \) to the result installs each choice for the rows in all possible ways. The general type of the right-hand side is

\[
\text{cp} \ . \ \text{map} \ \text{cp} :: [[[a]]] \rightarrow [[[a]]]
\]

and the declared type of \( \text{expand} \) is just a restricted version of this type. Note that \( \text{expand} \) returns the empty list if any element in any row is the empty list.

Finally, a valid grid is one in which no row, column or box contains duplicates:

\[
\text{valid} :: \text{Grid} \rightarrow \text{Bool} \\
\text{valid} g = \text{all nodups (rows g)} \&\& \text{all nodups (cols g)} \&\& \text{all nodups (boxs g)}
\]
The prelude function \texttt{all} is defined by
\[
\texttt{all } p = \texttt{and} . \texttt{map } p
\]

Applied to a finite list \texttt{xs} the function \texttt{all} \texttt{p} returns \texttt{True} if all elements of \texttt{xs} satisfy \texttt{p}, and \texttt{False} otherwise. The function \texttt{nodups} can be defined by
\[
\texttt{nodups} :: (\texttt{Eq } a) => \texttt{[a]} \rightarrow \texttt{Bool}
\]
\[
\texttt{nodups } [] = \texttt{True}
\]
\[
\texttt{nodups } (x:xs) = \texttt{all } (/=x) \texttt{ xs } \&\& \texttt{ nodups } \texttt{ xs}
\]

Evaluation of \texttt{nodups} on a list of length \(n\) takes time proportional to \(n^2\). As an alternative we could sort the list and check that it is strictly increasing. Sorting can be done in time proportional to \(n \log n\) steps. That seems a big saving over \(n^2\). However, with \(n = 9\), it is not clear that using an efficient sorting algorithm is worthwhile. What would you prefer: \(2n^2\) steps or \(100n \log_2 n\) steps?

It remains to define \texttt{rows}, \texttt{cols} and \texttt{boxs}. If a matrix is given by a list of its rows, then \texttt{rows} is just the identity function on matrices:
\[
\texttt{rows} :: \texttt{Matrix } a \rightarrow \texttt{Matrix } a
\]
\[
\texttt{rows } = \texttt{id}
\]

The function \texttt{cols} computes the transpose of a matrix. Thus if a matrix consists of \(m\) rows, where each row has length \(n\), the transpose is a list of \(n\) rows, where each row has length \(m\). Assuming both \(m\) and \(n\) are not zero, we can define
\[
\texttt{cols} :: \texttt{Matrix } a \rightarrow \texttt{Matrix } a
\]
\[
\texttt{cols } [xs] = [[x] \mid x \leftarrow xs]
\]
\[
\texttt{cols } (xs:xss) = \texttt{zipWith } (_,) \texttt{ xs } \texttt{(cols } xss)
\]

It is usual in matrix algebra to suppose that the matrix is nonempty, and that certainly suffices here, but it is interesting to consider what happens if we allow \(m\) or \(n\) to be zero. This point is taken up in the exercises.

The function \texttt{boxs} is a little more interesting. We give the definition first and explain it afterwards:
\[
\texttt{boxs} :: \texttt{Matrix } a \rightarrow \texttt{Matrix } a
\]
\[
\texttt{boxs } = \texttt{map ungroup} . \texttt{ungroup} . \texttt{map cols} . \texttt{group} . \texttt{map group}
\]

The function \texttt{group} splits a list into groups of three:
\[
\texttt{group} :: [a] \rightarrow [[a]]
\]
5.2 Lawful program construction

The function `ungroup` takes a grouped list and ungroups it:

```haskell
ungroup :: [[a]] -> [a]
ungroup = concat
```

The action of `boxs` in the $4 \times 4$ case, when `group` splits a list into groups of two rather than three, is illustrated by the picture

\[
\begin{pmatrix}
  a & b & c & d \\
  e & f & g & h \\
  i & j & k & l \\
  m & n & o & p \\
\end{pmatrix}
\rightarrow
\begin{pmatrix}
  \begin{pmatrix}
    a & b \\
    e & f \\
  \end{pmatrix} &
  \begin{pmatrix}
    c & d \\
    g & h \\
  \end{pmatrix} \\
  \begin{pmatrix}
    i & j \\
    k & l \\
  \end{pmatrix} &
  \begin{pmatrix}
    m & n \\
    o & p \\
  \end{pmatrix}
\end{pmatrix}
\]

Grouping produces a list of matrices; transposing each matrix and ungrouping yields the boxes, as a matrix whose rows are the boxes of the original matrix.

5.2 Lawful program construction

Observe that instead of thinking about matrices in terms of indices, and doing arithmetic on indices to identify the rows, columns and boxes, we have gone for definitions of these functions that treat the matrix as a complete entity in itself. This style has aptly been called wholemeal programming. Wholemeal programming is good for you: it helps to prevent a disease called indexitis, and encourages lawful program construction.

For example, here are three laws that are valid on Sudoku grids:

```haskell
rows . rows = id
cols . cols = id
boxs . boxs = id
```

In other words, all three functions are involutions. The first two are valid on all matrices, and the third is valid on arbitrary $n^2 \times n^2$ matrices (provided we change
the definition of group to group by \( n \). Two are easy to prove, but one is more difficult. The difficult law is not the one about boxs, as you might expect, but the involution property of cols. Though it is intuitively obvious that transposing a matrix twice gets you back to the original matrix, proving it from the definition of cols is a little tricky and we won’t go into details, basically because we haven’t yet discussed the tools available to do the job.

By contrast, here is the proof of the involution property of boxs. The proof is by simple equational reasoning. It makes use of various laws, including the functor laws of map, the fact that \( \text{id} \) is the identity element of composition, and the facts that

\[
\text{ungroup} \cdot \text{group} = \text{id} \\
\text{group} \cdot \text{ungroup} = \text{id}
\]

The second equation is valid only on grouped lists, but that will be the case in the calculation to come.

We will talk through the proof rather than lay everything out in a long chain. The starting point is to use the definition of boxs to rewrite boxs . boxs:

\[
\text{map ungroup} \cdot \text{ungroup} \cdot \text{map cols} \cdot \text{group} \cdot \text{map group} . \\
\text{map ungroup} \cdot \text{ungroup} \cdot \text{map cols} \cdot \text{group} \cdot \text{map group}
\]

The middle expression map group . map ungroup simplifies to \( \text{id} \) using the functor law of map and the property that group and ungroup are inverses. That gives

\[
\text{map ungroup} \cdot \text{ungroup} \cdot \text{map cols} \cdot \text{group} . \\
\text{ungroup} \cdot \text{map cols} \cdot \text{group} \cdot \text{map group}
\]

An appeal to \( \text{group} \cdot \text{ungroup} = \text{id} \) gets us to

\[
\text{map ungroup} \cdot \text{ungroup} \cdot \text{map cols} . \\
\text{map cols} \cdot \text{group} \cdot \text{map group}
\]

The functor law of map and the involution property of cols now gets us to

\[
\text{map ungroup} \cdot \text{ungroup} \cdot \text{group} \cdot \text{map group}
\]

And the proof is finished off using \( \text{ungroup} \cdot \text{group} = \text{id} \) twice more. As you can see, it’s a very simple calculation.

Here are three more laws, valid on \( N^2 \times N^2 \) matrices of choices:

\[
\text{map rows} \cdot \text{expand} = \text{expand} \cdot \text{rows} \\
\text{map cols} \cdot \text{expand} = \text{expand} \cdot \text{cols}
\]
5.3 Pruning the matrix of choices

map boxs . expand = expand . boxs

We will make use of these laws in a short while.

Finally, here are two laws about \(\text{cp}\):

\[
\begin{align*}
\text{map} \ (\text{map} \ f) \ . \ \text{cp} &= \text{cp} \ . \ \text{map} \ (\text{map} \ f) \\
\text{filter} \ (\text{all} \ p) \ . \ \text{cp} &= \text{cp} \ . \ \text{map} \ (\text{filter} \ p)
\end{align*}
\]

The first law, a naturality law, is suggested solely by the type of \(\text{cp}\); we saw similar laws in the previous chapter. The second law says that as an alternative to taking the cartesian product of a list of lists, and then retaining only those lists all of whose elements satisfy \(p\), we can first filter the original lists to retain only those elements that satisfy \(p\) and then take the cartesian product. As the previous sentence illustrates, one equation can be worth a thousand words.

5.3 Pruning the matrix of choices

Summarising what we have at the moment,

\[
\text{solve} :: \text{Grid} \rightarrow [\text{Grid}]
\]

\[
\text{solve} = \text{filter valid} \ . \ \text{expand} \ . \ \text{choices}
\]

Though executable in theory, this definition of \text{solve} is hopeless in practice. Assuming about 20 of the 81 entries are fixed initially, there are about \(9^{61}\), or

\[
\text{ghci}> 9^{61}
16173092699229880893718618465586445357583280647840659957609
\]

grids to check! We therefore need a better approach.

To make a more efficient solver, an obvious idea is to remove any choices from a cell \(c\) that already occur as singleton entries in the row, column and box containing \(c\). A singleton entry corresponds to a fixed choice. We therefore seek a function

\[
\text{prune} :: \text{Matrix} \ [\text{Digit}] \rightarrow \text{Matrix} \ [\text{Digit}]
\]

so that

\[
\text{filter valid} \ . \ \text{expand} = \text{filter valid} \ . \ \text{expand} \ . \ \text{prune}
\]

How can we define \text{prune}? Well, since a matrix is a list of rows, a good place to start is by pruning a single row. The function \text{pruneRow} is defined by

\[
\begin{align*}
\text{pruneRow} :: \text{Row} \ [\text{Digit}] \rightarrow \text{Row} \ [\text{Digit}] \\
\text{pruneRow row} &= \text{map} \ (\text{remove fixed}) \ \text{row}
\end{align*}
\]
A simple Sudoku solver

where fixed = [d | [d] <- row]

The fixed choices are the singleton entries in each row. The definition of fixed uses a list comprehension involving a pattern: all elements of row that are not singletons are discarded.

The function remove removes the fixed choices from any choice that is not fixed:

remove :: [Digit] -> [Digit] -> [Digit]
remove ds [x] = [x]
remove ds xs = filter (\notElem\ ds) xs

The standard prelude function notElem is defined by

notElem :: (Eq a) => a -> [a] -> Bool
notElem x xs = all (/= x) xs

Here are a couple of examples of the use of pruneRow:

ghci> pruneRow [[6],[1,2],[3],[1,3,4],[5,6]]
[[6],[1,2],[3],[1,4],[5]]

ghci> pruneRow [[6],[3,6],[3],[1,3,4],[4]]
[[6],[],[3],[1],[4]]

In the first example, [6] and [3] are the fixed choices; removing these choices from the other entries reduces the last entry to a fixed choice. In the second example, removing the fixed choices reduces the second entry to the empty list of choices.

The function pruneRow satisfies the equation

\[ \text{filter nodups . cp = filter nodups . cp . pruneRow} \]

In words, this equation says that pruning a row will not throw away any list that contains no duplicates. We will also make use of this law in a short while.

We are now nearly ready for a calculation that will determine the function prune. Nearly, but not quite because we are going to need two more laws: If \( f . f = \text{id} \), then

\[ \text{filter (p . f) = map f . filter p . map f} \]
\[ \text{filter (p . f) . map f = map f . filter p} \]
The second law follows from the first (Why?). Here is the proof of the first law:

\[
\begin{align*}
\text{map } f \cdot \text{filter } p \cdot \text{map } f &= \{ \text{we proved in the previous chapter that} \\
&\quad \text{filter } p \cdot \text{map } f = \text{map } f \cdot \text{filter } (p \cdot f) \} \\
\text{map } f \cdot \text{map } f \cdot \text{filter } (p \cdot f) &= \{ \text{functor law of } \text{map} \text{ and } f \cdot f = \text{id} \} \\
\text{filter } (p \cdot f)
\end{align*}
\]

Now for the main calculation. The starting point is to use the definition of \textit{valid} to rewrite the expression \text{filter valid . expand} in the form

\[
\begin{align*}
\text{filter valid . expand} &= \text{filter (all nodups . boxs) .} \\
&\quad \text{filter (all nodups . cols) .} \\
&\quad \text{filter (all nodups . rows) . expand}
\end{align*}
\]

The order in which the filters appear on the right is not important. The plan of attack is to send each of these filters into battle with \text{expand}. For example, in the boxs case we can calculate:

\[
\begin{align*}
\text{filter (all nodups . boxs) . expand} &= \{ \text{above law of } \text{filter}, \text{since boxs . boxs = id} \} \\
\text{map boxs . filter (all nodups) . map boxs . expand} &= \{ \text{since map boxs . expand = expand . boxs} \} \\
\text{map boxs . filter (all nodups) . expand . boxs} &= \{ \text{definition of } \text{expand} \} \\
\text{map boxs . filter (all nodups) . cp . map cp . boxs} &= \{ \text{since filter (all p) . cp = cp . map (filter p)} \} \\
\text{map boxs . cp . map (filter nodups) . map cp . boxs} &= \{ \text{functor law of } \text{map} \} \\
\text{map boxs . cp . map (filter nodups . cp) . boxs}
\end{align*}
\]

Now we use the property

\[
\text{filter nodups . cp = filter nodups . cp . pruneRow}
\]

to rewrite the final expression in the form

\[
\text{map boxs . cp . map (filter nodups . cp . pruneRow) . boxs}
\]
The remaining steps essentially repeat the calculation above, but in the reverse direction:

\[
\text{map boxes . cp . map (filter nodups . cp . pruneRow) . boxes = \{functor law of map\}}
\]

\[
\text{map boxes . cp . map (filter nodups) . map (cp . pruneRow) . boxes = \{since cp . map (filter p) = filter (all p) . cp\}}
\]

\[
\text{map boxes . filter (all nodups) . cp . map (cp . pruneRow) . boxes = \{functor law of map\}}
\]

\[
\text{map boxes . filter (all nodups) . cp . map cp . map pruneRow . boxes = \{definition of expand\}}
\]

\[
\text{map boxes . filter (all nodups) . expand . map pruneRow . boxes = \{law of filter since boxes . boxes = id\}}
\]

\[
\text{filter (all nodups . boxes) . map boxes . expand . map pruneRow . boxes = \{since map boxes . expand = expand . boxes\}}
\]

\[
\text{filter (all nodups . boxes) . expand . boxes . map pruneRow . boxes = \{introducing pruneBy f = f . pruneRow . f\}}
\]

\[
\text{filter (all nodups . boxes) . expand . pruneBy boxes}
\]

We have shown that

\[
\text{filter (all nodups . boxes) . expand = filter (all nodups . boxes) . expand . pruneBy boxes}
\]

where \(\text{pruneBy f = f . map pruneRow . f}\). Repeating the same calculation for rows and columns, we obtain

\[
\text{filter valid . expand = filter valid . expand . prune}
\]

where

\[
\text{prune = pruneBy boxes . pruneBy cols . pruneBy rows}
\]
In conclusion, the previous definition of `solve` can now be replaced with a new one:

\[
\text{solve} = \text{filter valid} \ . \ \text{expand} \ . \ \text{prune} \ . \ \text{choices}
\]

In fact, rather than have just one `prune` we can have as many prunes as we like. This is sensible because after one round of pruning some choices may be resolved into singleton choices and another round of pruning may remove still more impossible choices.

So, let us define

\[
\text{many} \ :: \ (\text{Eq a}) \Rightarrow (a \rightarrow a) \rightarrow a \rightarrow a
\]

\[
\text{many } f \ x = \begin{cases} \text{x} & \text{if } \ x == \ y \\ \text{many } f \ y & \text{else} \end{cases}
\]

where \( y = f \ x \)

and redefine `solve` once again to read

\[
\text{solve} = \text{filter valid} \ . \ \text{expand} \ . \ \text{many prune} \ . \ \text{choices}
\]

The simplest Sudoku problems are solved just by repeatedly pruning the matrix of choices until only singleton choices are left.

### 5.4 Expanding a single cell

The result of `many prune . choices` is a matrix of choices that can be put into one of three classes:

1. A *complete* matrix in which every entry is a singleton choice. In this case `expand` will extract a single grid that can be checked for validity.
2. A matrix that contains the empty choice somewhere. In this case `expand` will produce the empty list.
3. A matrix that does not contain the empty choice but does contain some entry with two or more choices.

The problem is what to do in the third case. Rather than carry out full expansion, a more sensible idea is to make use of a partial expansion that installs the choices for just one of the entries, and to start the pruning process again on each result. The hope is that mixing pruning with single-cell expansions can lead to a solution more quickly. Our aim therefore is to construct a partial function

\[
\text{expand1} :: \text{Matrix [Digit]} \rightarrow \text{[Matrix [Digit]]}
\]
A simple Sudoku solver

that expands the choices for one cell only. This function will return well-defined results only for incomplete matrices, and on such matrices is required to satisfy

$$\text{expand} = \text{concat} \cdot \text{map expand} \cdot \text{expand1}$$

Actually this equality between two lists is too strong. We want to ensure that no possible choice is lost by partial expansion, but do not really care about the precise order in which the two sides deliver their results. So we will interpret the equation as asserting the equality of the two sides up to some permutation of the answers.

Which cell should we perform expansion on? The simplest answer is to find the first cell in the matrix with a non-singleton entry. Think of a matrix \(\text{rows}\) broken up as follows:

\[
\text{rows} = \text{rows1} ++ [\text{row}] ++ \text{rows2} \\
\text{row} = \text{row1} ++ [\text{cs}] ++ \text{row2}
\]

The cell \(\text{cs}\) is a non-singleton list of choices in the middle of \(\text{row}\), which in turn is in the middle of the matrix \(\text{rows}\).

Then we can define

\[
\text{expand1} :: \text{Matrix [Digit]} \rightarrow [\text{Matrix [Digit]}] \\
\text{expand1} \ \text{rows} \\
= [\text{rows1} ++ [\text{row1} ++ [c]:\text{row2}] ++ \text{rows2} | c < cs]
\]

To break up the matrix in this way, we use the prelude function \(\text{break}\):

\[
\text{break} :: (a -> \text{Bool}) -> [a] -> ([a],[a]) \\
\text{break} \ p = \text{span} \ (\text{not} . p)
\]

The function \(\text{span}\) was defined in Section 4.8. For example,

\[
\text{ghci}> \text{break} \ \text{even} \ [1,3,7,6,2,3,5] \\
([1,3,7],[6,2,3,5])
\]

We also need the standard prelude function \(\text{any}\), defined by

\[
\text{any} :: (a -> \text{Bool}) -> [a] -> \text{Bool} \\
\text{any} \ p = \text{or} . \text{map} \ p
\]

where \(\text{or}\) takes a list of booleans and returns \(\text{True}\) if any element is \(\text{True}\), and \(\text{False}\) otherwise:

\[
\text{or} :: [\text{Bool}] -> \text{Bool} \\
\text{or} [] = \text{False} \\
\text{or} (x:xs) = x \mid\mid \text{or} \ xs
\]
Finally, the single test is defined (using don’t care patterns) by

\[
\text{single} :: [a] \rightarrow \text{Bool} \\
\text{single } [\_] = \text{True} \\
\text{single } _ = \text{False}
\]

Now we can define

\[
\text{expand1} :: \text{Matrix } [\text{Digit}] \rightarrow [\text{Matrix } [\text{Digit}]] \\
\text{expand1 } \text{rows} \\
= [\text{rows1 } ++ [\text{row1 } ++ [c]:\text{row2}] ++ \text{rows2 } | \ c <- \text{cs}] \\
\hspace{1cm} \text{where} \\
(\text{rows1},\text{row}:\text{rows2}) = \text{break } (\text{any } (\text{not . single})) \text{ rows} \\
(\text{row1},\text{cs}:\text{row2}) = \text{break } (\text{not . single}) \text{ row}
\]

The first where clause breaks a matrix into two lists of rows with the row at the head of the second list being one that contains a non-singleton choice. A second appeal to break then breaks this row into two lists, with the head of the second list being the first non-singleton element. If the matrix contains only singleton entries, then

\[
\text{break } (\text{any } (\text{not . single})) \text{ rows} = [\text{rows},[\_]]
\]

and execution of expand1 returns an error message.

The problem with this definition of expand1 is that it can lead to wasted work. If the first non-singleton entry found in this way happens to be the empty list, then expand1 will return the empty list, but if such a list is buried deep in the matrix, then expand1 will do a lot of useless calculation trying to find a solution that isn’t there. It is arguable that a better choice of cell on which to perform expansion is one with the smallest number of choices (not equal to 1 of course). A cell with no choices means that the puzzle is unsolvable, so identifying such a cell quickly is a good idea.

The change to expand1 to implement this idea is as follows:

\[
\text{expand1} :: \text{Matrix } [\text{Digit}] \rightarrow [\text{Matrix } [\text{Digit}]] \\
\text{expand1 } \text{rows} \\
= [\text{rows1 } ++ [\text{row1 } ++ [c]:\text{row2}] ++ \text{rows2 } | \ c <- \text{cs}] \\
\hspace{1cm} \text{where} \\
(\text{rows1},\text{row}:\text{rows2}) = \text{break } (\text{any } \text{smallest}) \text{ rows} \\
(\text{row1},\text{cs}:\text{row2}) = \text{break } \text{smallest row} \\
\text{smallest } \text{cs} = \text{length } \text{cs} = \text{n} \\
\text{n} = \text{minimum } (\text{counts } \text{rows})
\]
The function \texttt{counts} is defined by
\[
\texttt{counts} = \text{filter} (\neq 1) . \text{map} \ \text{length} . \ \text{concat}
\]
The value \( n \) is the smallest number of choices, not equal to 1, in any cell of the matrix of choices. We will leave the definition of \texttt{minimum} as an exercise. The value of \( n \) will be 0 if the matrix has an empty choice entry anywhere, and in this case \texttt{expand1} will return the empty list. On the other hand, if the matrix of choices contains only singleton choices, then \( n \) is the minimum of the empty list, which is the undefined value \( \perp \). In this case \texttt{expand1} will also return \( \perp \), so we had better ensure that \texttt{expand1} is applied only to incomplete matrices. A matrix is incomplete if it does not satisfy \texttt{complete}:
\[
\texttt{complete} :: \text{Matrix} [\text{Digit}] \rightarrow \text{Bool} \\
\texttt{complete} = \text{all} \ (\text{all single})
\]
We can also usefully generalise \texttt{valid} to a test on matrices of choices. Suppose we define \texttt{safe} by
\[
\texttt{safe} :: \text{Matrix} [\text{Digit}] \rightarrow \text{Bool} \\
\texttt{safe} \ m = \text{all ok (rows cm)} \ \&\& \\
\quad \text{all ok (cols cm)} \ \&\& \\
\quad \text{all ok (boxes cm)} \\
\quad \text{ok row} = \text{nodups [x | [x] <- row]}
\]
A matrix is safe if none of the singleton choices in any row, column or box contain duplicates. But a safe matrix may contain non-singleton choices. Pruning can turn a safe matrix into an unsafe one, but if a matrix is safe after pruning it has to be safe beforehand. In symbols, \texttt{safe . prune} = \texttt{safe}. A complete and safe matrix yields a solution to the Sudoku problem, and this solution can be extracted by a simplified version of \texttt{expand}:
\[
\texttt{extract} :: \text{Matrix} [\text{Digit}] \rightarrow \text{Grid} \\
\texttt{extract} = \text{map} \ (\text{map head})
\]
Hence on a safe and complete matrix \( m \) we have
\[
\texttt{filter valid} (\texttt{expand} \ m) = [\texttt{extract} \ m]
\]
On a safe but incomplete matrix we have
\[
\texttt{filter valid} \ . \ \texttt{expand} \\
= \texttt{filter valid} \ . \ \texttt{concat} \ . \ \texttt{map expand} \ . \ \texttt{expand1}
\]
up to permutation of each side. Since
filter p . concat = concat . map (filter p)

we obtain that filter valid . expand simplifies to

concat . map (filter p . expand) . expand1

And now we can insert a single prune to obtain

concat . map (filter p . expand . prune) . expand1

Hence, introducing

search = filter valid . expand . prune

we have, on safe but incomplete matrices, that

search = concat . map search . expand1 . prune

And now we can replace solve by a third version:

solve = search . choices

search cm
  | not (safe pm) = []
  | complete pm    = [extract pm]
  | otherwise      = concat (map search (expand1 pm))

where pm = prune cm

This is our final simple Sudoku solver. We could replace prune in the last line by many prune. Sometimes many prunes work faster than one prune; sometimes not. Note that the very first safety test occurs immediately after one round of pruning on the installed choices; consequently flawed input is detected quickly.

5.5 Exercises

Exercise A

How would you add 1 to every element in a given matrix of integers? How would you sum the elements of a matrix? The function zipWith (+) adds two rows, but what function would add two matrices? How would you define matrix multiplication?

Exercise B

What are the dimensions of the matrix [[], []]? Of the matrix []? The function cols (here renamed as transpose) was defined by
transpose :: [[a]] -> [[a]]
transpose [xs] = [[x] | x <- xs]
transpose (xs:xss) = zipWith (:) xs (transpose xss)

Fill in the dots that would enable you to replace the first clause by

\[
\text{transpose \ } [] \quad = \ldots
\]

The above definition of \text{transpose} proceeds row by row. Here is part of a definition that proceeds column by column:

\[
\text{transpose \ } xss = \text{map head \ } xss:\text{transpose \ } (\text{map tail \ } xss)
\]

Complete this definition.

Exercise C

Which of the following equations are true (no justification is necessary):

\[
\begin{align*}
\text{any } p & = \text{not . all (not } p) \\
\text{any } \text{null} & = \text{null . } \text{cp}
\end{align*}
\]

Exercise D

Given a function \text{sort} :: (Ord a) => [a] -> [a] that sorts a list, construct a definition of

\[
\text{nodups} :: (Ord a) => [a] -> \text{Bool}
\]

Exercise E

The function \text{nub} :: (Eq a) => [a] -> [a] removes duplicates from a list (a version of this function is available in the library Data.List). Define \text{nub}. Assuming the order of the elements in the result is not important, define

\[
\text{nub} :: (\text{Ord a}) => [a] -> [a]
\]

so that the result is a more efficient function.

Exercise F

The functions \text{takeWhile} and \text{dropWhile} satisfy

\[
\text{span } p \ xs = (\text{takeWhile } p \ xs, \text{dropWhile } p \ xs)
\]

Give direct recursive definitions of \text{takeWhile} and \text{dropWhile}.

Assuming \text{whiteSpace} :: Char -> \text{Bool} is a test for whether a character is
white space (such as a space, a tab or a newline) or a visible character, construct a
definition of

words :: String -> [Word]

that breaks a string up into a list of words.

**Exercise G**

Define \( \text{minimum} :: \text{Ord } a \Rightarrow [a] \rightarrow a \).

**Exercise H**

Why didn’t we define \( \text{solve} \) by the following?

\[
\begin{align*}
\text{solve} &= \text{search} \cdot \text{choices} \\
\text{search } m &= \begin{cases} 
\text{[]} & \text{not (\text{safe } m)} \\
\text{[\text{extract } m]} & \text{complete } m \\
\text{process } m & \text{otherwise}
\end{cases} \\
\text{where process} &= \text{concat} \cdot \text{map search} \cdot \text{expand1} \cdot \text{prune}
\end{align*}
\]

**Answer to Exercise A**

Adding 1 to every matrix element is defined by \( \text{map } (\text{map } (+1)) \).

Summing a matrix is defined by \( \text{sum } \cdot \text{map } \text{sum} \), where \( \text{sum} \) sums a list of num-
bers. Alternatively, we could use \( \text{sum } \cdot \text{concat} \).

Matrix addition is defined by \( \text{zipWith } (\text{zipWith } (+)) \).

For matrix multiplication we first define

\[
\begin{align*}
\text{scalarMult} :: \text{Num } a \Rightarrow [a] \rightarrow [a] \rightarrow a \\
\text{scalarMult } xs \ ys &= \text{sum } (\text{zipwith } (*) \ xs \ ys)
\end{align*}
\]

Then we have

\[
\begin{align*}
\text{matMult} :: \text{Num } a \Rightarrow \text{Matrix } a \rightarrow \text{Matrix } a \rightarrow \text{Matrix } a \\
\text{matMult } ma \ mb &= \text{map } (\text{scalarMult } \text{row}) \ \text{mbt } | \ \text{row } <- \ \text{ma} \\
\text{where } \text{mbt} &= \text{transpose } \text{mb}
\end{align*}
\]
Answer to Exercise B

The matrix \([[], []]\) has dimensions \(2 \times 0\). The matrix \([[]]\) has dimensions \(0 \times n\) for every \(n\). The transpose of such a matrix therefore has to have dimensions \(n \times 0\) for every \(n\). The only reasonable possibility is to let \(n\) be infinite:

\[
\text{transpose} :: [[a]] \rightarrow [[a]] \\
\text{transpose} [] = \text{repeat} [] \\
\text{transpose} (xs:xss) = \text{zipWith} (:) xs (\text{transpose} xss)
\]

where \text{repeat} \(x\) gives an infinite list of repetitions of \(x\). Note that

\[
\text{transpose} [xs] = \text{zipWith} (:) xs (\text{repeat} []) \\
= [[x] | x <- xs]
\]

The alternative definition is

\[
\text{transpose} ([]:xss) = [] \\
\text{transpose} xss = \text{map} \ \text{head} \ xss: \text{transpose} (\text{map} \ \text{tail} \ xss)
\]

The assumption in the first line is that if the first row is empty, then all the rows are empty and the transpose is the empty matrix.

Answer to Exercise C

Both the equations are true.

Answer to Exercise D

\[
\text{nodups} :: (\text{Ord} a) \Rightarrow [a] \rightarrow \text{Bool} \\
\text{nodups} \ x = \text{and} (\text{zipWith} (\neq) \ ys (\text{tail} \ ys)) \\
\text{where} \ ys = \text{sort} \ x
\]

Answer to Exercise E

\[
\text{nub} :: (\text{Eq} a) \Rightarrow [a] \rightarrow [a] \\
\text{nub} [] = [] \\
\text{nub} (x:xs) = x: \text{nub} (\text{filter} (\neq x) \ x) \\
\text{nub} :: (\text{Ord} a) \Rightarrow [a] \rightarrow [a] \\
\text{nub} = \text{remdups} \ . \ \text{sort}
\]

\[
\text{remdups} [] = [] \\
\text{remdups} (x:xs) = x: \text{remdups} (\text{dropWhile} (==x) \ x) \\
\]

The function \text{dropWhile} is defined in the next exercise.
Answer to Exercise F

takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs) = if p x then x:takeWhile p xs else []
dropWhile p [] = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x:xs

The definition of words is

words :: String -> [Word]
words xs | null ys = []
         | otherwise = w:words zs
         where ys = dropWhile whiteSpace xs
                (w,zs) = break whiteSpace ys

Answer to Exercise G

minimum :: Ord a => [a] -> a
minimum [x] = x
minimum (x:xs) = x `min` minimum xs

Note that the minimum of the empty list is undefined.

Answer to Exercise H

The suggested definition of solve would return the undefined value if the matrix becomes complete after one round of pruning.

5.7 Chapter notes

The Independent newspaper no longer uses the rubric for Sudoku quoted at the start of the chapter. The presentation follows that in my book Pearls of Functional Algorithm Design (Cambridge, 2010). The site

haskell.org/haskellwiki/Sudoku

contains about 20 Haskell implementations of Sudoku, many of which use arrays and/or monads. We will meet arrays and monads in Chapter 10.
We have seen a lot of laws in the previous two chapters, though perhaps the word ‘law’ is a little inappropriate because it suggests something that is given to us from on high and which does not have to be proved. At least the word has the merit of being short. All of the laws we have encountered so far assert the equality of two functional expressions, possibly under subsidiary conditions; in other words, laws have been equations or identities between functions, and calculations have been point-free calculations (see Chapter 4, and the answer to Exercise K for more on the point-free style). Given suitable laws to work with, we can then use equational reasoning to prove other laws. Equational logic is a simple but powerful tool in functional programming because it can guide us to new and more efficient definitions of the functions and other values we have constructed. Efficiency is the subject of the following chapter. This one is about another aspect of equational reasoning, proof by induction. We will also show how to shorten proofs by introducing a number of higher-order functions that capture common patterns of computations. Instead of proving properties of similar functions over and over again, we can prove more general results about these higher-order functions, and appeal to them instead.

6.1 Induction over natural numbers

Consider the following definition of the exponential function:

\[
\begin{align*}
\text{exp} & : \text{Num} \ a \Rightarrow a \rightarrow \text{Nat} \rightarrow a \\
\text{exp} \ x \ \text{Zero} & = 1 \\
\text{exp} \ x \ (\text{Succ} \ n) & = x \ast \text{exp} \ x \ n
\end{align*}
\]

In the old days we could have written
6.1 Induction over natural numbers

exp :: Num a => a -> Int -> a
exp x 0 = 1
exp x (n+1) = x * exp x n

but this precise form of definition using a \( (n+1) \)-pattern is no longer allowed in the current standard version of Haskell, Haskell 2010.

Anyway, we would expect that the equation

\[
exp x (m+n) = exp x m * exp x n
\]

is true for all \( m \) and \( n \). After all, \( x^{m+n} = x^m x^n \) is a true equation of mathematics. But how can we prove this law?

The answer, of course, is by induction. Every natural number is either \texttt{Zero} or of the form \texttt{Succ } \( n \) for some natural number \( n \). That is exactly what the definition

\[
data Nat = \text{Zero} \mid \text{Succ } Nat
\]

of the data type \texttt{Nat} tells us. So to prove that \( P(n) \) holds for all natural numbers \( n \), we can prove

1. \( P(0) \) holds;
2. For all natural numbers \( n \), that \( P(n+1) \) holds assuming that \( P(n) \) does.

We have reverted to writing 0 for \texttt{Zero} and \( n+1 \) for \texttt{Succ } \( n \), and we shall continue to do so. In the second proof we can assume \( P(n) \) and use this assumption to prove \( P(n+1) \).

As an example we prove that

\[
exp x (m+n) = exp x m * exp x n
\]

for all \( x, m \) and \( n \) by induction on \( m \). We could also prove it by induction on \( n \) but that turns out to be more complicated. Here is the proof:

**Case 0**

\[
\begin{align*}
exp x (0 + n) & = \{ \text{since } 0 + n = n \} \exp x n \\
\exp x n & = \{ \text{exp.1} \} 1 * \exp x n \\
& = \{ \text{since } 1 * x = x \} \exp x n
\end{align*}
\]
Proofs

Case $m+1$

\[
\begin{align*}
\exp x ((m + 1) + n) &= \exp x (m+1) \ast \exp x n \\
= \{ \text{arithmetic} \} &= \{ \exp.2 \} \\
\exp x ((m + n) + 1) &= (x \ast \exp x m) \ast \exp x n \\
= \{ \exp.2 \} &= \{ \text{since } \ast \text{ is associative} \} \\
x \ast \exp x (m + n) &= x \ast (\exp x m \ast \exp x n) \\
= \{ \text{induction} \} &= x \ast (\exp x m \ast \exp x n)
\end{align*}
\]

The above format will be used in all induction proofs. The proof breaks into two cases, the base case $0$ and the inductive case $n + 1$. Each case is laid out in two columns, one for the left-hand side of the equation, and one for the right-hand side. (When there is not enough space for two columns, we display one after the other.) Each side is simplified until one can go no further, and the proof of each case is completed by observing that each side simplifies to the same result. The hints $\exp.1$ and $\exp.2$ refer to the first and second equations defining $\exp$.

Finally, observe that the proof depends on three further laws, namely that

\[
\begin{align*}
(m + 1) + n &= (m + n) + 1 \\
1 \ast x &= x \\
(x \ast y) \ast z &= x \ast (y \ast z)
\end{align*}
\]

If we were recreating all of arithmetic from scratch – and that would be a tedious thing to do – we would also have to prove these laws. In fact, only the first can be proved because it is entirely about natural numbers and we have defined the operation of addition on natural numbers. The second two rely on the implementation of multiplication prescribed by Haskell for the various instances of the type class Num.

In fact, the associative law breaks down for floating-point numbers:

\[
\begin{align*}
\text{ghci}> (9.9e10 \ast 0.5e-10) \ast 0.1e-10 :: \text{Float} \\
4.95e-11 \\
\text{ghci}> 9.9e10 \ast (0.5e-10 \ast 0.1e-10) :: \text{Float} \\
4.94999998e-11
\end{align*}
\]

Recall that in scientific notation $9.9e10$ means $9.9 \ast 10^{-10}$. So, although our proof was correct mathematically, one of the provisos in it wasn’t, at least in Haskell.
6.2 Induction over lists

We have seen that every finite list is either the empty list \([\ ]\) or of the form \(x:xs\) where \(xs\) is a finite list. Hence, to prove that \(P(xs)\) holds for all finite lists \(xs\), we can prove:

1. \(P([])\) holds;
2. For all \(x\) and for all finite lists \(xs\), that \(P(x:xs)\) holds assuming \(P(xs)\) does.

As an example, recall the definition of concatenation (++):

\[
\begin{align*}
[] ++ ys & = ys \\
(x:xs) ++ ys & = x : (xs ++ ys)
\end{align*}
\]

We prove that ++ is associative:

\[
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
\]

for all finite lists \(xs\) and for all lists \(ys\) and \(zs\) (note that neither of the last two is required to be a finite list), by induction on \(xs\):

**Case []**

\[

\begin{align*}
([] ++ ys) ++ zs & = \{++ .1\} ys ++ zs \\
[] ++ (ys ++ zs) & = \{++ .1\} ys ++ zs
\end{align*}
\]

**Case x:xs**

\[

\begin{align*}
((x:xs) ++ ys) ++ zs & = \{++ .2\} x:(xs ++ (ys ++ zs)) \\
x:(xs ++ (ys ++ zs)) & = \{induction\} x:((xs ++ ys) ++ zs)
\end{align*}
\]

As another example, given the definition

\[
\begin{align*}
\text{reverse } [] & = [] \\
\text{reverse } (x:xs) & = \text{reverse } xs ++ [x]
\end{align*}
\]

We prove that reverse is an involution:

\[
\text{reverse } (\text{reverse } xs) = xs
\]

for all finite lists \(xs\). The base case is easy and the inductive case proceeds:
Case $x:xs$

\[
\begin{align*}
\text{reverse } (\text{reverse } (x:xs)) &= \{\text{reverse.2}\} \\
\text{reverse } (\text{reverse } xs ++ [x]) &= \{??\} \\
x:\text{reverse } (\text{reverse } xs) &= \{\text{induction}\} \\
x:xs
\end{align*}
\]

The right-hand column is omitted in this example, since it consists solely of $x:xs$. But we got stuck in the proof halfway through. We need an auxiliary result, namely that

\[
\text{reverse } (ys ++ [x]) = x:\text{reverse } ys
\]

for all finite lists $ys$. This auxiliary result is also proved by induction:

Case $[]$

\[
\begin{align*}
\text{reverse } ([] ++ [x]) &= \{++1\} \\
\text{reverse } [x] &= \{\text{reverse.1}\} \\
\text{reverse } [] ++ [x] &= \{\text{reverse.1 and ++1}\} \\
[x]
\end{align*}
\]

Case $y:ys$

\[
\begin{align*}
\text{reverse } ((y:ys) ++ [x]) &= \{++2\} \\
\text{reverse } (y:(ys ++ [x])) &= \{\text{reverse.2}\} \\
\text{reverse } (ys ++ [x]) ++ [y] &= \{\text{induction}\} \\
(x:\text{reverse } ys) ++ [y] &= \{++2\} \\
x:(\text{reverse } ys ++ [y])
\end{align*}
\]
The auxiliary result holds, and therefore so does the main result.

**Induction over partial lists**

Every partial list is either the undefined list or of the form \(x:xs\) for some \(x\) and some partial list \(xs\). Hence, to prove that \(P(xs)\) holds for all partial lists \(xs\) we can prove that

1. \(P(\text{undefined})\) holds;
2. \(P(x:xs)\) holds assuming \(P(xs)\) does, for all \(x\) and all partial lists \(xs\).

As an example, we prove that

\[xs ++ ys = xs\]

for all partial lists \(xs\) and all lists \(ys\):

**Case undefined**

\[
\text{undefined} ++ ys = \{++.0\} \text{undefined}
\]

**Case \(x:xs\)**

\[
(x:xs) ++ ys = \{++.2\} \ x:(xs ++ ys) = \{\text{induction}\} \ x:xs
\]

In each case the trivial right-hand column is omitted. The hint \((++).0\) refers to the failing clause in the definition of \((++): since concatenation is defined by pattern matching on the left-hand argument, the result is undefined if the left-hand argument is.

**Induction over infinite lists**

Proving that something is true of all infinite lists requires a bit of background that we will elaborate on in a subsequent chapter. Basically an infinite list can
be thought of as the limit of a sequence of partial lists. For example, \([0..]\) is the limit of the sequence
\[
\text{undefined, 0:undefined, 0:1:undefined, 0:1:2:undefined,}
\]
and so on. A property \(P\) is called \textit{chain complete} if whenever \(xs_0, xs_1, \ldots\) is a sequence of partial lists with limit \(xs\), and \(P(xs_n)\) holds for all \(n\), then \(P(xs)\) also holds.

In other words, if \(P\) is a chain complete property that holds for all partial lists (and possibly all finite lists too), then it holds for all infinite lists.

Many properties are chain complete; for instance:

- All equations \(e_1 = e_2\), where \(e_1\) and \(e_2\) are Haskell expressions involving universally quantified free variables, are chain complete.
- If \(P\) and \(Q\) are chain complete, then so is their conjunction \(P \land Q\).

But inequalities \(e_1 \neq e_2\) are not necessarily chain complete, and neither are properties involving existential quantification. For example, consider the assertion
\[
\text{drop n xs = undefined}
\]
for some integer \(n\). This property is obviously true for all partial lists, and equally obviously not true for any infinite list.

Here is an example proof. Earlier we proved that
\[
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
\]
for all finite lists \(xs\) and for all lists \(ys\) and \(zs\). We can extend this chain complete property to \textit{all} lists \(xs\) by proving

**Case undefined**
\[
\begin{align*}
(\text{undefined ++ ys}) & \text{ ++ zs} & \text{undefined ++ (ys ++ zs)} \\
= \{++.0\} & = \{++.0\} \\
\text{undefined ++ zs} & \text{ undefined} \\
= \{++.0\} & \\
\text{undefined}
\end{align*}
\]
Thus ++ is a truly associative operation on lists, independent of whether the lists are finite, partial or infinite.

But we have to be careful. Earlier we proved
reverse (reverse xs) = xs

for all finite lists xs. Can we extend this property to all lists by proving the following additional case?

Case undefined

reverse (reverse undefined) = \{reverse.0\} undefined

That goes through but something is wrong: as a Haskell equation we have

reverse (reverse xs) = undefined

for all partial lists xs. What did we miss?

The answer is that in proving the involution property of reverse we made use of an auxiliary result:

reverse (ys ++ [x]) = x:reverse ys

for all finite lists ys. This result is not true for all lists, indeed not true for any partial list ys.

It follows that reverse . reverse is not the identity function on lists. A functional equation \( f = g \) over lists asserts that \( f \ x s = g \ x s \) for all lists xs, finite, partial and infinite. If the equation is true only for finite lists, we have to say so explicitly.

6.3 The function foldr

All the following functions have a common pattern:

\[
\begin{align*}
\text{sum} \ [\ ] &= 0 \\
\text{sum} \ (x:xs) &= x + \text{sum} \ xs \\
\text{concat} \ [\ ] &= [] \\
\text{concat} \ (xs:xss) &= xs ++ \text{concat} \ xss \\
\text{filter} \ p \ [\ ] &= [] \\
\text{filter} \ p \ (x:xs) &= \text{if} \ p \ x \ \text{then} \ x: \text{filter} \ p \ xs \\
&\quad \text{else} \ \text{filter} \ p \ xs
\end{align*}
\]
map f [] = []
map f (x:xs) = f x:map f xs

Similarly, the proofs by induction of the following laws all have a common pattern:

sum (xs ++ ys) = sum xs + sum ys
concat (xss ++ yss) = concat xss ++ concat yss
filter p (xs ++ ys) = filter p xs ++ filter p ys
map f (xs ++ ys) = map f xs ++ map f ys

Can we not ensure that the functions above are defined as instances of a more general function, and the laws above as instances of a more general law? That would save a lot of repetitive effort.

The function foldr (fold from the right) is defined by

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f e [] = e
foldr f e (x:xs) = f x (foldr f e xs)

To appreciate this definition, consider

foldr (@) e [x,y,z] = x @ (y @ (z @ e))
[x,y,z] = x : (y : (z : []))

In other words, foldr (@) e applied to a list replaces the empty list by e, and (:) by (@) and evaluates the result. The parentheses group from the right, whence the name.

It follows at once that foldr (:) [] is the identity function on lists. Furthermore,

sum = foldr (+) 0
concat = foldr (++) []
filter p = foldr (\x xs -> if p x then x:xs else xs) []
map f = foldr ((:) . f) []

The following fact captures all the identities mentioned above:

foldr f e (xs ++ ys) = foldr f e xs @ foldr f e ys

for some operation (@) satisfying various properties. We prove this equation by induction on xs. Along the way, we discover what properties of f, e and (@) we need.
6.3 The function \texttt{foldr}

**Case \(\texttt{[]}\)**

\[
\texttt{foldr} \ f \ e \ (\texttt{[]} ++ \texttt{ys}) = \{++1\} \texttt{foldr} \ f \ e \ \texttt{[]} @ \texttt{foldr} \ f \ e \ \texttt{ys}
\]

Hence we need \(e @ x = x\) for all \(x\).

**Case \(x:xs\)**

\[
\texttt{foldr} \ f \ e \ ((x:xs) ++ \texttt{ys}) = \{++2\}
\]

\[
\texttt{foldr} \ f \ e \ (x:(xs ++ \texttt{ys})) = \{\texttt{foldr}2\}
\]

\[
f \ x \ (\texttt{foldr} \ f \ e \ (xs ++ \texttt{ys})) = \{\texttt{induction}\}
\]

\[
f \ x \ (\texttt{foldr} \ f \ e \ xs @ \texttt{foldr} \ f \ e \ \texttt{ys})
\]

The right-hand side in this case simplifies to

\[
f \ x \ (\texttt{foldr} \ f \ e \ xs) @ \texttt{foldr} \ f \ e \ \texttt{ys}
\]

So, in summary, we require that

\[
e @ x = x
\]

\[
f \ x \ (y @ z) = f \ x \ y @ z
\]

for all \(x, y\) and \(z\). In particular the two requirements are met if \(f = (@)\) and \((@)\) is associative with identity \(e\). That immediately proves

\[
\text{sum} \ (xs ++ \texttt{ys}) = \text{sum} \ xs + \text{sum} \ \texttt{ys}
\]

\[
\text{concat} \ (xss ++ yss) = \text{concat} \ xss ++ \text{concat} \ yss
\]

For the map law, we require that

\[
\texttt{[]} \ ++ \texttt{xs} = \texttt{xs}
\]

\[
f \ x:(xs ++ \texttt{ys}) = (f \ x:ys) ++ \texttt{ys}
\]

Both immediately follow from the definition of concatenation.

For the law of \texttt{filter} we require that

\[
\text{if} \ p \ x \ \text{then} \ x:(ys ++ zs) \ \text{else} \ ys ++ zs = \text{if} \ p \ x \ \text{then} \ x:ys \ \text{else} \ ys +\ + zs
\]
This is immediate from the definitions of concatenation and conditional expressions.

**Fusion**

The most important property of `foldr` is the *fusion law*, which asserts that

\[ f \cdot \text{foldr} \, g \, a = \text{foldr} \, h \, b \]

provided certain properties of the ingredients hold. As two simple examples,

\[ \text{double} \cdot \text{sum} = \text{foldr} \, ((+) \cdot \text{double}) \, 0 \]
\[ \text{length} \cdot \text{concat} = \text{foldr} \, ((+) \cdot \text{length}) \, 0 \]

In fact, many of the laws we have seen already are instances of the fusion law for `foldr`. In a word, the fusion law is a ‘pre-packaged’ form of induction over lists.

To find out what properties we need, we carry out an induction proof of the fusion law. The law is expressed as a functional equation, so we have to show that it holds for all finite and all partial lists:

**Case** `undefined`

\[ f \, (\text{foldr} \, g \, a \, \text{undefined}) = f \, \text{undefined} \]
\[ = \{\text{foldr.0}\} \quad \text{foldr} \, h \, b \, \text{undefined} \]
\[ = f \, \text{undefined} \quad = \{\text{foldr.0}\} \]

So the first condition is that `f` is a strict function.

**Case** `[]`

\[ f \, (\text{foldr} \, g \, a \, []) = f \, a \]
\[ = \{\text{foldr.1}\} \quad \text{foldr} \, h \, b \, [] = \{\text{foldr.1}\} \]
\[ = b \]

The second condition is that `f a = b`.

**Case** `x:xs`

\[ f \, (\text{foldr} \, g \, a \, (x:xs)) = h \, x \, (\text{foldr} \, h \, b \, (x:xs)) \]
\[ = \{\text{foldr.2}\} \quad = \{\text{foldr.2}\} \]
\[ f \, (g \, x \, (\text{foldr} \, g \, a \, xs)) = h \, x \, (\text{foldr} \, h \, b \, xs) \]
\[ = \{\text{induction}\} \quad = \{\text{induction}\} \]
\[ = h \, x \, (f \, (\text{foldr} \, g \, a \, xs)) \]
The third condition is met by \( f(x \cdot y) = h \cdot x \cdot (f \cdot y) \) for all \( x \) and \( y \).

Let us apply the fusion law to show that

\[
\text{foldr} \ f \ a \ . \ \text{map} \ g = \text{foldr} \ h \ a
\]

Recall that \( \text{map} \ g = \text{foldr} \ ((:) \ . \ g) \ [] \). Looking at the conditions of the fusion law we have that

\[
\text{foldr} \ f \ a \ \text{undefined} = \text{undefined} \\
\text{foldr} \ f \ a \ [] = a
\]

So the first two fusion conditions are satisfied. The third one is

\[
\text{foldr} \ f \ a \ (g \ x:xs) = h \cdot x \cdot (\text{foldr} \ f \ a \ xs)
\]

The left-hand side simplifies to

\[
f(x \cdot (\text{foldr} \ f \ a \ xs))
\]

so we can define \( h \cdot x \cdot y = f(x \cdot y) \). More briefly, \( h = f . g \). Hence we have the useful rule:

\[
\text{foldr} \ f \ a \ . \ \text{map} \ g = \text{foldr} \ (f \ . \ g) \ a
\]

In particular,

\[
\begin{align*}
\text{double} \ . \ \text{sum} &= \text{sum} \ . \ \text{map} \ \text{double} \\
&= \text{foldr} \ ((+) \ . \ \text{double}) \ 0
\end{align*}
\]

\[
\begin{align*}
\text{length} \ . \ \text{concat} &= \text{sum} \ . \ \text{map} \ \text{length} \\
&= \text{foldr} \ ((+) \ . \ \text{length}) \ 0
\end{align*}
\]

Other simple consequences of the fusion law are explored in the exercises.

_A variant_

Sometimes having the empty list around is a pain. For example, what is the minimum element in an empty list? For this reason, Haskell provides a variant on \text{foldr}, called \text{foldr1}, restricted to nonempty lists. The Haskell definition of this function is

\[
\begin{align*}
\text{foldr1} &: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a \\
\text{foldr1} \ f \ [x] &= x \\
\text{foldr1} \ f \ (x:xs) &= f \ x \ (\text{foldr1} \ f \ xs)
\end{align*}
\]
So we can define

\[
\text{minimum, maximum :: Ord } a \Rightarrow [a] \rightarrow a
\]

\[
\text{minimum = foldr1 min}
\]

\[
\text{maximum = foldr1 max}
\]

and avoid two other explicit recursions. Actually the Haskell definition of foldr1 is not as general as it should be, but we will leave that discussion to an exercise.

6.4 The function foldl

Recall that

\[
\text{foldr } (@) \varepsilon \ [w, x, y, z] = w @ (x @ (y @ (z @ e)))
\]

Sometimes a more convenient pattern for the right-hand side is

\[
(((\varepsilon @ w) @ x) @ y) @ z
\]

This pattern is encapsulated by a function foldl (fold from the left):

\[
\text{foldl} :: (b -> a -> b) -> b -> [a] -> b
\]

\[
\text{foldl } f \varepsilon \ [] = \varepsilon
\]

\[
\text{foldl } f \varepsilon \ (x:xs) = \text{foldl } f (f \varepsilon x) xs
\]

As an example, suppose we are given a string, such as 1234.567, representing a real number and we want to compute its integer part and fractional part. We could define

\[
\text{ipart :: String } \rightarrow \text{ Integer}
\]

\[
\text{ipart } xs = \text{read } (\text{takeWhile } (/= \ '_.') \ xs) \ : : \text{ Integer}
\]

\[
\text{fpart :: String } \rightarrow \text{ Float}
\]

\[
\text{fpart } xs = \text{read } ('0':\text{dropWhile } (/= \ '_.') \ xs) \ : : \text{ Float}
\]

This uses the function read of the type class Read. Note by the way that .567 is not a well-formed literal in Haskell. It is necessary to include at least one digit before and after the decimal point to ensure that the decimal point cannot be mistaken for functional composition. For example,

ghci> :t 3 . 4
3 . 4 :: (Num (b -> c), Num (a -> b)) => a -> c

As an alternative, we can define
6.4 The function foldl

```
parts :: String -> (Integer,Float)
parts ds = (ipart es,fpart fs)
  where (es,d:fs) = break (== '.') ds
ipart = foldl shiftl 0 . map toDigit
  where shiftl n d = n*10 + d
fpart = foldr shiftr 0 . map toDigit
  where shiftr d x = (d + x)/10
toInt d = fromIntegral (fromEnum d - fromEnum '0')
```

We have

\[
1234 = 1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \\
= (((0 \times 10 + 1) \times 10 + 2) \times 10 + 3) \times 10 + 4
\]

\[
0.567 = 5/10 + 6/100 + 7/1000 \\
= (5 + (6 + (7 + 0)/10)/10)/10
\]

so use of foldl for the integer part and foldr for the fractional part are both indicated.

Here is another example. The function reverse was defined above by the equations

\[
\text{reverse} \; [] = [] \\
\text{reverse} \; (x:xs) = \text{reverse} \; xs ++ [x]
\]

We are wiser now and would now write

\[
\text{reverse} = \text{foldr} \; \text{snoc} \; [] \\
\text{where} \; \text{snoc} \; x \; xs = xs ++ [x]
\]

But a little learning is a dangerous thing: both definitions of reverse are terrible because they take of the order of \(n^2\) steps to reverse a list of length \(n\). Much better is to define

\[
\text{reverse} = \text{foldl} \; (\text{flip} \; (:)) \; []
\]

where \(\text{flip} \; f \; x \; y = f \; y \; x\). The new version reverses a list in linear time:

\[
\text{foldl} \; (\text{flip} \; (:)) \; [] \; [1,2,3] \\
= \text{foldl} \; (\text{flip} \; (:)) \; (1:[])) \; [2,3] \\
= \text{foldl} \; (\text{flip} \; (:)) \; (2:1:[])) \; [3] \\
= \text{foldl} \; (\text{flip} \; (:)) \; (3:2:1:[])) \; [] \\
= 3:2:1:[]
\]

That seems a bit of a trick, but there is a sound principle at work behind this new definition that we will take up in the following chapter.
As this example suggests, there are the following relationships between `foldr` and `foldl`: for all finite lists `xs` we have

\[
\text{foldl}\ f\ e\ xs = \text{foldr}\ (\text{flip}\ f)\ e\ (\text{reverse}\ xs)
\]

\[
\text{foldr}\ f\ e\ xs = \text{foldl}\ (\text{flip}\ f)\ e\ (\text{reverse}\ xs)
\]

Proofs are left as an exercise. Note the restriction to finite lists, even though both sides reduce to `⊥` when `xs` is `⊥`. That means the proofs have to rely on a subsidiary result that is true only for finite lists.

Here is another relationship between the two folds:

\[
\text{foldl}\ (\@)\ e\ xs = \text{foldr}\ (\&\&)\ e\ xs
\]

for all finite lists `xs`, provided that

\[
(x \&\& y) @ z = x \&\& (y @ z)
\]

\[
e @ x = x \&\& e
\]

Again, the proof is left as an exercise. As one instructive application of this law, suppose `(\&\&)` = `(\@)` and `(\@)` is associative with identity `e`. Then the two provisos are satisfied and we can conclude that

\[
\text{foldr}\ (\@)\ e\ xs = \text{foldl}\ (\@)\ e\ xs
\]

for all finite lists `xs` whenever `(\@)` is associative with identity `e`. In particular,

\[
\text{concat}\ xss = \text{foldr}\ (\&\&)\ []\ xss = \text{foldl}\ (\&\&)\ []\ xss
\]

for all finite lists `xss`. The two definitions are not the same if `xss` is an infinite list:

\[
\text{ghci}> \text{foldl}\ (\&\&)\ []\ [[i]\mid i<-\[1..]]
\]

Interrupted.

\[
\text{ghci}> \text{foldr}\ (\&\&)\ []\ [[i]\mid i<-\[1..]]
\]

\[1,2,3,4,\{\text{Interrupted}\}\]

In response to the first expression, GHCi went into a long silence that was interrupted by pressing the ‘Stop program execution’ button. In response to the second, GHCi started printing an infinite list.

OK, so the definition in terms of `foldr` works on infinite lists, but the other one doesn’t. But maybe the definition of `concat` in terms of `foldl` leads to a more efficient computation when all the lists are finite? To answer this question, observe that

\[
\text{foldr}\ (\&\&)\ []\ [xs,ys,us,vs]
\]

\[
= xs \&\& (ys \&\& (us \&\& (vs \&\& [])))
\]
6.5 The function \texttt{scanl}

The function \texttt{scanl \_\_} \texttt{e} applies \texttt{foldl \_\_} \texttt{e} to each initial segment of a list. For example

\begin{verbatim}
ghci> scanl (+) 0 [1..10]
[0,1,3,6,10,15,21,28,36,45,55]
\end{verbatim}

The expression computes the \textit{running sums} of the first ten positive numbers:

\[0, \ 0+1, \ (0+1)+2, \ ((0+1)+2)+3, \ (((0+1)+2)+3)+4, \ldots \]

The specification of \texttt{scanl} is

\begin{verbatim}
scanl (b -> a -> b) -> b -> [a] -> [b]
scanl \_\_ e = map (foldl \_\_ e) . inits

inits :: [a] -> [[a]]
inits [] = [[]]
inits (x:xs) = [] : map (x:) (inits xs)
\end{verbatim}

For example

\begin{verbatim}
ghci> inits "barbara"
["","b","ba","bar","barb","barba","barbar","barbara"]
\end{verbatim}

The function \texttt{inits} is in the library \texttt{Data.List}.

But this definition of \texttt{scanl \_\_} \texttt{e} involves evaluating \texttt{e} a total of

\[0 + 1 + 2 + \cdots + n = n(n+1)/2\]

times on a list of length \(n\). Can we do better?

Yes, we can calculate a better definition by doing a kind of induction proof, except that we don’t know what it is we are proving!
Proofs

Case \[
\]

\[
\begin{align*}
\text{scanl } f & \ e \ [] \\
& = \ \{\text{definition}\} \\
& \quad \text{map} \ (\text{foldl } f \ e) \ (\text{inits } []) \\
& = \ \{\text{inits.1}\} \\
& \quad \text{map} \ (\text{foldl } f \ e) \ [[]] \\
& = \ \{\text{map.1 and map.2}\} \\
& \quad [\text{foldl } f \ e] \\
& = \ \{\text{foldl.1}\} \\
& \quad [e]
\end{align*}
\]

Hence we have shown that \(\text{scanl } f \ e \ [] = [e]\)

Case \text{x:xs}

\[
\begin{align*}
\text{scanl } f & \ e \ (\text{x:xs}) \\
& = \ \{\text{definition}\} \\
& \quad \text{map} \ (\text{foldl } f \ e) \ (\text{inits } (\text{x:xs})) \\
& = \ \{\text{inits.2}\} \\
& \quad \text{map} \ (\text{foldl } f \ e) \ [[]:\text{map} \ (\text{x:}) \ (\text{inits } \text{xs})) \\
& = \ \{\text{map.1 and map.2}\} \\
& \quad \text{foldl } f \ e \ []:\text{map} \ (\text{foldl } f \ e \ . \ (\text{x:})) \ (\text{inits } \text{xs}) \\
& = \ \{\text{foldl.1}\} \\
& \quad \text{e:map} \ (\text{foldl } f \ e \ . \ (\text{x:})) \ (\text{inits } \text{xs}) \\
& = \ \{\text{claim: foldl } f \ e \ . \ (\text{x:}) = \text{foldl } f \ (\text{f e x})\} \\
& \quad \text{e:map} \ (\text{foldl } f \ (\text{f e x})) \ (\text{inits } \text{xs}) \\
& = \ \{\text{definition of scanl}\} \\
& \quad \text{e:scanl } f \ (\text{f e x})
\end{align*}
\]

The claim is an easy consequence of the definition of foldl. Hence, in summary, we have shown

\[
\begin{align*}
\text{scanl } f & \ e \ [] \quad = \ [e] \\
\text{scanl } f & \ e \ (\text{x:xs}) \quad = \ \text{e:scanl } f \ (\text{f e x}) \ \text{xs}
\end{align*}
\]

This definition evaluates \(f\) only a linear number of times.
What we have just done is an example of optimising a function by program calculation. One of the exciting things about Haskell is that you can do this without fuss. There is no need to bring in a totally different logical language to reason about programs.

However, the prelude definition of \( \text{scanl} \) is a little different:

\[
\text{scanl} \ f \ e \ \text{xs} = e : (\text{case} \ \text{xs} \ \text{of} \\
\quad [] \ -> \ [] \\
\quad x:xs -> \text{scanl} \ f \ (f \ e \ x) \ xs)
\]

Whereas for our version \( \text{scanl} \ f \ e \ \text{undefined} = \text{undefined} \), the prelude version has

\[
\text{scanl} \ f \ e \ \text{undefined} = e : \text{undefined}.
\]

The reason is that the right-hand sides of the two clauses defining \( \text{scanl} \) are both lists that begin with \( e \). We do not have to know anything about the left-hand sides to determine this fact, and laziness dictates that we don’t ask.

The prelude version also uses a case expression. We won’t go into details since such expressions are used rarely in this book. Haskell allows us many ways to say the same thing.

### 6.6 The maximum segment sum

Here is another example of program calculation. The maximum segment sum problem is a famous one and its history is described in J. Bentley’s *Programming Pearls* (1987). Given is a sequence of integers and it is required to compute the maximum of the sums of all segments in the sequence. A segment is also called a contiguous subsequence. For example, the sequence

\([-1,2,-3,5,-2,1,3,-2,-2,-3,6]\)

has maximum sum 7, the sum of the segment \([5,-2,1,3]\). On the other hand, the sequence \([-1,-2,-3]\) has a maximum segment sum of zero, since the empty sequence is a segment of every list and its sum is zero. It follows that the maximum segment sum is always nonnegative.

Our problem is specified by

\[
\text{mss} :: \ [\text{Int}] \rightarrow \text{Int} \\
\text{mss} = \text{maximum} \ . \ \text{map} \ \text{sum} \ . \ \text{segments}
\]
where \texttt{segments} returns a list of all segments of a list. This function can be defined in a number of ways, including

\[
\texttt{segments} = \texttt{concat . map \texttt{inits . tails}}
\]

where \texttt{tails} is dual to \texttt{inits} and returns all the tail segments of a list:

\[
\texttt{tails} :: [a] \rightarrow [[[a]]] \\
\texttt{tails} [\textit{\texttt{[]}}} = [[\text{\texttt{[]}}]
\texttt{tails} (\texttt{x:x:xs}) = (\texttt{x:xs}):\texttt{tails} \texttt{xs}
\]

The definition of \texttt{segments} describes the process of taking all the initial segments of all the tail segments. For example,

\texttt{ghci> segments "abc" ["","a","ab","abc","","b","bc","","c",""]}

The empty sequence appears four times in this list, once for every tail segment.

Direct evaluation of \texttt{mss} will take a number of steps proportional to \(n^3\) on a list of length \(n\). There are about \(n^2\) segments, and summing each of them will take \(n\) steps, so in total it will take \(n^3\) steps. It is not obvious that we can do better than cubic time for this problem.

However, let’s see where some program calculation leads us. We can start by installing the definition of \texttt{segments}:

\[
\texttt{maximum . map sum . concat . map \texttt{inits . tails}}
\]

Searching for a law we can apply, we spot that

\[
\texttt{map f . concat = concat . map (map f)}
\]

applies to the subterm \texttt{map sum . concat}. That gives

\[
\texttt{maximum . concat . map (map sum) . map \texttt{inits . tails}}
\]

Now we can use the law \texttt{map f . map g = map (f . g)} to give

\[
\texttt{maximum . concat . map (map sum . \texttt{inits}) . tails}
\]

Oh, we can also use the law

\[
\texttt{maximum . concat = maximum . map maximum}
\]

can’t we? No, not unless the argument to \texttt{concat} is a nonempty list of nonempty lists, because the maximum of the empty list is undefined. In the present example the rule is valid because both \texttt{inits} and \texttt{tails} return nonempty lists. That leads to
maximum . map (maximum . map sum . inits) . tails

The next step is to use the property of \texttt{scanl} described in the previous section, namely

\[
\text{map sum . inits} = \text{scanl (+) 0}
\]

That leads to

\[
\text{maximum . map (maximum . scanl (+) 0) . tails}
\]

Already we have reduced a \(n^3\) algorithm to a \(n^2\) one, so we are making progress. But now we appear stuck since there is no law in our armoury that seems to help.

The next step obviously concerns \texttt{maximum . scanl (+) 0}. So, let’s see what we can prove about

\[
\text{foldr1 max . scanl (+) 0}
\]

This looks like a fusion rule, but can \texttt{scanl (+) 0} be expressed as a \texttt{foldr}? Well, we do have, for instance,

\[
\text{scanl (+) 0} [x,y,z] = [0,0+x,(0+x)+y,((0+x)+y)+z] = [0,x,x+y,x+y+z] = 0:\text{map (x+)} [0,y,y+z] = 0:\text{map (x+)} (\text{scanl (+) 0} [y,z])
\]

This little calculation exploits the associativity of \((+\) and the fact that 0 is the identity element of \((+\). The result suggests, more generally, that

\[
\text{scanl (@) e} = \text{foldr f [e]}
\]

where \(f\) is associative with identity \(e\). Let us take this on trust and move on to the conditions under which

\[
\text{foldr1 (<>)} . \text{foldr f [e]} = \text{foldr h b}
\]

where \(f\) is associative with identity \(e\). Let us take this on trust and move on to the conditions under which

\[
\text{foldr1 (<>)} . \text{foldr f [e]} = \text{foldr h b}
\]

where \(f\) is associative with identity \(e\). Let us take this on trust and move on to the conditions under which

\[
\text{foldr1 (<>)} (e:\text{map (x@) xs}) = h x (\text{foldr1 (<>)} xs)
\]

for all \(x\) and \(xs\). The left-hand side simplifies to
Proofs

\[ e \leftrightarrow (\text{foldr1} (\leftrightarrow) (\text{map} (x@) \text{xs})) \]

Taking the singleton case \( \text{xs} = [y] \), we find that

\[ h \ x \ y = e \leftrightarrow (x \ @ \ y) \]

That gives us our definition of \( h \), but we still have to check that

\[ \text{foldr1} (\leftrightarrow) (e:\text{map} (x@) \text{xs}) = e \leftrightarrow (x \ @ \ \text{foldr1} (\leftrightarrow) \text{xs}) \]

Simplifying both sides, this equation holds provided

\[ \text{foldr1} (\leftrightarrow) \cdot \text{map} (x@) = (x@) \cdot \text{foldr1} (\leftrightarrow) \]

This final equation holds provided \((@)\) distributes over \((\leftrightarrow)\); that is

\[ x \ @ (y \leftrightarrow z) = (x \ @ y) \leftrightarrow (x \ @ z) \]

The proof is left as an exercise.

Does addition distribute over (binary) maximum? Yes:

\[
\begin{align*}
x + (y \ `\text{max}` \ z) &= (x + y) \ `\text{max}` (x + z) \\
x + (y \ `\text{min}` \ z) &= (x + y) \ `\text{min}` (x + z)
\end{align*}
\]

Back to the maximum segment sum. We have arrived at

\[
\text{maximum} \cdot \text{map} (\text{foldr} (\@) 0) \cdot \text{tails}
\]

where \( x \ @ y = 0 \ `\text{max}` (x + y) \)

What we have left looks very like an instance of the \text{scanl} rule of the previous section, except that we have a \text{foldr} not a \text{foldl} and a \text{tails} not an \text{inits}. But a similar calculation to the one about \text{scanl} reveals

\[ \text{map} (\text{foldr} f e) \cdot \text{tails} = \text{scanr} f e \]

where

\[
\begin{align*}
\text{scanr} :: (a \to b \to b) \to b \to [a] \to [b] \\
\text{scanr} f e [] &= [e] \\
\text{scanr} f e (x:xs) &= f \ x \ (\text{head} \ ys):ys \\
&\quad \text{where} \ ys = \text{scanr} f e \ xs
\end{align*}
\]

The function \text{scanr} is also defined in the standard prelude. In summary,

\[ \text{mss} = \text{maximum} \cdot \text{scanr} (\@) 0 \]

where \( x \ @ y = 0 \ `\text{max}` (x + y) \)

The result is a linear-time program for the maximum segment sum.
Exercise A

In Chapter 3 we defined multiplication on natural numbers. The following definition is slightly different:

\[
\text{mult} :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \\
\text{mult} \Zero y = \Zero \\
\text{mult} \Succ x = \text{mult} x y + y
\]

Prove that \(\text{mult} \ (x+y) \ z = \text{mult} \ x \ z + \text{mult} \ y \ z\). You can use only the facts that \(x+0 = x\) and that \((+\) is associative. That means a long think about which variable \(x\), \(y\) or \(z\) is the best one on which to do the induction.

Exercise B

Prove that

\[
\text{reverse} \ (xs ++ ys) = \text{reverse} \ ys ++ \text{reverse} \ xs
\]

for all finite lists \(xs\) and \(ys\). You may assume that \((++)\) is associative.

Exercise C

Recall our friends Eager Beaver and Lazy Susan from Exercise D in Chapter 2. Susan happily used the expression \(\text{head} \ . \ \text{map} \ f\), while Beaver would probably prefer \(f \ . \ \text{head}\). Wait a moment! Are these two expressions equal? Carry out an induction proof to check.

Exercise D

Recall the cartesian product function \(\text{cp} :: [[[a]]] \rightarrow [[[a]]]\) from the previous chapter. Give a definition of the form \(\text{cp} = \text{foldr} \ f \ e\) for suitable \(f\) and \(e\). You can use a list comprehension for the definition of \(f\) if you like.

The rest of this exercise concerns the proof of the identity

\[
\text{length} \ . \ \text{cp} = \text{product} \ . \ \text{map} \ \text{length}
\]

where \(\text{product}\) returns the result of multiplying a list of numbers.

1. Using the fusion theorem, express \(\text{length} \ . \ \text{cp}\) as an instance of \(\text{foldr}\).
2. Express \(\text{map} \ \text{length}\) as an instance of \(\text{foldr}\).
3. Using the fusion theorem again, express \( \text{product} \cdot \text{map length} \) as an instance of \( \text{foldr} \).

4. Check that the two results are identical. If they aren’t, your definition of \( \text{cp} \) was wrong.

**Exercise E**

The first two arguments of \( \text{foldr} \) are replacements for the constructors

\[
(\_):= a \rightarrow [a] \rightarrow [a] \\
[] := [a]
\]

of lists. A fold function can be defined for any data type: just give replacements for the constructors of the data type. For example, consider

\[
data \text{Either } a \text{ } b = \text{Left } a \mid \text{Right } b
\]

To define a fold for \( \text{Either} \) we have to give replacements for

\[
\text{Left} :: a \rightarrow \text{Either } a \text{ } b \\
\text{Right} :: b \rightarrow \text{Either } a \text{ } b
\]

That leads to

\[
\text{foldE} ::= (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow \text{Either } a \text{ } b \rightarrow c \\
\text{foldE } f \text{ } g \text{ (Left } x) = f \text{ } x \\
\text{foldE } f \text{ } g \text{ (Right } x) = g \text{ } x
\]

The type \( \text{Either} \) is not a recursive data type and \( \text{foldE} \) is not a recursive function. In fact \( \text{foldE} \) is a standard prelude function, except that it is called \( \text{either} \) not \( \text{foldE} \).

Now define fold functions for

\[
data \text{Nat} = \text{Zero} \mid \text{Succ } \text{Nat} \\
data \text{NEList } a = \text{One } a \mid \text{Cons } a \text{ (NEList } a)
\]

The second declaration introduces nonempty lists.

What is wrong with the Haskell definition of \( \text{foldr1} \)?

**Exercise F**

Prove that

\[
\text{foldl } f \text{ } e \text{ } xs = \text{foldr } (\text{flip } f) \text{ } e \text{ (reverse } xs)
\]

for all finite lists \( xs \). Also prove that
6.7 Exercises

\[
foldl \ (\@) \ e \ xs = foldr \ (<> \ ) \ e \ xs
\]
for all finite lists \( xs \), provided that
\[
(x \ <> \ y) @ z = x \ <> \ (y @ z)
\]
\[
e @ x = x \ <> \ e
\]

**Exercise G**

Using
\[
foldl \ f \ e \ (xs ++ ys) = foldl \ f \ (foldl \ f \ e \ xs) \ ys
\]
\[
foldr \ f \ e \ (xs ++ ys) = foldr \ f \ (foldr \ f \ e \ ys) \ xs
\]
prove that
\[
foldl \ f \ e \ . \ concat = foldl \ (foldl \ f) \ e
\]
\[
foldr \ f \ e \ . \ concat = foldr \ (flip \ (foldr \ f)) \ e
\]

**Exercise H**

Mathematically speaking, what is the value of
\[
\text{sum} \ (\text{scanl} \ (/) \ 1 \ [1..]) \ ?
\]

**Exercise I**

Calculate the efficient definition of \( \text{scanr} \) from the specification
\[
\text{scanr} \ f \ e = \text{map} \ (\text{foldr} \ f \ e) \ . \ \text{tails}
\]

**Exercise J**

Consider the problem of computing
\[
mss :: [\text{Int}] \to \text{Int}
mss = \text{maximum} \ . \ \text{map} \ \text{sum} \ . \ \text{subseqs}
\]
where \( \text{subseqs} \) returns all the subsequences of a finite list, including the list itself:
\[
\text{subseqs} :: [a] \to [[[a]]]
\]
\[
\text{subseqs} \ [] = [[]]
\]
\[
\text{subseqs} \ (x:xs) = \text{xss} ++ \text{map} \ (x:) \ \text{xss}
\]
\[
\text{where} \ \text{xss} = \ \text{subseqs} \ \text{xs}
\]

Find a more efficient alternative for \( \text{mss} \).
**Exercise K**

This question is in pieces.

1. The function `takePrefix p` applied to a list `xs` returns the longest initial segment of `xs` that satisfies `p`. Hence

   ```
   takePrefix :: ([a] -> Bool) -> [a] -> [a]
   ```

   What are the values of the following expressions?

   - `takePrefix nondec [1,3,7,6,8,9]`
   - `takePrefix (all even) [2,4,7,8]`

   Complete the right-hand side of

   ```
   takePrefix (all p) = ...
   ```

   Give a definition of `takePrefix` in terms of standard functions, including `inits`.

   We will return to `takePrefix` in the final part of this question.

2. The functions `one` and `none` are defined by the equations

   ```
   one x = [x]
   none x = []
   ```

   Complete the right-hand side of the following identities:

   ```
   none . f = ...
   map f . none = ...
   map f . one = ...
   ```

3. Recall that `fork (f,g) x = (f x, g x)`. Complete the identities

   ```
   fst . fork (f,g) = ...
   snd . fork (f,g) = ...
   fork (f,g) . h = ...
   ```

4. Define

   ```
   test p (f,g) x = if p x then f x else g x
   ```

   Complete the right-hand sides of

   ```
   test p (f,g) . h = ...
   h . test p (f,g) = ...
   ```

   The function `filter` can be defined by
6.8 Answers

\[ \text{filter } p = \text{concat . map (test } p \ (\text{one, none}) \) \]

Using the identities above, together with other standard identities, prove using equational reasoning that

\[ \text{filter } p = \text{map fst . filter snd . map (fork (id, p))} \]

(Hint: as always in calculations, start with the more complicated side.)

5. Recall the standard prelude functions \texttt{curry} and \texttt{uncurry} from the answer to Exercise K in Chapter 4:

\[
\begin{align*}
\texttt{curry} & : ( (a,b) \to c ) \to a \to b \to c \\
\texttt{curry } f \ x \ y & = f (x,y)
\end{align*}
\]

\[
\begin{align*}
\texttt{uncurry} & : ( a \to b \to c ) \to (a,b) \to c \\
\texttt{uncurry } f \ (x,y) & = f \ x \ y
\end{align*}
\]

Complete the right-hand side of

\[ \text{map (fork (f,g)) = uncurry zip . (??)} \]

6. Returning to \texttt{takePrefix}, use equational reasoning to calculate an efficient program for the expression

\[ \text{takePrefix} \ (p \ . \ \text{foldl } f \ e) \]

that requires only a linear number of applications of \( f \).

6.8 Answers

\textbf{Answer to Exercise A}

The proof is by induction on \( y \):

\textbf{Case 0}

\[
\begin{align*}
\texttt{mult } (x+0) \ z & \quad \texttt{mult } x \ z + \texttt{mult } 0 \ z \\
= \{ \text{since } x + 0 = x \} & \quad = \{ \text{mult.1} \} \\
\texttt{mult } x \ z & \quad \texttt{mult } x \ z + 0 \\
= \{ \text{since } x + 0 = x \} & \quad \texttt{mult } x \ z
\end{align*}
\]
Case \( y+1 \)

\[
\begin{align*}
mult (x+(y+1)) z &= \text{mult } x \ z + \text{mult } (y+1) \ z \\
 &= \{\text{as } (+) \text{ is associative}\} \\
&= \{\text{mult.2}\} \\
mult ((x+y)+1) z &= \text{mult } x \ z + (\text{mult } y \ z + z) \\
&= \{\text{mult.2}\} \\
mult (x+y) \ z + z &= (\text{mult } x \ z + \text{mult } y \ z) + z \\
&= \{\text{induction}\} \\
(mult \ x \ z + \text{mult } y \ z) + z
\end{align*}
\]

### Answer to Exercise B

The proof is by induction on \( xs \):

**Case \([\ ]\)**

\[
\begin{align*}
\text{reverse } ([\ ]+ys) &= \text{reverse } ys + \text{reverse } [\] \\
&= \{++.1\} \\
\text{reverse } ys &= \{\text{reverse.1}\} \\
&= \{\text{since } xs + [\] = xs\} \\
&= \text{reverse } ys
\end{align*}
\]

**Case \( x:xs \)**

\[
\begin{align*}
\text{reverse } ((x:xs)+ys) &= \text{reverse } ys + \text{reverse } (x:xs) \\
&= \{++.2\} \\
\text{reverse } (x:(xs+ys)) &= \{\text{reverse.2}\} \\
\text{reverse } (xs+ys) + [x] &= \{\text{induction}\} \\
(\text{reverse } ys + \text{reverse } xs) + [x]
\end{align*}
\]

and

\[
\begin{align*}
\text{reverse } ys + \text{reverse } (x:xs) &= \{\text{reverse.2}\} \\
\text{reverse } ys + (\text{reverse } xs + [x]) &= \{\text{since } (+) \text{ is associative}\} \\
(\text{reverse } ys + \text{reverse } xs) + [x]
\end{align*}
\]
Answer to Exercise C

We have to prove that

\[ \text{head} \left( \text{map} \ f \ \text{xs} \right) = f \left( \text{head} \ \text{xs} \right) \]

for all lists \( \text{xs} \), finite, partial or infinite. The case \( \text{undefined} \) and the inductive case \( x : \text{xs} \) are okay, but the case \( [] \) gives

\[
\begin{align*}
\text{head} \left( \text{map} \ f \ [] \right) &= \text{head} [] = \text{undefined} \\
f \left( \text{head} [] \right) &= f \text{ undefined}
\end{align*}
\]

Hence the law holds only if \( f \) is a strict function. Eager Beaver is not bothered by this since he can only construct strict functions.

Answer to Exercise D

We have

\[ \text{cp} = \text{foldr} \ \text{op} \ [] \]

where \( \text{op} \ \text{xs} \ \text{xss} = [x : y : s | x < - x : s, y < - x : s] \)

1. \( \text{length} \ . \ \text{cp} = \text{foldr} \ \text{h} \ b \) provided \( \text{length} \) is strict (it is) and

\[
\begin{align*}
\text{length} [] &= b \\
\text{length} \left( \text{op} \ \text{xs} \ \text{xss} \right) &= h \ \text{xs} \ (\text{length} \ \text{xss})
\end{align*}
\]

The first equation gives \( b = 1 \) and as

\[
\text{length} \left( \text{op} \ \text{xs} \ \text{xss} \right) = \text{length} \ \text{xs} \ * \ \text{length} \ \text{xss}
\]

the second equation gives \( h = (*) \ . \ \text{length} \).

2. \( \text{map} \ \text{length} = \text{foldr} \ f \ [] \) where \( f \ \text{xs} \ \text{ns} = \text{length} \ \text{xs} : \text{ns} \). A shorter definition is \( f = (:) \ . \ \text{length} \).

3. \( \text{product} \ . \ \text{map} \ \text{length} = \text{foldr} \ h \ b \) provided \( \text{product} \) is strict (it is) and

\[
\begin{align*}
\text{product} [] &= b \\
\text{product} \left( \text{length} \ \text{xs} : \text{ns} \right) &= h \ \text{xs} \ (\text{product} \ \text{ns})
\end{align*}
\]

The first equation gives \( b = 1 \), and as

\[
\text{product} \left( \text{length} \ \text{xs} : \text{ns} \right) = \text{length} \ \text{xs} \ * \ \text{product} \ \text{ns}
\]

the second equation gives \( h = (*) \ . \ \text{length} \).

4. The two definitions of \( h \) and \( b \) are identical.
Answer to Exercise E

The definition of $\text{foldN}$ is straightforward:

\[
\text{foldN} :: (a \rightarrow a) \rightarrow a \rightarrow \text{Nat} \rightarrow a
\]

\[
\text{foldN } f \ e \ \text{Zero} = e
\]

\[
\text{foldN } f \ e \ \text{(Succ } n) = f \ (\text{foldN } f \ e \ n)
\]

In particular,

\[
m+n = \text{foldN } \text{Succ } m \ n
\]

\[
m* n = \text{foldN } (+m) \ \text{Zero} \ n
\]

\[
m^n = \text{foldN } (*m) \ \text{(Succ Zero)} \ n
\]

For nonempty lists, the definition of $\text{foldNE}$ is:

\[
\text{foldNE} :: (a \rightarrow b \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow \text{NEList } a \rightarrow b
\]

\[
\text{foldNE } f \ g \ (\text{One } x) = g \ x
\]

\[
\text{foldNE } f \ g \ (\text{Cons } x \ xs) = f \ x \ (\text{foldNE } f \ g \ xs)
\]

To be a proper fold over nonempty lists, the correct definition of $\text{foldr1}$ should have been

\[
\text{foldr1} :: (a \rightarrow b \rightarrow b) \rightarrow (a \rightarrow b) \rightarrow \text{[a]} \rightarrow b
\]

\[
\text{foldr1 } f \ g \ [x] = g \ x
\]

\[
\text{foldr1 } f \ g \ (x:xs) = f \ x \ (\text{foldr1 } f \ g \ xs)
\]

The Haskell definition of $\text{foldr1}$ restricts $g$ to be the identity function.

Answer to Exercise F

Write $g = \text{flip } f$ for brevity. We prove that

\[
\text{foldl } f \ e \ xs = \text{foldr } g \ e \ (\text{reverse } xs)
\]

for all finite lists $xs$ by induction:

Case $[]$

\[
\text{foldl } f \ e \ [] = \{\text{foldl.1}\} \ e
\]

\[
\text{foldl } g \ e \ (\text{reverse } []) = \{\text{reverse.1}\} \ e
\]

\[
\text{foldl } g \ e \ [] = \{\text{foldl.1}\} \ e
\]
Case $x:xs$

\[
\text{foldl } f \ e \ (x:xs) \\
= \ \{\text{foldl.2}\} \\
\text{foldl } f \ (f \ e \ x) \ xs \\
= \ \{\text{induction}\} \\
\text{foldr } g \ (f \ e \ x) \ (\text{reverse } xs)
\]

and

\[
\text{foldr } g \ e \ (\text{reverse } (x:xs)) \\
= \ \{\text{reverse.2}\} \\
\text{foldr } g \ e \ (\text{reverse } xs \ ++ \ [x]) \\
= \ \{\text{claim: see below}\} \\
\text{foldr } g \ (\text{foldr } g \ e \ [x]) \ (\text{reverse } xs) \\
= \ \{\text{since } \text{foldr } (\text{flip } f) \ e \ [x] = f \ e \ x\} \\
\text{foldr } g \ (f \ e \ x) \ (\text{reverse } xs)
\]

The claim is that

\[
\text{foldr } f \ e \ (xs \ ++ \ ys) = \text{foldr } f \ (\text{foldr } f \ e \ ys) \ xs
\]

We leave the proof to the reader. By the way, we have the companion result that

\[
\text{foldl } f \ e \ (xs \ ++ \ ys) = \text{foldl } f \ (\text{foldl } f \ e \ xs) \ ys
\]

Again, the proof is left to you.

We prove

\[
\text{foldl } (\emptyset) \ e \ xs = \text{foldr } (<>) \ e \ xs
\]

for all finite lists $xs$ by induction. The base case is trivial. For the inductive case:

Case $x:xs$

\[
\text{foldl } (\emptyset) \ e \ (x:xs) \\
= \ \{\text{foldl.2}\} \\
\text{foldl } (\emptyset) \ (e \ @ \ x) \ xs \\
= \ \{\text{given that } e \ @ \ x = x \ <> \ e\} \\
\text{foldl } (\emptyset) \ (x \ <> \ e) \ xs
\]

\[
\text{foldr } (<>) \ e \ (x:xs) \\
= \ \{\text{foldr.2}\} \\
\text{foldr } (<>) \ (e \ @ \ x) \ xs \\
= \ \{\text{induction}\} \\
\text{foldr } (<>) \ (x \ <> \ e) \ xs
\]
The two sides have simplified to different results. We need another induction hypothesis:

\[ \text{foldl } (\@) (x \leftrightarrow y) \text{ xs} = x \leftrightarrow \text{foldl } (\@) y \text{ xs} \]

The base case is trivial. For the inductive case

**Case z:zs**

\[
\begin{align*}
\text{foldl } (\@) (x \leftrightarrow y) (z:zs) \\
&= \{\text{foldl.2}\} \\
&\text{foldl } (\@) ((x \leftrightarrow y) @ z) zs \\
&= \{\text{since } (x \leftrightarrow y) @ z = x \leftrightarrow (y @ z)\} \\
&\text{foldl } (\@) (x \leftrightarrow (y @ z)) zs \\
&= \{\text{induction}\} \\
&x \leftrightarrow \text{foldl } (\@) (y @ z) zs 
\end{align*}
\]

and

\[
\begin{align*}
x \leftrightarrow \text{foldl } (\@) y (z:zs) \\
&= \{\text{foldl.2}\} \\
&x \leftrightarrow \text{foldl } (\@) (y @ z) zs 
\end{align*}
\]

**Answer to Exercise G**

The proofs are by induction. The base cases are easy and the inductive cases are

\[
\begin{align*}
\text{foldl } f e (\text{concat } (xs:xss)) \\
&= \{\text{definition of concat}\} \\
&\text{foldl } f e (xs ++ \text{concat } xss) \\
&= \{\text{given property of foldl}\} \\
&\text{foldl } f (\text{foldl } f e xs) (\text{concat } xss) \\
&= \{\text{induction}\} \\
&\text{foldl } (\text{foldl } f) (\text{foldl } f e xs) xss \\
&= \{\text{definition of foldl}\} \\
&\text{foldl } (\text{foldl } f) e (xs:xss) 
\end{align*}
\]
and

\[
\text{foldr } f \; e \; (\text{concat } (\text{xs}:\text{xss})) \\
= \quad \{\text{definition of concat}\} \\
\text{foldr } f \; e \; (\text{xs }++ \text{ concat xss}) \\
= \quad \{\text{given property of foldr}\} \\
\text{foldr } f \; (\text{foldr } f \; e \; (\text{concat xss})) \; \text{xs} \\
= \quad \{\text{using flip}\} \\
\text{flip } (\text{foldr } f) \; \text{xs} \; (\text{foldr } f \; e \; (\text{concat xss})) \\
= \quad \{\text{induction}\} \\
\text{flip } (\text{foldr } f) \; \text{xs} \; (\text{foldr } (\text{flip } (\text{foldr } f)) \; e \; \text{xss}) \\
= \quad \{\text{definition of foldr}\} \\
\text{foldr } (\text{flip } (\text{foldr } f)) \; e \; (\text{xs}:\text{xss})
\]

**Answer to Exercise H**

Mathematically speaking,

\[
\text{sum } (\text{scanl } (/) \; 1 \; [1..]) = e
\]

since \(\sum_{n=0}^{\infty} \frac{1}{n!} = e\). Computationally speaking, replacing \([1..]\) by a finite list \([1..n]\) gives an approximation to \(e\). For example,

ghci> sum (scanl (/) 1 [1..20])
2.7182818284590455
ghci> exp 1
2.718281828459045

The standard prelude function \(\text{exp}\) takes a number \(x\) and returns \(e^x\). By the way, the prelude function \(\text{log}\) takes a number \(x\) and returns \(\log_e x\). If you want logarithms in another base, use \(\text{logBase}\) whose type is

\[
\text{logBase :: Floating a} \Rightarrow a \rightarrow a \rightarrow a
\]

**Answer to Exercise I**

We synthesise a more efficient definition by cases. The base case yields

\[
\text{scanr } f \; e \; [] = [e]
\]
and the inductive case $x:xs$ is:

$$\text{scanr } f \ e \ (x:xs)$$
$$= \{\text{specification}\}$$
$$\text{map } (\text{foldr } f \ e) \ (\text{tails } (x:xs))$$
$$= \{\text{tails.2}\}$$
$$\text{map } (\text{foldr } f \ e) \ ((x:xs):\text{tails } xs)$$
$$= \{\text{definition of map}\}$$
$$\text{foldr } f \ e \ (x:xs):\text{map } (\text{foldr } f \ e) \ (\text{tails } xs)$$
$$= \{\text{foldr.2 and specification}\}$$
$$f \ x \ (\text{foldr } f \ e \ xs):\text{scan } f \ e \ xs$$
$$= \{\text{claim: foldr } f \ e \ xs = \text{head } (\text{scanr } f \ e \ xs)\}$$
$$f \ x \ (\text{head } ys):ys \text{ where } ys = \text{scanr } f \ e \ xs$$

**Answer to Exercise J**

Firstly,

$$\text{subseqs } = \text{foldr } \text{op } [[]]$$
$$\text{where } \text{op } x \text{ xss } = \text{xss } ++ \text{map } (x:) \text{xss}$$

Appeal to the fusion law yields

$$\text{map } \text{sum} \cdot \text{subseqs } = \text{foldr } \text{op } [0]$$
$$\text{where } \text{op } x \text{ xs } = \text{xs } ++ \text{map } (x+) \text{xs}$$

A second appeal to fusion yields

$$\text{maximum } \cdot \text{map } \text{sum} \cdot \text{subseqs } = \text{foldr } \text{op } 0$$
$$\text{where } \text{op } x \ y = y \ \text{`max` } (x+y)$$

That will do nicely. Of course, $\text{sum} \cdot \text{filter } (>0)$ also does the job.

**Answer to Exercise K**

1. We have

$$\text{takePrefix } \text{nondec } [1,3,7,6,8,9] = [1,3,7]$$
$$\text{takePrefix } (\text{all even}) \ [2,4,7,8] = [2,4]$$

The identity is

$$\text{takePrefix } (\text{all } p) = \text{takeWhile } p$$
The specification is

\[ \text{takePrefix } p = \text{last } \cdot \text{filter } p \cdot \text{inits} \]

2. We have

\[ \text{none } \cdot f = \text{none} \]
\[ \text{map } f \cdot \text{none } = \text{none} \]
\[ \text{map } f \cdot \text{one } = \text{one } \cdot f \]

3. We have

\[ \text{fst } \cdot \text{fork } (f,g) = f \]
\[ \text{snd } \cdot \text{fork } (f,g) = g \]
\[ \text{fork } (f,g) \cdot h = \text{fork } (f\cdot h, g\cdot h) \]

4. We have

\[ \text{test } p \cdot (f,g) \cdot h = \text{test } (p\cdot h) \cdot (f \cdot h, g \cdot h) \]
\[ h \cdot \text{test } p \cdot (f,g) = \text{test } p \cdot (h \cdot f, h \cdot g) \]

The reasoning is:

\[ \text{map } \text{fst } \cdot \text{filter } \text{snd } \cdot \text{map } (\text{fork } (\text{id},p)) \]
\[ = \{ \text{definition of filter} \} \]
\[ \text{map } \text{fst } \cdot \text{concat } \cdot \text{map } (\text{test } \text{snd } (\text{one},\text{none})) \cdot \]
\[ \text{map } (\text{fork } (\text{id},p)) \]
\[ = \{ \text{since } \text{map } f \cdot \text{concat } = \text{concat } \cdot \text{map } (\text{map } f) \} \]
\[ \text{concat } \cdot \text{map } (\text{map } \text{fst } \cdot \text{test } \text{snd } (\text{one},\text{none}) \cdot \]
\[ \text{fork } (\text{id},p)) \]
\[ = \{ \text{second law of test; laws of one and none} \} \]
\[ \text{concat } \cdot \text{map } (\text{test } \text{snd } (\text{one } \cdot \text{fst},\text{none}) \cdot \]
\[ \text{fork } (\text{id},p)) \]
\[ = \{ \text{first law of test; laws of fork} \} \]
\[ \text{concat } \cdot \text{map } (\text{test } p \cdot (\text{one } \cdot \text{id}, \text{none } \cdot \text{fork } (\text{id},p))) \]
\[ = \{ \text{laws of id and none} \} \]
\[ \text{concat } \cdot \text{map } (\text{test } p \cdot (\text{one},\text{none})) \]
\[ = \{ \text{definition of filter} \} \]
\[ \text{filter } p \]

5. We have
Proofs

\[ \text{map (fork (f, g))} = \text{uncurry zip . fork (map f, map g)} \]

6. We have

\[
\begin{align*}
\text{filter (p . foldl f e) . inits} \\
= \{ \text{derived law of filter} \} \\
\text{map fst . filter snd . map (fork (id, p . foldl f e)) . inits} \\
= \{ \text{law of zip} \} \\
\text{map fst . filter snd . uncurry zip . fork (id, map (p . foldl f e)) . inits} \\
= \{ \text{law of fork} \} \\
\text{map fst . filter snd . uncurry zip . fork (inits, map (p . foldl f e)) . inits} \\
= \{ \text{scan lemma} \} \\
\text{map fst . filter snd . uncurry zip . fork (inits, map p . scanl f e)}
\end{align*}
\]

Hence

\[
\text{takePrefix (p . foldl f e)} \\
= \text{fst . last . filter snd . uncurry zip . fork (inits, map p . scanl f e)}
\]

6.9 Chapter notes

Gofer, an earlier version of Haskell designed by Mark Jones, was so named because it was GOod For Equational Reasoning. HUGS (The Haskell Users Gofer System) was an earlier alternative to GHCi, and used in the second edition of the book on which the current one is based, but is no longer maintained.

Many people have contributed to the understanding of the laws of functional programming, too many to list. The Haskellwiki page

haskell.org/haskellwiki/Equational_reasoning_examples

contains examples of equational reasoning and links to various discussions about the subject.

The fascinating history of the maximum segment sum problem is discussed in Jon Bentley’s Programming Pearls (second edition) (Addison-Wesley, 2000).
The question of efficiency has been an ever-present undercurrent in recent discussions, and the time has come to bring this important subject to the surface. The best way to achieve efficiency is, of course, to find a decent algorithm for the problem. That leads us into the larger topic of Algorithm Design, which is not the primary focus of this book. Nevertheless we will touch on some fundamental ideas later on. In the present chapter we concentrate on a more basic question: functional programming allows us to construct elegant expressions and definitions, but do we know what it costs to evaluate them? Alan Perlis, a US computer scientist, once inverted Oscar Wilde’s definition of a cynic to assert that a functional programmer was someone who knew the value of everything and the cost of nothing.

7.1 Lazy evaluation

We said in Chapter 2 that, under lazy evaluation, an expression such as

\[ \text{sqr (sqr (3+4))} \]

where \( \text{sqr x} = x \times x \), is reduced to its simplest possible form by applying reduction steps from the outside in. That means the definition of the function \( \text{sqr} \) is installed first, and its argument is evaluated only when needed. The following evaluation sequence follows this prescription, but is not lazy evaluation:

\[
\begin{align*}
\text{sqr (sqr (3+4))} \\
= \text{sqr (3+4) * sqr (3+4)} \\
= ((3+4)*(3+4)) * ((3+4)*(3+4)) \\
= \ldots \\
= 2401
\end{align*}
\]
The ellipsis in the penultimate line hides no fewer than four evaluations of 3+4 and two of 7*7. Clearly the simple policy of substituting argument expressions into function expressions is a very inefficient way of carrying out reduction.

Instead, lazy evaluation guarantees that when the value of an argument is needed, it is evaluated only once. Under lazy evaluation, the reduction sequence would unfold basically as follows:

\[
\text{sqr (sqr (3+4))}
= \text{let } x = \text{sqr (3+4)} \text{ in } x*x
= \text{let } y = 3+4 \text{ in }
  \text{let } x = y*y \text{ in } x*x
= \text{let } y = 7 \text{ in }
  \text{let } x = y*y \text{ in } x*x
= \text{let } x = 49 \text{ in } x*x
= 2401
\]

The expression 3+4 is evaluated only once (and so is 7*7). The names x and y have been bound to expressions using let, though in the implementation of Haskell these names are anonymous pointers to expressions. When an expression is reduced to a value, the pointer then points to the value and that value can then be shared.

Even then, the headline 'Under lazy evaluation arguments are evaluated only when needed and then only once!' doesn't tell the full story. Consider evaluation of \text{sqr (head xs)}. In order to evaluate \text{sqr} we have to evaluate its argument, but in order to evaluate \text{head xs} we do not have to evaluate xs all the way, but only to the point where it becomes an expression of the form \text{y:ys}. Then \text{head xs} can return \text{y} and \text{sqr (head xs)} can return \text{y*y}. More generally, an expression is said to be in head normal form if it is a function (such as \text{sqr}) or if it takes the form of a data constructor (such as (:) applied to its arguments. Every expression in normal form (i.e. in fully reduced form) is in head normal form but not vice versa. For example, (e1,e2) is in head normal form (because it is equivalent to (,) e1 e2, where (,) is the data constructor for pairs), but is in normal form only if both e1 and e2 are. Of course, for numbers or booleans there is no distinction between the two kinds of normal form.

'Under lazy evaluation arguments are evaluated only when needed and then only once, and then maybe only to head normal form' is not as catchy a headline as before, but it does tell a better story.

Next, consider the following two definitions of the inductive case of the function \text{subseqs} that returns all the subsequences of a list:
7.1 Lazy evaluation

\[
\text{subseqs } (x:xs) = \text{subseqs } xs ++ \text{map } (x:\_) (\text{subseqs } xs)
\]

\[
\text{subseqs } (x:xs) = xss ++ \text{map } (x:\_) xss
\]

where \(xss = \text{subseqs } xs\)

In the first definition the expression \text{subseqs } xs appears twice on the right-hand side, so it is evaluated twice when the subsequences of a given list are required. In the second definition this duplication of effort has been recognised by the programmer and a \text{where} clause has been used to ensure that \text{subseqs } xs is evaluated only once (we could also have used a \text{let} expression).

The important point is that you, the programmer, are in control of which definition you want. It is quite possible for Haskell to recognise the double occurrence and to \textit{abstract} it away using the equivalent of an internal \text{let} expression. This is a well-known technique called \textit{common subexpression elimination}. But Haskell doesn’t do this, and for a very good reason: it can cause a \textit{space leak}. The second definition of \text{subseqs } (x:xs) has the following problem: the list \text{subseqs } xs is constructed only once, but it is retained in its entirety in memory because its value is used again, namely in the second expression \text{map } (x:\_) xss.

Look at it this way: the first definition takes longer because computation is duplicated; the second definition is faster (though still exponential) but can rapidly run out of available space. After all, there are \(2^n\) subsequences of a list of length \(n\). There is a fundamental dichotomy in programming we can never get away from: to avoid doing something twice you have to use up space to store the result of doing it once.

Here is a related example. Consider the following two definitions in a script:

\[
\text{foo1 } n = \text{sum } (\text{take } n \text{ primes})
\]
\[
\text{where}
\]
\[
\text{primes } = [x | x \leftarrow [2..], \text{divisors } x == [x]]
\]
\[
\text{divisors } x = [d | d \leftarrow [2..x], x \ `\text{mod}\` d == 0]
\]

\[
\text{foo2 } n = \text{sum } (\text{take } n \text{ primes})
\]
\[
\text{primes } = [x | x \leftarrow [2..], \text{divisors } x == [x]]
\]
\[
\text{divisors } x = [d | d \leftarrow [2..x], x \ `\text{mod}\` d == 0]
\]

The programmer who wrote \text{foo1} decided to structure their script by making the definitions of both \text{primes} and \text{divisors} local to the definition of \text{foo1}, presumably because neither definition was used elsewhere in the script. The programmer who wrote \text{foo2} decided to allow these two subsidiary definitions to float to the status of a global or \textit{top-level} definition. You might think that doesn’t make any
Efficiency difference to the efficiency, but consider the following interaction with GHCi. (The command :set +s turns on some statistics which are printed after an expression is evaluated.)

```
ghci> :set +s
ghci> foo1 1000
   3682913
   (4.52 secs, 648420808 bytes)
ghci> foo1 1000
   3682913
   (4.52 secs, 648412468 bytes)
ghci> foo2 1000
   3682913
   (4.51 secs, 647565772 bytes)
ghci> foo2 1000
   3682913
   (0.02 secs, 1616096 bytes)
```

Why was the second evaluation of `foo2 1000` so much faster than the first, while the two evaluations of `foo1 1000` took the same time?

The answer is that in the definition of `foo2` the first 1000 elements of the list `primes` is demanded, so after evaluation `primes` now points to a list in which the first 1000 primes appear explicitly. The second evaluation of `foo 1000` does not require these primes to be computed again. Internally, the script has grown in size because `primes` now occupies at least 1000 units of space.

Programmer Three chooses to write `foo` in the following way:

```haskell
foo3 = \n -> sum (take n primes)
   where
     primes     = [x | x <- [2..], divisors x == [x]]
     divisors x = [d | d <- [2..x], x `mod` d == 0]
```

This uses a lambda expression to express `foo3` at the function level, but otherwise the definition is exactly the same as that of `foo1`. The alternative

```haskell
foo3 = sum . flip take primes
```

also works but seems a little obscure. Now we have

```
ghci> foo3 1000
   3682913
   (3.49 secs, 501381112 bytes)
```
7.2 Controlling space

Suppose we define \texttt{sum} by \texttt{sum = foldl (+) 0}. Under lazy evaluation the expression \texttt{sum [1..1000]} is reduced as follows

\[
\texttt{sum [1..1000]} \\
= \texttt{foldl (+) 0 [1..1000]} \\
= \texttt{foldl (+) (0+1) [2..1000]} \\
= \texttt{foldl (+) ((0+1)+2) [3..1000]} \\
= \ldots \\
= \texttt{foldl (+) (((0+1)+2)+ \ldots +1000) []} \\
= ((0+1)+2)+ \ldots +1000 \\
= \ldots \\
= 500500
\]

It requires 1000 units of space just to build up the arithmetic expression that sums the first 1000 numbers before it pops to the surface and is finally evaluated.

Much better is to use a mixture of lazy and eager evaluation:
\begin{verbatim}
sum [1..1000] 
= foldl (+) 0 [1..1000] 
= foldl (+) (0+1) [2..1000] 
= foldl (+) 1 [2..1000] 
= foldl (+) (1+2) [3..1000] 
= foldl (+) 3 [3..1000] 
= ... 
= foldl (+) 500500 []
= 500500
\end{verbatim}

While the list expression \([1..1000]\) is evaluated lazily, the second argument of \texttt{foldl}, the accumulated sum, is evaluated eagerly. The result of interleaving lazy and eager evaluation steps is a sequence that uses a constant amount of space.

This suggests that it would be useful to have some way of controlling the reduction order. Such a method is provided by a primitive function \texttt{seq} with type

\[ \texttt{seq} :: a \rightarrow b \rightarrow b \]

Evaluation of \(x \ `\texttt{seq}`\ y\) proceeds by first evaluating \(x\) (to head normal form) and then returning the result of evaluating \(y\). If evaluation of \(x\) does not terminate, then neither does \(x \ `\texttt{seq}`\ y\). It’s not possible to define \texttt{seq} in Haskell; instead Haskell provides it as a primitive function.

Now consider the following version \texttt{foldl’} of \texttt{foldl} that evaluates its second argument strictly:

\[ \texttt{foldl’} :: (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \]
\[
\texttt{foldl’} f e [] = e
\texttt{foldl’} f e (x:xs) = y `\texttt{seq}` \texttt{foldl’} f y xs
\text{where } y = f e x
\]

Haskell provides the function \texttt{foldl’} in the standard prelude (yes, with just this unimaginative name). Now we can define \texttt{sum = foldl’ (+) 0}, with the consequence that evaluation proceeds in constant space. In fact, \texttt{sum} is another prelude function with essentially this definition.

Is it the case that \texttt{foldl} is now redundant and can be replaced by the new improved \texttt{foldl’}’? The answer is in practice yes, but in theory no. It is possible to construct \(f, e\) and \(xs\) such that

\[ \texttt{foldl} f e xs \neq \texttt{foldl’} f e xs \]

However, when \(f\) is strict (recall that \(f\) is strict if \(f \bot = \bot\)) the two expressions do return the same result. The exercises go into details.
Tackling space

Armed with the above information, let's now consider a very instructive example: how to compute the average or mean of a list of numbers. Surely that is an easy problem, you might think, just divide the sum of the list by the length of the list:

```haskell
mean :: [Float] -> Float
mean xs = sum xs / length xs
```

There are lots of things wrong with this definition, not the least of which is that the expression on the right is not well-formed! The function \( \text{length} \) in Haskell has type \([a] \rightarrow \text{Int}\) and we can't divide a \( \text{Float} \) by an \( \text{Int} \) without performing an explicit conversion.

There is a function in the standard prelude that comes to our aid:

```haskell
fromIntegral :: (Integral a, Num b) => a -> b
fromIntegral = fromInteger . toInteger
```

Recall from Chapter 3 the two conversion functions

```haskell
toInteger :: (Integral a) => a -> Integer
toInteger :: (Num a) => Integer -> a
```

The first converts any integral type to an integer, and the second converts an integer to a number. Their composition converts an integral number, such as \( \text{Int} \), to a more general kind of number, such as \( \text{Float} \).

We can now rewrite \( \text{mean} \) to read

```haskell
mean :: [Float] -> Float
mean xs = sum xs / fromIntegral (length xs)
```

The second thing wrong with this definition is that it silently ignores the case of the empty list. What is \( 0/0 \)? Either we should identify the failing case with an explicit error message, or else adopt one common convention, which is to agree that the mean of the empty list should be zero:

```haskell
mean [] = 0
mean xs = sum xs / fromIntegral (length xs)
```

Now we are ready to see what is really wrong with \( \text{mean} \): it has a space leak. Evaluating \( \text{mean [1..1000]} \) will cause the list to be expanded and retained in memory after summing because there is a second pointer to it, namely in the computation of its length.
Efficiency

We can replace the two traversals of the list by one, using a strategy of program optimisation called *tupling*. The idea is simple enough in the present example: define \( \text{sumlen} \) by

\[
\text{sumlen} :: [\text{Float}] \to (\text{Float}, \text{Int})
\]

\[
\text{sumlen } x s = (\text{sum } x s, \text{length } x s)
\]

and then calculate an alternative definition that avoids the two traversals. It is easy to carry out the calculation and we just state the result:

\[
\begin{align*}
\text{sumlen } [] &= (0, 0) \\
\text{sumlen } (x:x:s) &= (s+x, n+1) \quad \text{where } (s, n) = \text{sumlen } x s
\end{align*}
\]

The pattern of the definition of \( \text{sumlen} \) should be familiar by now. An alternative definition is

\[
\text{sumlen} = \text{foldr } f (0, 0) \quad \text{where } f x (s, n) = (s+x, n+1)
\]

Even better, we can replace \( \text{foldr } f \) by \( \text{foldl } g \), where

\[
g (s, n) x = (s+x, n+1)
\]

The justification of this step is the law in the previous chapter that said

\[
\text{foldr } f e x s = \text{foldl } g e x s
\]

for all finite lists \( x s \), provided

\[
\begin{align*}
f x (g y z) &= g (f x y) z \\
f x e &= g e x
\end{align*}
\]

The verification of these two conditions is left as an exercise.

And that means we can use \( \text{foldl}' \):

\[
\text{sumlen} = \text{foldl}' g (0, 0) \quad \text{where } g (s, n) x = (s+x, n+1)
\]

Now we can replace our heavily criticised definition of \( \text{mean} \) by

\[
\begin{align*}
\text{mean } [] &= 0 \\
\text{mean } x s &= s / \text{fromIntegral } n \\
& \quad \text{where } (s, n) = \text{sumlen } x s
\end{align*}
\]

Surely we have now achieved our goal of a constant-space computation for \( \text{mean} \)?

Unfortunately not. The problem is with \( \text{sumlen} \) and it is a little tricky to spot. Expanding the definition out a little, we find
7.2 Controlling space

foldl' f (s,n) (x:xs) = y `seq` foldl' f y xs
  where y = (s+x,n+1)

Ah, but y `seq` z reduces y to head normal form and the expression (s+x,n+1) is already in head normal form. Its two components are not evaluated until the end of the computation. That means we have to dig deeper with our seqs and rewrite sumlen in the following way:

\[
\text{sumlen = foldl' f (0,0) }
\]
\[
\text{where f (s,n) x = s `seq` n `seq` (s+x,n+1) }
\]

Finally, everything in the garden is rosy and we have a computation that runs in constant space.

Two more application operators

Function application is the only operation not denoted by any visible sign. However, Haskell provides two more application operators, ($$) and ($$!):

\[
\text{infixr 0 $,$!}
\]
\[
($$),($$!) :: (a -> b) -> a -> b
\]
\[
f $$ x = f x
\]
\[
f $$! x = x `seq` f x
\]

The only difference between \( f \ x \) and \( f $$! \ x \) is that in the second expression the argument \( x \) is evaluated before \( f \) is applied. The only difference between \( f \ x \) and \( f $$ x \) is that ($$) (and also ($$!)) is declared to have the lowest binding power of 0 and to associate to the right in expressions. That is exactly what the fixity declaration in the first line provides. Why do we want that?

The answer is that we can now write, for example

\[
\text{process1 $$ process2 $$ process3 input}
\]

instead of having to write either of

\[
\text{process1 (process2 (process3 x))}
\]
\[
\text{(process1 . process2 . process3) x}
\]

It is undeniable that ($$) can be quite useful on occasions, especially when submitting expressions for evaluation with GHCI, so it’s worth mentioning its existence. And the strict application operator ($$!) is useful for the reasons discussed above.
We have seen that having an ‘eager’ button on our dashboard is a very simple way of controlling the space involved in driving a computation, but what about time? Unfortunately there is no analogous button for speeding up computations; instead we have to understand some of the things that can unintentionally slow down a computation. The Haskell platform comes with documentation on GHC, which contains useful advice on how to make your program run more quickly. The documentation makes three key points:

- Make use of GHC’s profiling tools. There is no substitute for finding out where your program’s time and space is really being used up. We will not discuss profiling in this book, but it is important to mention that such tools are available.

- The best way to improve a program’s performance is to use a better algorithm. We mentioned this point at the beginning of the chapter.

- It is far better to use library functions that have been Seriously Tuned by Someone Else, than to craft your own. You might be able to write a better sorting algorithm than the one provided in Data.List, but it will take you longer than just writing import Data.List (sort). This is particularly true when you use GHCi because GHCi loads compiled versions of the functions in its standard libraries. Compiled functions typically run about an order of magnitude faster than interpreted ones.

Much of the detailed advice in the GHC documentation is beyond the scope of this book, but two tips can be explained here. Firstly, the management of lazy evaluation involves more overheads than eager evaluation, so that if you know that a function’s value will be needed, it is better to push the eager button. As the documentation says: ‘Strict functions are your dear friends’.

The second piece of advice is about types. Firstly, Int arithmetic is faster than Integer arithmetic because Haskell has to perform more work in handling potentially very large numbers. So, use Int rather than Integer whenever it is safe to do so. Secondly, there is less housekeeping work for Haskell if you tailor the type of your function to the instance you want. For example, consider the type of foo1, defined in Section 7.1. There we did not provide a type signature for foo1 (or indeed for any of the other related functions) and that was a mistake. It turns out that

\[
\text{foo1} :: \text{Integral a} \Rightarrow \text{Int} \to a
\]
If we are really interested in the sum of the first \( n \) prime numbers, it is better to declare the type of \texttt{foo1} to be (say)

\[
\texttt{foo1 :: Int \to Integer}
\]

With this more specialised definition Haskell does not have to carry around a dictionary of the methods and instances of the type class \texttt{Integral}, and that lightens the load.

These pieces of advice can help shave off constant amounts of time and do not affect \textit{asymptotic} time complexity, the order of magnitude of the timing function. But sometimes we can write code that is inadvertently less efficient asymptotically than we intended. Here is an instructive example. Consider the cartesian product function \texttt{cp} discussed in Chapter 5:

\[
\texttt{cp \ [\] = [[]]}
\]
\[
\texttt{cp (xs:xss) = [x:ys \mid x \leftarrow xs, ys \leftarrow cp xss]}
\]

Pretty and clear enough you would think, but compare it with

\[
\texttt{cp' = foldr op [[]]}
\]
\[
\texttt{where \ op \ xs \ yss = [x:ys \mid x \leftarrow xs, ys \leftarrow yss]}
\]

The first version is a direct recursive definition, while the second uses \texttt{foldr} to encapsulate the pattern of the recursion. The two ‘algorithms’ are the same, aren’t they? Well,

\[
\texttt{ghci> sum$map sum$cp [[1..10] \mid j \leftarrow [1..6]]}
\]
\[
33000000
\]
\[
(12.11 \text{ secs}, 815874256 \text{ bytes})
\]

\[
\texttt{ghci> sum$map sum$cp' [[1..10] \mid j \leftarrow [1..6]]}
\]
\[
33000000
\]
\[
(4.54 \text{ secs}, 369640332 \text{ bytes})
\]

The expression \texttt{sum $ map $ sum} is there just to force complete evaluation of the cartesian product. Why is the first computation three times slower than the second?

To answer this question, look at the translation that eliminates the list comprehension in the first definition:

\[
\texttt{cp \ [\] = [[]]}
\]
\[
\texttt{cp (xs:xss) = concat (map f xs)}
\]
\[
\quad \texttt{where f x = [x:ys \mid ys \leftarrow cp xss]}
\]

Now we can see that \texttt{cp xss} is evaluated \textit{each time} \texttt{f} is applied to elements of \texttt{xss}. That means, in the examples above, that \texttt{cp} is evaluated many more times in
the first example than in the second. We cannot be more precise at this point, but will be below when we develop a little calculus for estimating running times. But the issue should be clear enough: the simple recursive definition of \( cp \) has led us inadvertently into a situation in which more evaluations are carried out than we intended.

One other way to get a more efficient cartesian product is to just write

\[
\begin{align*}
\text{cp } [] & = [[]] \\
\text{cp } (xs:xss) & = [x:ys \mid x \leftarrow xs, ys \leftarrow yss] \\
& \text{ where } yss = \text{cp } xss
\end{align*}
\]

This definition has exactly the same efficiency as the one in terms of \( \text{foldr} \). The lesson here is that innocent-looking list comprehensions can hide the fact that some expressions, though only written once, are evaluated multiple times.

### 7.4 Analysing time

Given the definition of a function \( f \) we will write \( T(f)(n) \) to denote an asymptotic estimate of the number of reduction steps required to evaluate \( f \) on an argument of ‘size’ \( n \) in the worst case. Moreover, for reasons explained in a moment, we will assume eager, not lazy, evaluation as the reduction strategy involved in defining \( T \).

The definition of \( T \) requires some amplification. Firstly, \( T(f) \) refers to the complexity of a given definition of \( f \). Time complexity is a property of an expression, not of the value of that expression.

Secondly, the number of reduction steps does not correspond exactly to the elapsed time between submitting an expression for evaluation and waiting for the answer. No account is taken of the time to find the next subexpression to be reduced in a possibly large and complicated expression. For this reason the statistics facility of GHCi does not count reduction steps, but produces a measure of elapsed time.

Thirdly, we do not formalise the notion of size, since different measures are appropriate in different situations. For example, the cost of evaluating \( xs++ys \) is best measured in terms of \( (m,n) \), a pair describing the lengths of the two lists. In the case of \( \text{concat } xss \) we could take the length of \( \text{concat } xss \) as a measure of size, but if \( xss \) is a list of length \( m \) consisting of lists all of length \( n \), then \( (m,n) \) might be a more suitable measure.

The fourth and crucial remark is that \( T(f)(n) \) is determined under an \textit{eager} evaluation model of reduction. The reason is simply that estimating the number of
reduction steps under lazy evaluation is difficult. To illustrate, consider the definition \( \text{minimum} = \text{head} \circ \text{sort} \). Under eager evaluation, the time to evaluate the minimum on a list of length \( n \) under this definition is given by

\[
T(\text{minimum})(n) = T(\text{sort})(n) + T(\text{head})(n).
\]

In other words we first have to completely sort a list of length \( n \) and then take the head of the result (presumably a constant-time operation). This equation does not hold under lazy evaluation, since the number of reduction steps required to find the head of \( \text{sort} \ \text{xs} \) requires only that \( \text{sort} \ \text{xs} \) be reduced to head normal form. How long that takes depends on the precise algorithm used for \( \text{sort} \). Timing analysis under eager reduction is simpler because it is \textit{compositional}. Since lazy evaluation never requires more reduction steps than eager evaluation, any upper bound for \( T(\text{f})(n) \) will also be an upper bound under lazy evaluation. Furthermore, in many cases of interest, a lower bound will also be a lower bound under lazy evaluation.

In order to give some examples of timing analyses we have to introduce a little order notation. So far, we have used the awkward phrase ‘taking a number of steps proportional to’ whenever efficiency is discussed. It is time to replace it by something shorter. Given two functions \( f \) and \( g \) on the natural numbers, we say that \( f \) is of order \( g \), and write \( f = \Theta(g) \) if there are positive constants \( C_1 \) and \( C_2 \) and a natural number \( n_0 \) such that \( C_1 g(n) \leq f(n) \leq C_2 g(n) \) for all \( n > n_0 \). In other words, \( f \) is bounded above and below by some constant times \( g \) for all sufficiently large arguments.

The notation is abused to the extent that one conventionally writes, for example, \( f(n) = \Theta(n^2) \) rather than the more correct \( f = \Theta(\lambda n. n^2) \). Similarly, one writes \( f(n) = \Theta(n) \) rather than \( f = \Theta(id) \). The main use of \( \Theta \)-notation is to hide constants; for example, we can write

\[
\sum_{j=1}^{n} j = \Theta(n^2) \quad \text{and} \quad \sum_{j=1}^{n} j^2 = \Theta(n^3)
\]

without bothering about the exact constants involved. When \( \Theta(g) \) appears in a formula it stands for some unnamed function \( f \) satisfying \( f = \Theta(g) \). In particular, \( \Theta(1) \) denotes an anonymous constant.

With that behind us, we give three examples of how to analyse the running time of a computation. Consider first the following two definitions of \( \text{concat} \):

\[
\text{concat} \ \text{xss} \ = \ \text{foldr} \ (++) \ [] \ \text{xss} \\
\text{concat'} \ \text{xss} \ = \ \text{foldl} \ (++) \ [] \ \text{xss}
\]

The two definitions are equivalent provided \( \text{xss} \) is a finite list. Suppose \( \text{xss} \) is a
list of length \( m \) of lists all of length \( n \). Then the first definition gives

\[
T(\text{concat})(m,n) = T(\text{foldr } (++) \ [])(m,n),
T(\text{foldr } (++) \ [])(0,n) = \Theta(1),
T(\text{foldr } (++) \ [])(m+1,n) = T(++)(n,mn) +
T(\text{foldr } (++) \ [])(m,n).
\]

The estimate \( T(++)(n,mn) \) arises because a list of length \( n \) is concatenated with a list of length \( mn \). Since \( T(++)(n,m) = \Theta(n) \), we obtain

\[
T(\text{foldr } (++) \ [])(m,n) = \sum_{k=0}^{m} \Theta(n) = \Theta(mn).
\]

For the second definition of \( \text{concat} \) we have

\[
T(\text{concat'})(m,n) = T(\text{foldl } (++))(0,m,n),
T(\text{foldl } (++))(k,0,n) = O(1),
T(\text{foldl } (++))(k,m+1,n) = T(++)(k,n) +
T(\text{foldl } (++))(k+n,m,n).
\]

The additional argument \( k \) refers to the length of the accumulated list in the second argument of \( \text{foldl} \). This time we obtain

\[
T(\text{foldl } (++))(k,m,n) = \sum_{j=0}^{m-1} \Theta(k+jn) = \Theta(k+m^2n).
\]

Hence \( T(\text{concat'})(m,n) = \Theta(m^2n) \). The conclusion, which was anticipated in the previous chapter, is that using \( \text{foldr} \) rather than \( \text{foldl} \) in the definition of \( \text{concat} \) leads to an asymptotically faster program.

For the second example let us time the two programs for \( \text{subseqs} \) discussed in Section 7.1, where we had either of the following two possibilities:

\[
\text{subseqs (x:xs)} = \text{subseqs xs ++ map (x:) (subseqs xs)}
\]
\[
\text{subseqs'} (x:xs) = \text{xss ++ map (x:) xss}
where \( \text{xss} = \text{subseqs'} \text{ xs} \)
\]

Bearing in mind that (i) if \( \text{xs} \) has length \( n \), then \( \text{subseqs } \text{xs} \) has length \( 2^n \); and (ii) the time for both the concatenation and for applying \( \text{map (x:)} \) is therefore \( \Theta(2^n) \), the two timing analyses give

\[
T(\text{subseqs})(n+1) = 2T(\text{subseqs})(n) + \Theta(2^n),
T(\text{subseqs'})(n+1) = T(\text{subseqs'})(n) + \Theta(2^n)
\]
together with $T(\text{subseqs})(0) = \Theta(1)$. We will just state the two solutions (which can be proved by a simple induction argument):

$$
T(\text{subseqs})(n) = \Theta(n2^n),
$$
$$
T(\text{subseqs}')(n) = \Theta(2^n).
$$

The latter is therefore asymptotically faster than the former by a logarithmic factor.

For the third example, let us time the two programs for \text{cp} discussed at the beginning of this section. The first one was

\[
\text{cp} \;[] = [[]] \\
\text{cp} \; (x:xss) = \text{reverse} xss ++ [x]
\]

Suppose once again that $xss$ is a list of length $m$ of lists all of length $n$. Then the length of \text{cp} $xss$ is $n^m$. Then we have

$$
T(\text{cp})(0,n) = \Theta(1),
$$
$$
T(\text{cp})(m+1,n) = nT(\text{cp})(m,n) + \Theta(n^m).
$$

because it takes $\Theta(n^m)$ steps to apply $(x:\text{cp})$ to every subsequence. The solution is

$$
T(\text{cp})(m,n) = \Theta(mn^m).
$$

On the other hand, the definition of \text{cp} in terms for \text{foldr} gives

$$
T(\text{cp})(0,n) = \Theta(1),
$$
$$
T(\text{cp})(m+1,n) = T(\text{cp})(m,n) + \Theta(n^m).
$$

with solution $T(\text{cp})(m,n) = \Theta(n^m)$. The second version is therefore asymptotically faster, again by a logarithmic factor.

\section{7.5 Accumulating parameters}

Sometimes we can improve the running time of a computation by adding an extra argument, called an \textit{accumulating parameter}, to a function. The canonical example is the function \text{reverse}:

\[
\text{reverse} \;[] = [] \\
\text{reverse} \; (x:xs) = \text{reverse} \; xs ++ [x]
\]

With this definition we have $T(\text{reverse})(n) = \Theta(n^2)$. In search of a linear-time program, suppose we define
revcat :: [a] -> [a] -> [a]
revcat xs ys = reverse xs ++ ys

It is clear that reverse xs = revcat xs [], so if we can obtain an efficient version of revcat we can obtain an efficient version of reverse. To this end we calculate a recursive definition of revcat. The base case revcat [] ys = ys is left as an exercise, and the inductive case is as follows:

\[
\begin{align*}
\text{revcat} \ (x:xs) \ ys &= \\
&= \text{reverse} \ (x:xs) \ ++ \ ys \\
&= \{\text{definition of reverse}\} \\
&= (\text{reverse} \ xs \ ++ \ [x]) \ ++ \ ys \\
&= \{\text{associativity of} (++)\} \\
&= \text{reverse} \ xs \ ++ \ ([x] \ ++ \ ys) \\
&= \{\text{definition of} (:)\} \\
&= \text{reverse} \ xs \ ++ \ (x:ys) \\
&= \{\text{definition of revcat}\} \\
&= \text{revcat} \ xs \ (x:ys)
\end{align*}
\]

Hence

\[
\begin{align*}
\text{revcat} \ [] \ ys &= ys \\
\text{revcat} \ (x:xs) \ ys &= \text{revcat} \ xs \ (x:ys)
\end{align*}
\]

As to the running time, \(T(\text{revcat})(m,n) = \Theta(m)\). In particular,

\[
T(\text{reverse}(n)) = T(\text{revcat}(n,0)) = \Theta(n)
\]

That gives a linear-time computation for reversing a list.

Here is another example. The function \text{length} is defined by

\[
\begin{align*}
\text{length} :: [a] \to \text{Int} \\
\text{length} [] &= 0 \\
\text{length} \ (x:xs) &= \text{length} \ xs \ + \ 1
\end{align*}
\]

We have \(T(\text{length})(n) = \Theta(n)\), so there is no time advantage in calculating another definition. Nevertheless, define \text{lenplus} by

\[
\begin{align*}
\text{lenplus} :: [a] \to \text{Int} \to \text{Int} \\
\text{lenplus} \ xs \ n &= \text{length} \ xs \ + \ n
\end{align*}
\]
If we go through exactly the same calculation for `lenplus` as we did for `revcat`, we arrive at

\[
\begin{align*}
\text{lenplus} & \quad [\text{xs}] \quad n \quad = \quad n \\
\text{lenplus} & \quad (\text{x}:\text{xs}) \quad n \quad = \quad \text{lenplus} \quad \text{xs} \quad (1+n)
\end{align*}
\]

The reason the calculation goes through is that `(+)`, like `(++)`, is an associative operation. The advantage of defining

\[
\text{length} \quad \text{xs} \quad = \quad \text{lenplus} \quad \text{xs} \quad 0 \quad = \quad \text{foldl} \quad (\n \quad x \rightarrow \quad 1+n) \quad 0 \quad \text{xs}
\]

is that, by using \text{foldl}' in place of \text{foldl}, the length of a list can be computed in constant space. That indeed is how \text{length} is defined in Haskell’s prelude.

As the really astute reader might have spotted, there is actually no need to go through the calculations above. Both the examples are, in fact, instances of a law already described in the previous chapter, namely that

\[
\text{foldr} \quad (\rightarrow) \quad e \quad \text{xs} \quad = \quad \text{foldl} \quad (\emptyset) \quad e \quad \text{xs}
\]

for all finite lists \text{xs} provided

\[
\begin{align*}
\text{x} \rightarrow (\text{y} @ \text{z}) &= (\text{x} \rightarrow \text{y}) @ \text{z} \\
\text{x} \rightarrow \text{e} &= \text{e} @ \text{x}
\end{align*}
\]

The two instances are:

\[
\begin{align*}
\text{foldr} \quad (\n x \rightarrow n+1) \quad 0 \quad \text{xs} &= \text{foldl} \quad (\n x \rightarrow 1+n) \quad 0 \quad \text{xs} \\
\text{foldr} \quad (\n \text{xs} \rightarrow \text{xs}++[\text{x}]) \quad [] \quad \text{xs} &= \text{foldl} \quad (\n \text{xs} \rightarrow [\text{x}]++\text{xs}) \quad [] \quad \text{xs}
\end{align*}
\]

We leave the detailed verification of these equations as an exercise.

For a final demonstration of the accumulating parameter technique we move from lists to trees. Consider the data declaration

\[
\text{data GenTree} \quad a = \quad \text{Node} \quad a \quad [\text{GenTree} \quad a]
\]

An element of this type is a tree consisting of a node with a label and a list of subtrees. Such trees arise in problems that can be formulated in terms of positions and moves. The label of a node specifies the current position, and the number of subtrees corresponds to the number of possible moves in the current position. Each subtree has a label that specifies the result of making the move, and its subtrees describe the moves that can be made from the new position. And so on.

Here is a function for computing the list of labels in a tree:
labels :: GenTree a -> [a]
lables (Node x ts) = x:concat (map labels ts)

The method is simple enough: compute the labels of each subtree, concatenate the
results, and stick the label of the tree at the front of the final list.

Let us analyse the running time of this program on a tree \( t \). To keep things simple,
suppose that \( t \) is a \textit{perfect} \( k \)-ary tree of height \( h \). What that means is that if \( h = 1 \)
then \( t \) has no subtrees, while if \( h > 1 \) then \( t \) has exactly \( k \) subtrees, each with height
\( h-1 \). The number \( s(h,k) \) of labels in such a tree satisfies

\[
\begin{align*}
  s(1,t) &= 1, \\
  s(h+1,k) &= 1 + ks(h,k),
\end{align*}
\]

with solution \( s(h,k) = \Theta(k^h) \). Now we have

\[
\begin{align*}
  T(\text{labels})(1,k) &= \Theta(1), \\
  T(\text{labels})(h+1,k) &= \Theta(1) + T(\text{concat})(k,s) + T(\text{map labels})(h,k),
\end{align*}
\]

where \( s = s(h,k) \). The term \( T(\text{map labels})(h,k) \) estimates the running time of
applying \text{map labels} to a list of length \( k \) of trees all of height \( h \). In general, given
a list of length \( k \) consisting of elements each of size \( n \), we have

\[
T(\text{map } f)(k,n) = kT(f)(n) + \Theta(k).
\]

Furthermore \( T(\text{concat})(k,s) = \Theta(ks) = \Theta(k^{h+1}) \). Hence

\[
T(\text{labels})(h+1,k) = \Theta(k^{h+1}) + kT(\text{labels})(h,k)
\]

since \( \Theta(1) + \Theta(k) = \Theta(k) \). The solution is given by

\[
T(\text{labels})(h,k) = \Theta(hk^h) = \Theta(s \log s).
\]

In words, computing the labels of a tree using the definition above takes time that
is asymptotically greater than the size of the tree by a logarithmic factor.

Let us now see what an accumulating parameter can do. Define \text{labcat} by

\[
\begin{align*}
  \text{labcat} :: [\text{GenTree a}] -> [a] -> [a] \\
  \text{labcat} ts xs &= \text{concat} (\text{map labels ts}) ++ xs
\end{align*}
\]

As well as adding in a list \( xs \) we have also generalised the first argument from a
tree to a list of trees. We have \text{labels} \( t = \text{labcat} [t] [] \), so any improvement
on \text{labcat} leads to a corresponding improvement on \text{labels}.

We now synthesise an alternative definition for \text{labcat}. For the base case we obtain

\[
\text{labcat} [] xs = xs
\]
7.5 Accumulating parameters

For the inductive case we reason:

\[
\text{labcat (Node } x \, \text{us:vs}) \, \text{xs} \\
= \{\text{definition}\} \\
\text{concat (map labels (Node } x \, \text{us:vs})) \, ++ \, \text{xs} \\
= \{\text{definitions}\} \\
\text{labels (Node } x \, \text{us}) \, ++ \, \text{concat (map labels vs)} \, ++ \, \text{xs} \\
= \{\text{definition}\} \\
x: \text{concat (map labels us)} \, ++ \, \text{concat (map labels vs)} \, ++ \, \text{xs} \\
= \{\text{definition of labcat}\} \\
x: \text{concat (map labels us)} \, ++ \, \text{labcat vs xs} \\
= \{\text{definition of labcat (again)}\} \\
\text{labcat us (labcat vs xs)}
\]

The result of this calculation is the following program for \text{labels}:

\[
\begin{align*}
\text{labels } t &= \text{labcat } [t] \, [] \\
\text{labcat } [] \, \text{xs} &= \text{xs} \\
\text{labcat (Node } x \, \text{us:vs}) &= x: \text{labcat us (labcat vs xs)}
\end{align*}
\]

For the timing analysis, let \(T(\text{labcat})(h,k,n)\) estimate the running time of

\[
\text{labcat } ts \, \text{xs}
\]

when \(ts\) is a list of length \(n\) of trees, each of which is a perfect \(k\)-ary tree of height \(h\) (the size of \(xs\) is ignored since it doesn’t affect the estimate). Then

\[
T(\text{labcat})(h,k,0) = \Theta(1), \\
T(\text{labcat})(1,k,n+1) = \Theta(1) + T(\text{labcat})(1,k,n)), \\
T(\text{labcat})(h+1,k,n+1) = \Theta(1) + T(\text{labcat})(h,k,k) + T(\text{labcat})(h+1,k,n)).
\]

Solving the first two equations gives \(T(\text{labcat})(1,k,n) = \Theta(n)\). An induction argument now shows \(T(\text{labcat})(h,k,n) = \Theta(k^n n)\). Hence

\[
T(\text{labels})(h,k) = T(\text{labcat})(h,k,1) = \Theta(k^h) = \Theta(s).
\]

That means we can compute the labels of a tree in time proportional to the size of the tree, a logarithmic improvement over our first version.
7.6 Tupling

We met the idea of tupling two functions in the discussion of the function `mean`. Tupling is sort of dual to the method of accumulating parameters: we generalise a function not by including an extra argument but by including an extra result.

The canonical example of the power of tupling is the Fibonacci function:

```haskell
fib :: Int -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

The time to evaluate `fib` by these three equations is given by

\[
T(fib)(0) = \Theta(1),
T(fib)(1) = \Theta(1),
T(fib)(n) = T(fib)(n-1) + T(fib)(n-2) + \Theta(1).
\]

The timing function therefore satisfies equations very like that of `fib` itself. In fact \( T(fib)(n) = \Theta(\phi^n) \), where \( \phi \) is the golden ratio \( \phi = (1 + \sqrt{5})/2 \). That means that the running time to compute `fib` on an input \( n \) is exponential in \( n \).

Now consider the function `fib2` defined by

```haskell
fib2 n = (fib n, fib (n+1))
```

Clearly `fib n = fst (fib2 n)`. Synthesis of a direct recursive definition of `fib2` yields

```haskell
fib2 0 = (0,1)
fib2 n = (b,a+b) where (a,b) = fib2 (n-1)
```

This program takes linear time. In this example the tupling strategy leads to a dramatic increase in efficiency, from exponential time to linear time.

It’s great fun to formulate general laws that encapsulate gains in efficiency. One such law concerns the computation of

```haskell
(foldr f a xs, foldr g b xs)
```

As expressed above, the two applications of `foldr` involve two traversals of the list `xs`. There is a modest time advantage, and possibly a greater space advantage, in formulating a version that traverses the list only once. In fact

```haskell
(foldr f a xs, foldr g b xs) = foldr h (a,b) xs
```
where

\[ h \ x \ (y,z) = (f \ x \ y, g \ x \ z) \]

The result can be proved by induction and we leave details as an easy exercise.

As one more example, we again move from lists to trees. But this time we have a different kind of tree, a *leaf-labelled binary tree*:

\[
\text{data BinTree } a = \text{Leaf } a \mid \text{Fork (BinTree } a) \ (\text{BinTree } a)
\]

In contrast to a GenTree discussed above, a BinTree is either a leaf, with an associated label, or a fork of two subtrees.

Suppose we wanted to build such a tree with a given list as the labels. More precisely, we want to define a function \text{build} satisfying

\[
\text{labels (build } xs\text{)} = xs
\]

for all finite nonempty lists \(xs\), where \text{labels} returns the labels of a binary tree:

\[
\text{labels :: BinTree } a \to [a]
\]

\[
\text{labels (Leaf } x\text{)} = [x]
\]

\[
\text{labels (Fork } u \ v\text{)} = \text{labels } u \ ++ \ \text{labels } v
\]

We are attuned now to possible optimisations, and the definition of \text{labels} suggest that it could be improved with an accumulating parameter. So it can, but that is not our primary interest here, and we leave the optimisation as an exercise.

One way to build a tree is to arrange that half the list goes into the left subtree, and the other half into the right subtree:

\[
\text{build :: } [a] \to \text{BinTree } a
\]

\[
\text{build } [x] = \text{Leaf } x
\]

\[
\text{build } xs = \text{Fork (build } ys\text{)} \ (\text{build } zs)
\]

\[
\text{where } (ys,zs) = \text{halve } xs
\]

The function \text{halve} made an appearance in Section 4.8:

\[
\text{halve } xs = (\text{take } m \ xs, \text{drop } m \ xs)
\]

\[
\text{where } m = \text{length } xs \div 2
\]

Thus \text{halve} splits a list into two approximately equal halves. The definition of \text{halve} involves a traversal of the list to find its length, and two further (partial) traversals to compute the two components. It is therefore a prime candidate for applying the tupling strategy to get something better. But as with \text{labels} we are going to ignore that particular optimisation for now. And we are also going to
ignore the proof that this definition of build meets its specification. That’s three calculations we are leaving as exercises in order to concentrate on a fourth.

Let’s time build:

\[ T(\text{build})(1) = \Theta(1), \]
\[ T(\text{build})(n) = T(\text{build})(m) + T(\text{build})(n-m) + \Theta(n) \]

where \( m = n \div 2 \)

It takes \( \Theta(n) \) steps to halve a list of length \( n \), and then we recursively build two subtrees from lists of length \( m \) and \( n - m \), respectively. The solution is

\[ T(\text{build})(n) = \Theta(n \log n). \]

In words, building a tree by the above method takes longer than the length of the list by a logarithmic factor.

Having established this fact, let us define build2 by

\[
\text{build2 :: Int -> [a] -> (BinTree a,[a])}
\]
\[ \text{build2 n xs = (build (take n xs),drop n xs)} \]

This builds a tree from the first \( n \) elements, but also returns the list that is left. We have

\[ \text{build xs = fst (build2 (length xs) xs)} \]

so our original function can be determined from the tupled version.

Our aim now is to construct a direct recursive definition of build2. First of all, it is clear that

\[ \text{build2 1 xs = (Leaf (head xs),tail xs)} \]

For the recursive case we start with

\[
\text{build2 n xs = (Fork (build (take m (take n xs)))}
\]
\[ (\text{build (drop m (take n xs))),
\]
\[ \text{drop n xs) where m = n `div` 2} \]

This equation is obtained by substituting in the recursive case of build. It suggests that the next step is to use some properties of take and drop. Here they are: if \( m \leq n \) then

\[
\text{take m . take n = take m}
\]
\[ \text{drop m . take n = take (n-m) . drop m} \]

That leads to
build2 n xs = (Fork (build (take m xs))
   (build (take (n-m) (drop m xs))),
   drop n xs) where m = n `div` 2

Using the definition of build2 we can rewrite the above as follows:

build2 n xs = (Fork u v, drop n xs)
where (u,xs') = build2 m xs
   (v,xs'') = build2 (n-m) xs'
   m = n `div` 2

But as a final step, observe that

xs'' = drop (n-m) xs'
   = drop (n-m) (drop m xs)
   = drop n xs

Hence we can rewrite build2 once again to read

build2 1 xs = (Leaf (head xs),tail xs)
build2 n xs = (Fork u v, xs'')
where (u,xs') = build2 m xs
   (v,xs'') = build2 (n-m) xs'
   m = n `div` 2

Timing this program yields

\[ T(build2(1)) = \Theta(1), \]
\[ T(build2(n)) = T(build2(m)) + T(build2(n-m)) + \Theta(1). \]

with solution \( T(build2(n)) = \Theta(n) \). Using build2 as a subsidiary function has therefore improved the running time of build by a logarithmic factor.

### 7.7 Sorting

Sorting is a big topic and one can spend many happy hours tinkering with different algorithms. Knuth devotes about 400 pages to the subject in Volume 3 of his series *The Art of Computer Programming*. Even then some of his conclusions have to be reformulated when sorting is considered in a purely functional setting. Here we briefly consider two sorting algorithms, keeping an eye out for possible optimisations.
The sorting method called **Mergesort** made an appearance in Section 4.8:

```haskell
sort :: (Ord a) => [a] -> [a]
sort [] = []
sort [x] = [x]
sort xs = merge (sort ys) (sort zs)
  where (ys,zs) = halve xs
halve xs = (take m xs,drop m xs)
  where m = length xs `div` 2
```

In fact there are a number of variants for sorting by merging, and the standard prelude function `sort` uses a different variant than the one above.

As we said above, the definition of `halve` looks fairly inefficient in that it involves multiple traversals of its argument. One way to improve matters is to make use of the standard prelude function `splitAt`, whose specification is

```haskell
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs,drop n xs)
```

The prelude version of this function is the result of a tupling transformation:

```haskell
splitAt 0 xs = ([],xs)
splitAt n [] = ([],[])
splitAt n (x:xs) = (x:ys,zs)
  where (ys,zs) = splitAt (n-1) xs
```

It is easy enough to calculate this definition using the two facts that

```haskell
take n (x:xs) = x:take (n-1) xs
drop n (x:xs) = drop (n-1) xs
```

provided $0 < n$. Now we have

```haskell
halve xs = splitAt (length xs `div` 2) xs
```

There are still two traversals here of course.

Another way to improve `sort` is to define

```haskell
sort2 n xs = (sort (take n xs),drop n xs)
```

We have `sort xs = fst (sort2 (length xs) xs)`, so our original sorting function can be retrieved from the general one. An almost exactly similar calculation to the one in the previous section leads to
sort2 0 xs = ([],xs)
sort2 1 xs = ([head xs],tail xs)
sort2 n xs = (merge ys zs, xs'')
    where (ys,xs') = sort2 m xs
          (zs,xs'') = sort2 (n-m) xs'
          m = n `div` 2

With this definition there are no length calculations and no multiple traversals of xs.

Another way to optimise halve is to realise that no human would split up a list in this way if forced to do so by hand. If asked to divide a list into two, you and I would surely just deal out the elements into two piles:

halve [] = ([],[])  
ahalve [x] = ([x],[])  
ahalve (x:y:xs) = (x:ys,y:zs)
    where (ys,zs) = halve xs

Of course, this definition returns a different result than the previous one, but the order of the elements in the two lists does not matter if the result is to be sorted; what is important is that the elements are all there.

That is a total of three ways to improve the performance of sort. However, it turns out that none of them makes that much difference to the total running time. A few per cent perhaps, but nothing substantial. Furthermore, if we are using GHCi as our functional evaluator, none of the versions compares in performance to the library function sort because that function is given to us in a compiled form, and compiled versions of functions are usually about ten times faster. We can always compile our functions using GHC of course.

**Quicksort**

Our second sorting algorithm is a famous one called *Quicksort*. It can be expressed in just two lines of Haskell:

sort :: (Ord a) => [a] -> [a]
sort [] = []
sort (x:xs) = sort [y | y <- xs, y < x] ++ [x] ++
                  sort [y | y <- xs, x <= y]
That’s very pretty and a testament to the expressive power of Haskell. But the prettiness comes at a cost: the program can be very inefficient in its use of space. The situation is the same as with the program for \texttt{mean} seen earlier.

Before plunging into ways the code can be optimised, let’s compute $T(sort)$. Suppose we want to sort a list of length $n+1$. The first list comprehension can return a list of any length $k$ from 0 to $n$. The length of the result of the second list comprehension is therefore $n-k$. Since our timing function is an estimate of the worst-case running time, we have to take the maximum of these possibilities:

$$T(sort)(n+1) = \max \left[ T(sort)(k) + T(sort)(n-k) \mid k \leftarrow [0..n] \right] + \Theta(n).$$

The $\Theta(n)$ term accounts for both the time to evaluate the two list comprehensions and the time to perform the concatenations. Note, by the way, the use of a list comprehension in a mathematical expression rather than a Haskell one. If list comprehensions are useful notations in programming, they are useful in mathematics too.

Although not immediately obvious, the worst case occurs when $k=0$ or $k=n$. Hence

$$T(sort)(0) = \Theta(1),
T(sort)(n+1) = T(sort)(n) + \Theta(n),$$

with solution $T(sort)(n) = \Theta(n^2)$. Thus Quicksort is a quadratic algorithm in the worst case. This fact is intrinsic to the algorithm and has nothing to do with the Haskell expression of it. Quicksort achieved its fame for two other reasons, neither of which hold in a purely functional setting. Firstly, when Quicksort is implemented in terms of arrays rather than lists, the partitioning phase can be performed \textit{in place} without using any additional space. Secondly, the average case performance of Quicksort, under reasonable assumptions about the input, is $\Theta(n \log n)$ with a smallish constant of proportionality. In a functional setting this constant is not so small and there are better ways to sort than Quicksort.

With this warning, let us now see what we can do to optimise the algorithm without changing it in any essential way (i.e. to a completely different sorting algorithm). To avoid the two traversals of the list in the partitioning process, define

$$\text{partition } p \, \texttt{xs} = (\text{filter } p \, \texttt{xs}, \text{filter } (\not\cdot \, p) \, \texttt{xs})$$

This is another example of tupling two definitions to save on a traversal. Since \texttt{filter \ p} can be expressed as an instance of \texttt{foldr} we can appeal to the tupling law of \texttt{foldr} to arrive at
7.7 Sorting

\[
\text{partition } p = \text{foldr } \text{op } ([],[]) \\
\text{where } \text{op } x (ys,zs) | p x = (x:ys,zs) \\
| \text{otherwise } = (ys,x:zs)
\]

Now we can write

\[
\text{sort } [] = [] \\
\text{sort } (x:xs) = \text{sort } ys ++ [x] ++ \text{sort } zs \\
\text{where } (ys,zs) = \text{partition } (<x) xs
\]

But this program still contains a space leak. To see why, let us write the recursive case in the equivalent form

\[
\text{sort } (x:xs) = \text{sort } (\text{fst } p) ++ [x] ++ \text{sort } (\text{snd } p) \\
\text{where } p = \text{partition } (<x) xs
\]

Suppose \(x:xs\) has length \(n+1\) and is in strictly decreasing order, so \(x\) is the largest element in the list and \(p\) is a pair of lists of length \(n\) and 0, respectively. Evaluation of \(p\) is triggered by displaying the results of the first recursive call, but the \(n\) units of space occupied by the first component of \(p\) cannot be reclaimed because there is another reference to \(p\) in the second recursive call. Between these two calls further pairs of lists are generated and retained. All in all, the total space required to evaluate \(\text{sort}\) on a strictly decreasing list of length \(n+1\) is \(\Theta(n^2)\) units. In practice this means that evaluation of \(\text{sort}\) on some large inputs can abort owing to lack of sufficient space.

The solution is to force evaluation of \(\text{partition}\) and, equally importantly, to bind \(ys\) and \(zs\) to the components of the pair, not to \(p\) itself.

One way of bringing about a happy outcome is to introduce two accumulating parameters. Define \(\text{sortp}\) by

\[
\text{sortp } x \; xs \; us \; vs = \text{sort } (us ++ ys) ++ [x] ++ \\
\text{sort } (vs ++ zs) \\
\text{where } (ys,zs) = \text{partition } (<x) xs
\]

Then we have

\[
\text{sort } (x:xs) = \text{sortp } x \; xs \; [] \; []
\]

We now synthesise a direct recursive definition of \(\text{sortp}\). The base case is

\[
\text{sortp } x \; [] \; us \; vs = \text{sort } us ++ [x] ++ \text{sort } vs
\]
For the recursive case $y:xs$ let us assume that $y < x$. Then

$$\text{sortp } x \ (y:xs) \ \text{us vs}$$

$$= \ \{\text{definition of sortp with } (ys,zs) = \text{partition } (<x) \ xs\}$$

$$\text{sort } (\text{us ++ y:ys ++ [x] ++ sort } (\text{vs ++ zs}))$$

$$= \ \{\text{claim (see below)}\}$$

$$\text{sort } (y:us ++ ys) ++ [x] ++ \text{sort } (\text{vs ++ zs})$$

$$= \ \{\text{definition of sortp}\}$$

$$\text{sortp } x \ (y:us) \ \text{vs}$$

The claim is that if $as$ is any permutation of $bs$ then $\text{sort } as$ and $\text{sort } bs$ return the same result. The claim is intuitively obvious: sorting a list depends only on the elements in the input not on their order. A formal proof is omitted.

Carrying out a similar calculation in the case that $x \leq y$ and making $\text{sortp}$ local to the definition of $\text{sort}$, we arrive at the final program

$$\text{sort } [] = []$$

$$\text{sort } (x:xs) = \text{sortp } xs \ [] \ []$$

where

$$\text{sortp } [] \ \text{us vs} = \text{sort } \text{us ++ [x] ++ sort } \text{vs}$$

$$\text{sortp } (y:xs) \ \text{us vs} = \text{if } y < x$$

$$\text{then } \text{sortp } \text{xs } (y:us) \ \text{vs}$$

$$\text{else } \text{sortp } \text{xs } \text{us } (y:vs)$$

Not quite as pretty as before, but at least the result has $\Theta(n)$ space complexity.

7.8 Exercises

Exercise A

One simple definition of $\text{sort}$ is

$$\text{sort } [] = []$$

$$\text{sort } (x:xs) = \text{insert } x \ (\text{sort } xs)$$

$$\text{insert } x \ [] = [x]$$

$$\text{insert } x \ (y:ys)$$

$$= \text{if } x \leq y \ \text{then } x:y:ys \ \text{else } y:\text{insert } x \ ys$$

This method is called insertion sort. Reduce $\text{sort } [3,4,2,1]$ to head normal
form under lazy evaluation. Now answer the following questions: (i) How long, as a function of $n$, does it take to compute $\text{head . sort}$ when applied to a list of length $n$? (ii) How long does it take under eager evaluation? (iii) Does insertion sort, evaluated lazily, carry out exactly the same sequence of comparisons as the following selection sort algorithm?

$$
\text{sort} \; [] = [] \\
\text{sort} \; \text{xs} = y:\text{sort} \; \text{ys} \; \text{where} \; (y,\text{ys}) = \text{select} \; \text{xs}
$$

$$
\text{select} \; [x] = (x,[]) \\
\text{select} \; (x:\text{xs}) \; | \; x \leq y = (x,y:\text{ys}) \\
| \; \text{otherwise} = (y,x:\text{ys}) \\
\; \text{where} \; (y,\text{ys}) = \text{select} \; \text{xs}
$$

**Exercise B**

Write down a definition of $\text{length}$ that evaluates in constant space. Write a second definition of $\text{length}$ that evaluates in constant space but does not make use of the primitive $\text{seq}$ (either directly or indirectly).

**Exercise C**

Construct $f$, $e$ and $\text{xs}$ so that

$$
\text{foldl} \; f \; e \; \text{xs} \neq \text{foldl}' \; f \; e \; \text{xs}
$$

**Exercise D**

Would

$$
\text{cp} \; [] = [[]] \\
\text{cp} \; (\text{xss}:\text{xs}) = [x:ys \; | \; ys \leftarrow \text{cp} \; \text{xss}, \; x \leftarrow \text{xs}]
$$

be an alternative way of defining the function $\text{cp}$ that is as efficient as the definition in terms of $\text{foldr}$? Yes, No or Maybe?

Time for a calculation. Use the fusion law of $\text{foldr}$ to calculate an efficient alternative to

$$
\text{fcp} = \text{filter} \; \text{nondec} \; . \; \text{cp}
$$

See Section 4.7 for a definition of $\text{nondec}$. 
Exercise E

Suppose

\[ T(1) = \Theta(1), \]
\[ T(n) = T(n \div 2) + T(n - n \div 2) + \Theta(n) \]

for \( 2 \leq n \). Prove that \( T(2^k) = \Theta(k2^k) \). Hence prove \( T(n) = \Theta(n \log n) \).

Exercise F

Prove that

\[ \text{foldr } (\lambda x \ n \rightarrow n+1) \ 0 \ xs = \text{foldl } (\lambda n \rightarrow 1+n) \ 0 \ xs \]
\[ \text{foldr } (\lambda x \ xs \rightarrow xs++[x]) \ [] \ xs \]
\[ = \text{foldl } (\lambda xs \rightarrow [x]+xs) \ [] \ xs \]

Exercise G

Prove that if \( h x (y,z) = (f x y, g x z) \), then

\[ (\text{foldr } f \ a \ xs, \text{foldr } g \ b \ xs) = \text{foldr } h \ (a,b) \ xs \]

for all finite lists \( xs \). A tricky question: does the result hold for all lists \( xs \)?

Now find a definition of \( h \) such that

\[ (\text{foldl } f \ a \ xs, \text{foldl } g \ b \ xs) = \text{foldl } h \ (a,b) \ xs \]

Exercise H

Recall that

\[ \text{partition } p \ xs = (\text{filter } p \ xs, \text{filter } (\text{not} \ . \ p) \ xs) \]

Express the two components of the result as instances of \text{foldr}. Hence use the result of the previous exercise to calculate another definition of \text{partition}.

Define

\[ \text{part } p \ xs \ us \ vs = (\text{filter } p \ xs ++ \ us, \]
\[ \text{filter } (\text{not} \ . \ p) \ xs ++ \ vs) \]

Calculate another definition of \text{partition} that uses \text{part} as a local definition.

Exercise I

Recall that
labels :: BinTree a -> [a]
labels (Leaf x) = [x]
labels (Fork u v) = labels u ++ labels v

Compute $T(\text{labels})(n)$, where $n$ is the number of leaves in the tree. Now use the accumulating parameter technique to find a faster way of computing labels.

Prove that $\text{labels } (\text{build } \text{xs}) = \text{xs}$ for all finite nonempty lists $\text{xs}$.

**Exercise J**

Define $\text{select } k = (\!\!k) \cdot \text{sort}$, where $\text{sort}$ is the original Quicksort. Thus $\text{select } k$ selects the $k$th smallest element of a nonempty finite list of elements, the $0$th smallest being the smallest element, the $1$st smallest being the next smallest element, and so on. Calculate a more efficient definition of $\text{select}$ and estimate its running time.

7.9 Answers

**Answer to Exercise A**

$$\text{sort } [3,4,1,2]$$
$$= \text{insert } 3 (\text{sort } [4,1,2])$$
$$= ...$$
$$= \text{insert } 3 (\text{insert } 4 (\text{insert } 1 (\text{insert } 2 [])))$$
$$= \text{insert } 3 (\text{insert } 4 (\text{insert } 1 (2:[])))$$
$$= \text{insert } 3 (\text{insert } 4 (1:2:[]))$$
$$= \text{insert } 3 (1:\text{insert } 4 (2:[]))$$
$$= 1:\text{insert } 3 (\text{insert } 4 (2:[]))$$

It takes $\Theta(n)$ steps to compute $\text{head} \cdot \text{sort}$ on a list of length $n$. Under eager evaluation it takes about $n^2$ steps. As to part (iii), the answer is yes. You may think we have defined sorting by insertion, but under lazy evaluation it turns out to be selection sort. The lesson here is that, under lazy evaluation, you don’t always get what you think you are getting.

**Answer to Exercise B**

For the first part, the following does the job:

$$\text{length} = \text{foldl'} (\lambda x \rightarrow x + 1) 0$$

For the second part, one solution is
The test `n==0` forces evaluation of the first argument.

**Answer to Exercise C**

Take $f \ n \ x = \text{if } x==0 \ \text{then undefined else 0}$. Then

$$\text{foldl } f \ 0 \ [0,2] = 0$$
$$\text{foldl}' \ f \ 0 \ [0,2] = \text{undefined}$$

**Answer to Exercise D**

The answer is: maybe! Although the given version of $cp$ is efficient, it returns the component lists in a different order than any of the definitions in the text. That probably doesn’t matter if we are only interested in the set of results, but it might affect the running time and result of any program that searched $cp$ to find some list satisfying a given property.

According to the fusion rule we have to find a function $g$ so that

$$\text{filter nondec } (f \ xs \ yss) = g \ xs \ (\text{filter nondec yss})$$

where $f \ xs \ yss = [x:ys \ | \ x <- xs, ys <- yss]$. Then we would have

$$\text{filter nondec . } cp$$
$$= \text{filter nondec . foldr } f \ []$$
$$= \text{foldr } g \ []$$

Now

$$\text{nondec } (x:ys) = \text{null } ys \ || (x <= \text{head } ys \ \&\& \text{nondec } ys)$$

That leads to

$$g \ xs \ [] = [[x] \ | \ x <- xs]$$
$$g \ xs \ yss = [x:ys \ | \ x <- xs, ys <- yss, x <= \text{head } ys]$$

**Answer to Exercise E**

For the first part, we have

$$T(2^k) = 2T(2^{k-1}) + \Theta(2^k).$$
By induction we can show $T(2^k) = \sum_{i=0}^{k} k \Theta(2^k)$. The induction step is

$$T(2^k) = 2 \sum_{i=0}^{k-1} \Theta(2^{k-1}) + \Theta(2^k)$$

$$= \sum_{i=0}^{k-1} \Theta(2^k) + \Theta(2^k)$$

$$= \sum_{i=0}^{k} \Theta(2^k).$$

Hence $T(2^k) = \Theta(k2^k)$. Now suppose $2^k \leq n < 2^{k+1}$, so

$$\Theta(k2^k) = T(2^k) \leq T(n) \leq T(2^{k+1}) = \Theta((k+1)2^{k+1}) = \Theta(k2^k).$$

Hence $T(n) = \Theta(k2^k) = \Theta(n \log n)$.

Answer to Exercise F

Define $x <> n = n+1$ and $n @ x = 1+n$. We have

$$(x <> n) @ y = 1+(n+1) = (1+n)+1 = x <> (n @ y)$$

The second proof is similar.

Answer to Exercise G

The induction step is

$$(\text{foldr } f \ a \ (x:x:s), \text{foldr } g \ b \ (x:x:s))$$

$$= (f \ x \ (\text{foldr } f \ a \ x:s), g \ x \ (\text{foldr } g \ b \ x:s))$$

$$= h \ x \ (\text{foldr } f \ a \ x:s, \text{foldr } g \ b \ x:s)$$

$$= h \ x \ (\text{foldr } h \ (a,b) \ x:s)$$

The answer to the tricky question is No. The values $(\bot, \bot)$ and $\bot$ are different in Haskell. For example, suppose we define $\text{foo } (x,y) = 1$. Then

$$\text{foo } \text{undefined} = \text{undefined}$$

$$\text{foo } (\text{undefined}, \text{undefined}) = 1$$

For the last part, the definition of $h$ is that

$$h \ (y,z) \ x = (f \ y \ x, g \ z \ x)$$

Answer to Exercise H

We have $\text{filter } p = \text{foldr } (\text{op } p) \ []$, where
Efficiency

\[
\text{op } p \ x \ xs = \text{if } p \ x \ \text{then } x:xs \ \text{else } xs
\]

Now

\[
(\text{op } p \ x \ ys, \text{op } (\text{not } . \ p) \ x \ zs) \\
= \text{if } p \ x \ \text{then } (x:ys,zs) \ \text{else } (ys,x:zs)
\]

Hence

\[
\text{partition } p \ xs = \text{foldr } f (\text{[]}, \text{[]}) \ xs
\]
where \( f \ x \ (ys,zs) = \text{if } p \ x \)
then \((x:ys,zs)\)
else \((ys,x:zs)\)

For the last part we obtain

\[
\text{partition } p \ xs = \text{part } p \ xs \ [\text{[]}] \ [\text{[]}]
\]
\[
\text{part } p \ [\text{[]}] \ ys \ zs = (ys,zs)
\]
\[
\text{part } p \ (x:xs) \ ys \ zs = \text{if } p \ x
\]
then \part p xs (x:ys) zs
else \part p xs ys (z:zs)

**Answer to Exercise I**

Remember that \( T \) estimates the *worst case* running time. The worst case for \text{labels} arises when every right subtree of the tree is a leaf. Then we have

\[
T(\text{labels})(n) = T(\text{labels})(n-1) + \Theta(n),
\]

where \( \Theta(n) \) accounts for the time to concatenate a list of length \( n-1 \) with a list of length 1. Hence

\[
T(\text{labels})(n) = \sigma_{j=0}^{n} \Theta(n) = \Theta(n^2).
\]

The accumulating parameter method yields

\[
\text{labels } t = \text{labels2 } t \ [\text{[]}]
\]
\[
\text{labels2 } \text{(Leaf } x) \ xs = x:xs
\]
\[
\text{labels2 } \text{(Fork } u \ v) \ xs = \text{labels2 } u \ (\text{labels2 } v \ xs)
\]

and \( T(\text{labels2})(n) = \Theta(n) \). This improves the running time of \text{labels} from quadratic to linear time.

The induction step in the proof that \text{labels} \ (\text{build } xs) = xs \ is to assume the
hypothesis for all lists strictly shorter than \( xs \):

\[
\text{labels (build } xs \text{)} \equiv \{ \text{assume } xs \text{ has length at least two } \}
\]

\[
\text{labels (Fork (build } ys \text{) (build } zs \text{))} \equiv \{ \text{definition of labels} \}
\]

\[
\text{labels (build } ys \text{) ++ labels (build } zs \text{)} \equiv \{ \text{induction, since } ys \text{ and } zs \text{ are strictly shorter than } xs \}
\]

\[
y \text{ ++ } zs \equiv \{ \text{definition of halve } xs \} \equiv xs
\]

The induction here is general induction: in order to prove \( P(xs) \) for all finite lists \( xs \) it is sufficient to prove that: (i) \( P([],) \); and (ii) \( P(xs) \) holds under the assumption that \( P \) holds for all lists of length strictly less than \( xs \).

**Answer to Exercise J**

One key property is that

\[
(xs ++ [x] ++ ys)!!k | k < n \equiv xs!!k
\]

\[
| k==n \equiv x
\]

\[
| k > n \equiv ys!!(n-k)
\]

where \( n = \text{length } xs \)

The other key property is that sorting a list does not change the length of the list. Hence

\[
\text{select } k \ [] \equiv \text{error "list too short"}
\]

\[
\text{select } k \ (x:xs) | k < n \equiv \text{select } k \ ys
\]

\[
| k==n \equiv x
\]

\[
| \text{otherwise } \equiv \text{select } (n-k) \ zs
\]

where \( ys = [y | y <- xs, y < x] \)

\[
zs = [z | z <- xs, x <= z]
\]

\[
n \equiv \text{length } ys
\]

The worst-case running time for a list of length \( n \) occurs when \( k = 0 \) and the length of \( ys \) is \( n-1 \), i.e. when \( x:xs \) is in strictly decreasing order. Thus

\[
T(\text{select})(0,n) = T(\text{select})(0,n-1) + \Theta(n),
\]

with solution \( T(\text{select})(0,n) = \Theta(n^2) \). But, assuming a reasonable distribution
in which each permutation of the sorted result is equally likely as input, we have

\[ T(\text{select})(k,n) = \Theta(n). \]

### 7.10 Chapter notes

There are many books on algorithm design, but two that concentrate on functional programming are *Algorithms: A Functional Programming Approach* (second edition) (Addison-Wesley, 1999) by Fethi Rabbi and Guy Lapalme, and my own *Pearls of Functional Algorithm Design* (Cambridge, 2010).

This chapter is devoted to an example of how to build a small library in Haskell. A library is an organised collection of types and functions made available to users for carrying out some task. The task we have chosen to discuss is **pretty-printing**, the idea of taking a piece of text and laying it out over a number of lines in such a way as to make the content easier to view and understand. We will ignore many of the devices for improving the readability of a piece of text, devices such as a change of colour or size of font. Instead we concentrate only on where to put the line breaks and how to indent the contents of a line. The library won’t help you to lay out bits of mathematics, but it can help in presenting tree-shaped information, or in displaying lists of words as paragraphs.

### 8.1 Setting the scene

Let’s begin with the problem of displaying conditional expressions. In this book we have used three ways of displaying such expressions:

```haskell
if p then expr1 else expr2
```

```haskell
if p then expr1
else expr2
```

```haskell
if p
then expr1
else expr2
```

These three layouts, which occupy one, two or three lines, respectively, are considered acceptable, but the following two are not:
The decision as to what is or is not acceptable is down to me, the author. You may disagree with my choices (some do), and a flexible library should provide you with the ability to make your own reasonable choices. In any case, two basic questions have to be answered. Firstly, how can we describe the acceptable alternatives while rejecting the unacceptable ones? Secondly, how do we choose between the acceptable alternatives?

A quick answer to the second question is that the choice depends on the permitted line width. For instance we might choose a layout with the fewest lines, subject to the condition that each line fits within the allotted line width. Much more on this later.

As to the first question, one answer is just to write out all the acceptable alternatives. That’s going to involve a lot of writing. A better alternative is to provide the user with a suitable layout description language. As a rough and ready guide we might write something like

\[
\text{if } p <0> \text{ then } \text{expr1} ( <0> + <1>) \text{ else } \text{expr2 + if } p <1> \text{ then } \text{expr1 <1> else expr2}
\]

where \(<0>\) means a single space, \(<1>\) means a line break and \(+\) means ‘or’. The expression above yields our three layouts described earlier. However, the danger with providing the user with an unfettered choice of alternatives is that it becomes difficult to make a decision about the best layout without exploring every possible alternative, and that could take a long time.

Another possibility is to allow only restricted choice by forcing the user to describe layouts in terms of certain functions and operations provided by the library. For example, consider the description

\[
\text{group (group (if } p <1> \text{ then expr1) <> } <1> \text{ else expr2)}
\]

where group augments a set of layouts with one additional layout in which every \(<1>\) is replaced by \(<0>\), thereby flattening the layout to just one line, and \((<>\) means concatenation lifted to sets of alternatives. For example,

\[
\text{group (if } p <1> \text{ then expr1)}
\]

\[
= \{\text{if } p <0> \text{ then expr1, if } p <1> \text{ then expr1}\}
\]
Thus our set of three acceptable layouts is captured by the above description which contains two occurrences of group.

There is another aspect to the problem of displaying conditional expressions. What if expr1 or expr2 are themselves conditional expressions? Here we might want to allow a layout like

\[
\begin{align*}
\text{if } p \\
\quad \text{then if } q \\
\quad \quad \text{then expr1} \\
\quad \quad \text{else expr2} \\
\quad \text{else expr3}
\end{align*}
\]

The point is that we should allow for indentation in our description language. Indentation means putting in a suitable number of spaces after each line break. This idea can be captured by providing a function nest so that nest \( i \) \( x \) is a layout in which each line break in layout \( x \) is followed by \( i \) spaces.

8.2 Documents

For the sake of a name let us agree to call a document some entity that represents the set of possible layouts of a piece of text. Documents are given as elements of the type Doc whose definition is left for later on. On the other hand, a layout is simply a string:

\[
\text{type Layout} \ = \ \text{String}
\]

We are deliberately being cagey about what a document actually is because we want to consider two representations of Doc. For now we concentrate on the operations on documents that our library might provide.

The first operation is a function

\[
\text{pretty} \ :: \ \text{Int} \to \ \text{Doc} \to \ \text{Layout}
\]
that takes a given line width and a document, and returns the best layout. How to
define this function efficiently is really the main concern of the chapter.

The second operation is a function

\[
\text{layouts} :: \text{Doc} \rightarrow [\text{Layout}]
\]

that returns the set of possible layouts as a list. Why should we want such a func-
tion when we have \texttt{pretty}? Well, it takes a little experimentation to find the de-
finitions that describe the layouts we regard as acceptable. The way to experiment is
to formulate an initial definition and then rework it after inspecting all the result-
ing layouts on a small number of examples. That way we can see whether some
layouts should be excluded or others added. So, whatever our final representation
of documents turns out to be, we should provide \texttt{layouts} as a sensible diagnostic
tool for the user.

The remaining operations deal with constructing documents. First up is the opera-
tion of concatenating two documents to give a new one:

\[
(\langle\rangle) :: \text{Doc} \rightarrow \text{Doc} \rightarrow \text{Doc}
\]

Document concatenation should surely be an associative operation so we require
of any implementation of \(\langle\rangle\) that

\[(x \langle\rangle y) \langle\rangle z) = x \langle\rangle (y \langle\rangle z)\]

for all documents \(x, y\) and \(z\).

Whenever there is an associative operation there is usually an identity element, so
we also provide an empty document

\[
\text{nil} :: \text{Doc}
\]

We require \(\text{nil} \langle\rangle x = x\) and \(x \langle\rangle \text{nil} = x\) for all documents \(x\).

The next operation is a function

\[
\text{text} :: \text{String} \rightarrow \text{Doc}
\]

that takes a string not containing newlines into a document. To provide for docu-
ments containing more than one line, we can provide another basic document

\[
\text{line} :: \text{Doc}
\]

For example,

\[
\text{text "Hello" \langle\rangle line \langle\rangle text "World!"}
\]
is a document with a single layout that consists of two lines. You might think that
line is unnecessary because we could always allow newline characters in text
strings, but to indent a document we would then have to inspect the contents of
every text. Far better is to have an explicit newline document; that way we know
where line breaks are.

Next, the function

\[
\text{nest} :: \text{Int} \to \text{Doc} \to \text{Doc}
\]

provides a way of nesting documents: \text{nest} \ i \ indents a document by inserting
\ i \ spaces \ after \ every \ newline. Note the emphasis: indentation is not done at the
beginning of a document unless it begins with a newline. The reason for this choice
is explained below.

Finally, to complete a library of eight operations, we have the function

\[
\text{group} :: \text{Doc} \to \text{Doc}
\]

This is the function that produces multiple layouts. The function \text{group} takes a
document and adds an extra layout, one that consists of a single line of text with
no line breaks.

We have named eight operations and given informal descriptions of what they are
intended to mean, but can we be more precise about their properties and the re-
relationships between them? An even more fundamental question is whether these
operations are sufficiently flexible to allow for a reasonable class of layouts.

Let’s first concentrate on what equational laws we might want. Finding such laws
can boost our confidence that we have in hand an adequate and smoothly integrated
box of tools, and that there isn’t some crucial gadget we have missed. Such laws
can also influence the meanings of operations and guide implementations. We have
already asserted that \text{(<>)} should be associative with identity element \text{nil}, but
what else should we require?

Well, for text we want the following properties:

\[
\text{text} \ (s \ ++ \ t) = \text{text} \ s \ <\text{<} \ \text{text} \ t
\]
\[
\text{text} \ "" = \text{nil}
\]

In mathematical language this asserts that text is a \textit{homomorphism} from string
concatenation to document concatenation. An impressive (and possibly intimidat-
ing) name for something quite simple. Note that the associativity of string con-
catenation implies the associativity of document concatenation, at least for text
documents.
For `nest` we require the following equations to hold:

\[
\begin{align*}
nest i (x \shortleftarrow y) &= nest i x \shortleftarrow nest i y \\
nest i \text{ nil} &= \text{ nil} \\
nest i (\text{text } s) &= \text{ text } s \\
nest i \text{ line} &= \text{ line} \shortleftarrow \text{ text } (\text{replicate } i \ ' ') \\
nest i (nest j x) &= nest (i+j) x \\
nest 0 x &= x \\
nest i (\text{group } x) &= \text{ group } (nest i x)
\end{align*}
\]

All very reasonable (except possibly for the last), and we could give some of them mathematical names (`nest i` distributes through concatenation, `nest` is a homomorphism from numerical addition to functional composition and `nest i` commutes with `group`). The third law fails if `nest` were to indent from the beginning of a document; and it would also fail if we allowed text strings to contain newline characters. The last law holds because grouping adds a layout with no line breaks, and nesting has no effect on such a layout. See Exercise D for a more precise argument.

Turning to the properties of `layouts`, we require that

\[
\begin{align*}
\text{layouts } (x \shortleftarrow y) &= \text{ layouts } x \shortleftarrow \text{++ } \text{ layouts } y \\
\text{layouts } \text{ nil} &= ["""] \\
\text{layouts } (\text{text } s) &= [s] \\
\text{layouts } \text{ line} &= ["\n"] \\
\text{layouts } (nest i x) &= \text{ map } (nestl i) (\text{layouts } x) \\
\text{layouts } (\text{group } x) &= \text{ layouts } (\text{flatten } x) ++ \text{ layouts } x
\end{align*}
\]

The operation `(\shortleftarrow \text{++ })` is lifted concatenation:

\[
xss \shortleftarrow \text{++ } yss = [xs ++ ys | xs <- xss, ys <- yss]
\]

The function `nestl :: Int \rightarrow \text{Layout} \rightarrow \text{Layout}` is defined by

\[
nestl i = \text{concat } (\text{map } \text{indent } i) \\
\text{indent } i \text{ c } = \text{ if } c='\n' \text{ then } c:\text{replicate } i ' ' \text{ else } [c]
\]

Finally, `flatten :: Doc \rightarrow Doc` is the function that converts a document into one with a single layout in which each newline and its associated indentation is replaced by a single space. This function is not provided in the public interface of our documents library, though it will be needed internally. It is a missing gadget in the sense that we need it to complete the description of the algebraic laws.

We require that `flatten` should satisfy the following conditions:
8.3 A direct implementation

One obvious choice of representation is to identify a document with its list of layouts:

\[
\text{type \textit{Doc} = \text{[Layout]}}
\]

Such a representation is called a \textit{shallow embedding}. With a shallow embedding, the library functions are implemented directly in terms of the values of interest (here, \textit{layouts}). Later on we will abandon this representation in favour of a more structured alternative, but it is the obvious one to try first.

Here are the definitions of the operations above (we will leave \textit{pretty} until later):

\[
\begin{align*}
\text{layouts} & \quad = \text{id} \\
x \leftrightarrow y & \quad = x \leftrightarrow y \\
\text{nil} & \quad = [""] \\
\text{line} & \quad = ["\n"] \\
\text{text } s & \quad = [s] \\
\text{nest } i & \quad = \text{map (nestl } i) \\
\text{group } x & \quad = \text{flatten } x \leftrightarrow x \\
\text{flatten } x & \quad = [\text{flattenl } (\text{head } x)]
\end{align*}
\]

We have already defined \textit{nestl}, and \textit{flattenl} is defined by

\[
\begin{align*}
\text{flattenl } x & \quad = \text{flatten } x \leftrightarrow \text{flatten } y \\
\text{flattenl } \text{nil} & \quad = \text{nil} \\
\text{flattenl } \text{text } s & \quad = \text{text } s \\
\text{flattenl } \text{line} & \quad = \text{text } " " \\
\text{flattenl } \text{nest } i \ x & \quad = \text{flatten } x \\
\text{flattenl } \text{group } x & \quad = \text{flatten } x
\end{align*}
\]

That makes 24 laws in total (one for \textit{<>}, two each for \textit{nil} and \textit{text}, seven for \textit{nest} and six each for \textit{layouts} and \textit{flatten}). Many of the laws look like constructive Haskell definitions of functions over a data type in which \textit{nil}, \textit{text} and so on are constructors. More on this is in Section 8.6.

The eight operations certainly seem reasonable enough, but do they give us sufficient flexibility to describe the layouts we might want? The proof of the pudding is in the eating, so in a moment we will pause to consider three examples. Before doing so, some implementation of documents, however quick and dirty, will be needed to test the examples.
flattenl :: Layout -> Layout
flattenl [] = []
flattenl (c:cs)
  | c=='\n'   = ' ':flattenl (dropWhile (== ' ') cs)
  | otherwise = c:flattenl cs

Do the 24 laws hold for this implementation? Well, let's go through them. Lifted
congruentiation `<++>` is associative with `[[ ]]` as identity element, so the first three
laws are okay. The two laws of `text` are easy to check, and the six laws of `layouts`
are immediate. All but two laws of `nest` are routine. The remaining two, namely

\[
\text{nest } i \cdot \text{nest } j = \text{nest } (i+j)
\]
\[
\text{nest } i \cdot \text{group } = \text{group } \cdot \text{nest } i
\]

involve a bit of work (see Exercises C and D). That leaves the laws of `flatten`. Three are easy, and one can show

\[
\text{flatten } \cdot \text{nest } i = \text{flatten}
\]
\[
\text{flatten } \cdot \text{group } = \text{flatten}
\]

with a bit of work (see Exercises E and F). But the stumbling block is the law

\[
\text{flatten } (x <> y) = \text{flatten } x <> \text{flatten } y
\]

This one is false. Take \( x = \text{line} \) and \( y = \text{text } " \text{hello}" \). Then

\[
\text{flatten } (x <> y) = ["hello"]
\]
\[
\text{flatten } x <> \text{flatten } y = [" hello"]
\]

and the two results are different. The reason is that `flatten` removes the effect
of nesting, but does not remove spaces after newlines if they are present in an un-
nested document. On the other hand, `flattenl` removes spaces after every newline
in the document.

Rather than try to fix up this deficiency, we can accept the less than perfect imple-
mentation and move on. One can show that all layouts of a document flatten to the
same string (see the Answer to Exercise E). The shallow embedding also possesses
another property that we will exploit in the definition of `pretty`. To see what it is,
consider the function `shape` that returns the shape of a layout:

\[
\text{shape } :: \text{Layout } \rightarrow [\text{Int}]
\]
\[
\text{shape } = \text{map } \text{length } \cdot \text{lines}
\]

The prelude function `lines` breaks up a string on newline characters, returning a
list of strings without newlines. Thus the shape of a layout is the list of lengths of
the lines that make up the layout. The crucial property of `layouts` is that the list
of shapes of the layouts of a document is in lexicographically decreasing order. For example, one of the documents described in the following section has 13 possible layouts whose shapes are given by

\[
[94], [50, 43], [50, 28, 19], [50, 15, 17, 19], [10, 39, 43],
[10, 39, 28, 19], [10, 39, 15, 17, 19], [10, 28, 15, 17, 19],
[10, 28, 15, 28, 19], [10, 28, 15, 15, 17, 19],
[10, 13, 19, 15, 28, 19], [10, 13, 19, 15, 15, 17, 19]
\]

This list is in decreasing lexicographic order. The reason the property holds is that layout x puts the flattened layout at the head of the list of layouts of document x, and a flattened layout consists of a single line. Exercise G goes into more details.

8.4 Examples

Our first example deals with laying out conditional expressions. For present purposes a conditional expression can be represented as an element of the data type CExpr, where

\[
data CExpr = CExpr \mid Expr String \mid If String CExpr CExpr
\]

Here is a function cexpr that specifies the acceptable layouts described earlier:

\[
cexpr :: CExpr \rightarrow Doc
cexpr (Expr p) = text p
\]

\[
cexpr (If px y)
\]

\[
= group (group (text "if " <> text p <>
\]

\[
line <> text "then " <>
\]

\[
nest 5 (cexpr x) <>
\]

\[
line <> text "else " <>
\]

\[
nest 5 (cexpr y))
\]

This definition is similar to our previous version, except for the nesting of the subexpressions.

For example, two of the 13 possible layouts for one particular expression are as follows:

\[
if \text{wealthy}
\]

\[
then \text{if happy then lucky you else tough}
\]

\[
else \text{if in love then content else miserable}
\]
if wealthy
   then if happy
       then lucky you
       else tough
   else if in love
       then content
       else miserable

You can see from the last expression why we have chosen an indentation of five spaces. The 13 possible layouts for this particular conditional expression have the shapes displayed in the previous section.

The second example concerns how to lay out general trees, trees with an arbitrary number of subtrees:

\[
data \text{ GenTree } a = \text{ Node } a \ [\text{GenTree } a]
\]

Here is an example tree, laid out in two different ways:

\[
\text{Node 1}
\quad \text{[Node 2}
\quad \text{[Node 7 []}
\quad \text{,}
\quad \text{Node 8 []]}
\quad ,}
\text{Node 3}
\quad \text{[Node 9}
\quad \text{[Node 10 []}
\quad \text{,}
\quad \text{Node 11 []]]}
\quad ,}
\text{Node 4 []}
\quad ,}
\text{Node 5}
\quad \text{[Node 6 []]]}
\]

\[
\text{Node 1}
\quad \text{[Node 2 [Node 7 [], Node 8 []],}
\quad \text{Node 3 [Node 9 [Node 10 [], Node 11 []]],}
\quad \text{Node 4 [],}
\quad \text{Node 5 [Node 6 []]]}
\]

The function \texttt{gtree} that produced these trees (coincidentally, also among a total of 13 different ways) was defined as follows:

\[
gtree :: \text{Show } a \Rightarrow \text{GenTree } a \rightarrow \text{Doc}
gtree \ (\text{Node } x [])
   = \text{text ("Node " ++ show x ++ " []")}
\]
8.5 The best layout

\[
gtree (Node x ts) \\
= \text{text ("Node " ++ show x) <>} \\
\text{group (nest 2 (line <> bracket ts))}
\]

The first clause says that a tree with no subtrees is always displayed on a single line; the second clause says that a tree with at least one subtree is displayed either on a single line or has its subtrees each displayed on a new line with an indentation of two units. The function \text{bracket} is defined by

\[
bracket :: \text{Show a} \Rightarrow \text{[GenTree a]} \rightarrow \text{Doc} \\
bracket ts = \text{text "[" <> nest 1 (gtrees ts) <> text "]"}
\]

\[
gtrees [t] = gtree t \\
gtrees (t:ts) = gtree t <> \text{text "," <> line <> gtrees ts}
\]

To be honest, it took a little time and experimentation to find the definitions above (for which the function \text{layouts} proved indispensable), and the result is certainly not the only way to lay out trees.

Finally, here is a way of laying out a piece of text (a string of characters containing spaces and newlines, not a document text) as a single paragraph:

\[
\text{para} :: \text{String} \rightarrow \text{Doc} \\
\text{para} = \text{cvt} \cdot \text{map text} \cdot \text{words}
\]

\[
\text{cvt} [] = \text{nil} \\
\text{cvt} (x:xs) \\
= x <> \text{foldr (<>)} \text{nil [group (line <> x) | x <- xs]}
\]

First, the words of the text are computed using the standard library function \text{words}, a function we have encountered a number of times before. Then each word is converted into a document using \text{text}. Finally, each word, apart from the first, is laid out either on the same line or on a new line. If there are \(n+1\) words in the text, and so \(n\) inter-word spaces, the code above describes \(2^n\) possible layouts. We certainly don’t want to examine all these layouts in computing one that will fit within a given line width.

8.5 The best layout

As we said above, the best layout depends on the maximum permitted line width. That’s a simple decision, but not the only one. In general a pretty layout of a nested document will consist of a ribbon of text snaking across the page, and it is arguable
that the width of the ribbon should also play a part in determining the best layout. After all, is the best layout on an infinitely wide page one in which everything is placed on one line? However, for simplicity we will ignore this very reasonable refinement and take only the line width as the deciding factor.

There is also another decision to be made. Suppose we choose the best layout, according to some criterion, among those layouts all of whose lines fit within the given line width. That’s fine if there is at least one such layout, but what if there isn’t? The two options are either to abandon the formatting process with a suitable error message, or else to do the best we can, accepting that the width may be exceeded.

Psychologically and practically the second option seems the better one, so let us explore what it entails. We can start by comparing the first lines, $\ell_1$ and $\ell_2$, of two layouts. We can decide that line $\ell_1$ is better than $\ell_2$ if: (i) both lines fit into width $w$ and $\ell_1$ is longer than $\ell_2$; (ii) $\ell_1$ fits $w$ but $\ell_2$ doesn’t; or (iii) neither fits $w$ and $\ell_1$ is shorter than $\ell_2$. The decision is a reasonable one because it should be capable of being implemented by a greedy strategy: fill up the first line as much as possible without exceeding the line width; and if that is not possible, stop as soon as the width is exceeded.

The comparison test above doesn’t determine what should happen if the two lines have the same length. But it is a consequence of the fact that all layouts flatten to the same string that two first lines with the same length will be the same line. Consequently, the first line is fixed and the comparison can pass to the second pair of lines. And so on.

The second property about decreasing shapes can be used to simplify the comparison test slightly because if layout $l_x$ precedes layout $l_y$ in the list of layouts, then the first line of $l_x$ is known to be at least as long as the first line of $l_y$. And if the two lines are equally long, then the same statement is true of the second lines. And so on.

Given our shallow embedding of documents, here is a simple implementation of the function pretty that finds the best layout:

```haskell
pretty :: Int -> Doc -> Layout
pretty w = fst . foldr1 choose . map augment
  where
    augment lx = (lx,shape lx)
    choose alx aly
      = if better (snd alx) (snd aly) then alx else aly
    better [] ks = True
```
8.6 A term representation

better js [] = False
better (j:js) (k:ks) | j == k = better js ks
| otherwise = (j <= w)

Each layout is augmented with shape information to guide the choice of layout, which is then determined by a simple search. The test better implements the comparison operation described above. Finally, shape information is discarded.

This definition of pretty is hopelessly inefficient because every layout is computed and examined. If there are \( n \) possible choices of whether to have a line break or not, there are \( 2^n \) layouts to be examined and pretty-printing will be very slow indeed. For example,

ghci> putStrLn $ pretty 30 $ para pg
This is a fairly short paragraph with just twenty-two words. The problem is that pretty-printing it takes time, in fact 31.32 seconds.
(31.32 secs, 17650013284 bytes)

Ouch! What is worse, pretty-printing a longer paragraph will cause GHCi to crash with an ‘out of memory’ message. An exponential time and space algorithm is not acceptable.

What is wanted is an algorithm for pretty that can decide on which first line to choose without looking ahead more than \( w \) characters. The algorithm should also be efficient, taking linear time in the size of the document being pretty-printed. Ideally the running time should be independent of \( w \), but a running time that does depend on \( w \) is acceptable if a faster one means a much more complicated program.

8.6 A term representation

The problem with identifying a document with its list of possible layouts is that useful structure is lost. Rather than bring all the alternatives to the top level as a list, we really want to bury them as deep as possible. For example, consider the following two expressions for a document:

\[
\begin{align*}
A<0>B<0>D & + A<0>B<1>D + A<1>C<0>E + A<1>C<1>E \\
A(<0>B(<0>D + <1>D) + <1>C(<0>E + <1>E))
\end{align*}
\]
As before, $<0>$ denotes a single space and $<1>$ a single line break. The five letters denote five nonempty texts. Since all four alternatives have to flatten to the same document, we require that $B<0>D = C<0>E$. In the first expression (which is essentially what is given by representing a document by its list of layouts) we have four layouts to compare. In the second expression we can shortcut some of the comparisons. For example, if we know that the common prefix $A$ cannot fit in the given width, the first two layouts can be thrown away without further comparisons. Even better, if we choose between alternatives from the innermost to the outermost, we can base the comparison test on just the first lines of layouts. For instance, if we choose the better of $C<0>E$ and $C<1>E$ first, then that choice is not changed by subsequent choices.

The way to maintain the structure of documents is to represent a document as a tree:

```haskell
data Doc = Nil
  | Line
  | Text String
  | Nest Int Doc
  | Group Doc
  | Doc :<>: Doc
```

Note the use of an infix constructor in the last line. Haskell allows infix operators as constructors, but they have to begin with a colon. They do not have to end with a colon as well, but it seems more attractive if they do. This tree is called an abstract syntax tree; each operation of the library is represented by its own constructor. An implementation in terms of abstract syntax trees is known as a deep embedding.

We will not provide the user with the details of the data type `Doc`, just its name. To explain why not, it is useful to insert a short digression about Haskell data types. In Haskell the effect of a `data` declaration is to introduce a new data type by describing how its values are constructed. Each value is named by an expression built only from the constructors of the data type, in other words a term. Moreover, different terms denote different values (provided there are no strictness flags). We can define functions on the data type by pattern matching on the constructors. There is therefore no need to state what the operations on the data type are – we can just define them. Types in which the values are described, but the operations are not, are called concrete types.

The situation is exactly the reverse with abstract data types. Here the operations are named, but not how the values are constructed, at least not publicly. For example, `Float` is an abstract data type; we are given the names of the primitive arithmetic
and comparison operations, and also a way of displaying floating-point numbers, but it is not stated how such numbers are actually represented. We cannot define functions on these numbers by pattern matching, but only in terms of the given operations. What can and should be stated publicly are intended meanings and the algebraic properties of the operations. However, Haskell provides no means for such descriptions beyond informal comments.

As it stands, Doc is a concrete type. But in our understanding of this type, different terms do not denote different values. For instance, we intend each constructor to be a replacement for the corresponding operation. Thus

\[
\begin{align*}
nil & = \text{Nil} \\
\text{line} & = \text{Line} \\
\text{text } s & = \text{Text } s \\
\text{nest } i \ x & = \text{Nest } i \ x \\
\text{group } x & = \text{Group } x \\
x \ <\> \ y & = x :<>: y
\end{align*}
\]

We also want to keep the algebraic properties of these operations, so equations such as

\[
\begin{align*}
(x :<>: y) :<>: z & = x :<>: (y :<>: z) \\
\text{Nest } i \ (\text{Nest } j \ x) & = \text{Nest } (i+j) \ x
\end{align*}
\]

should hold. But of course they do not. The solution is to use the module structure to hide the constructors of Doc from the user and insist only that the laws are ‘observably’ true. For instance we require

\[
\text{layouts } ((x :<>: y) :<>: z) = \text{layouts } (x :<>: (y :<>: z))
\]

The only way we can observe documents is through layouts; from the user’s point of view if two documents produce the same layouts, then they are essentially the same document.

Let’s get back to programming. Here is one definition of layouts. It is just the laws of layouts that we saw earlier, but now expressed as a proper Haskell definition:

\[
\begin{align*}
\text{layouts } :: \text{Doc } \to \ [\text{Layout}] \\
\text{layouts } (x :<>: y) & = \text{layouts } x <++> \text{layouts } y \\
\text{layouts } \text{Nil} & = ["\""] \\
\text{layouts } \text{Line} & = ["\n"] \\
\text{layouts } (\text{Text } s) & = [s] \\
\text{layouts } (\text{Nest } i \ x) & = \text{map } (\text{nestl } i) \ (\text{layouts } x) \\
\text{layouts } (\text{Group } x) & = \text{layouts } (\text{flatten } x) ++ \text{layouts } x
\end{align*}
\]
The function `flatten` is similarly defined by

```hs
flatten :: Doc -> Doc
flatten (x :<>: y) = flatten x :<>: flatten y
flatten Nil = Nil
flatten Line = Text " "
flatten (Text s) = Text s
flatten (Nest i x) = flatten x
flatten (Group x) = flatten x
```

With these definitions, our 24 laws are either true by definition, or are observably true in the sense above.

The definition of `layouts` is simple enough, but it is unnecessarily inefficient. There are two separate reasons why this is so. First, consider the function `egotist` defined by

```hs
egotist :: Int -> Doc
egotist n | n==0 = nil
          | otherwise = egotist (n-1) <> text "me"
```

The document `egotist n` is a very boring one, and its sole layout consists of a string of `n` repetitions of `me`. By the way, we could have expressed the definition using `Nil`, `(:<>:)` and `Text` but, as we have said, we are not going to make these constructors public. As it stands, the definition of `egotist` could have been made by a user of the library. Anyway, back to the main point, which is that the association of the `(<>)` operations is to the left, and it takes $\Theta(n^2)$ steps to compute its layout(s). The `(++)` operations pile up to the left. The situation is entirely analogous to the fact that `concat` defined in terms of `foldl` is an order of magnitude less efficient than one defined in terms of `foldr`.

The second source of inefficiency concerns nesting. For example, consider the function `egoist` defined by

```hs
egoist :: Int -> Doc
egoist n | n==0 = nil
          | otherwise = nest 1 (text "me" <> egoist (n-1))
```

There are no line breaks in sight, so `egoist n` describes the same boring document as `egotist n`. But although the concatenation associates to the right, it still takes quadratic time to construct the layout. Each nesting operation is carried out by running through the entire document. Try it and see.
The way to solve the first problem is to delay concatenation, representing a concatenated document by a list of its component documents. The way to solve the second problem is to delay nesting, representing a nested document by a pair consisting of an indentation to be applied only when necessary and the document it is to be applied to. Combining both solutions, we represent a document by a list of indentation-document pairs. Specifically, consider the function toDoc defined by

\[
\text{toDoc} :: \text{[(Int,Doc)]} \rightarrow \text{Doc}
\]

\[
\text{toDoc} \; \text{ids} = \text{foldr} \; (\langle\rangle:) \; \text{Nil} \; \text{[Nest} \; \text{i} \; \text{x} \mid (\text{i},\text{x}) \leftarrow \text{ids}]
\]

We can now calculate a definition of a function layr such that

\[
\text{layr} = \text{layouts} \cdot \text{toDoc}
\]

and then define a new version of layouts based on layr. We leave the details as an exercise, but here is the result:

\[
\begin{align*}
\text{layouts} \; \text{x} &= \text{layr} \; [(0,\text{x})] \\
\text{layr} \; [] &= [""] \\
\text{layr} \; ((\text{i},x :\langle\rangle: y):\text{ids}) &= \text{layr} \; ((\text{i},x):(\text{i},y):\text{ids}) \\
\text{layr} \; ((\text{i},\text{Nil}):\text{ids}) &= \text{layr} \; \text{ids} \\
\text{layr} \; ((\text{i},\text{Line}):\text{ids}) &= [\backslash\text{n}:\text{replicate} \; \text{i} \; ′ ′ ++ \text{ls} \\
& \quad | \text{ls} \leftarrow \text{layr} \; \text{ids}] \\
\text{layr} \; ((\text{i},\text{Text} \; \text{s}):\text{ids}) &= [\text{s} ++ \text{ls} \mid \text{ls} \leftarrow \text{layr} \; \text{ids}] \\
\text{layr} \; ((\text{i},\text{Nest} \; \text{j} \; \text{x}):\text{ids}) &= \text{layr} \; ((\text{i}+\text{j},\text{x}):\text{ids}) \\
\text{layr} \; ((\text{i},\text{Group} \; \text{x}):\text{ids}) &= \text{layr} \; ((\text{i},\text{flatten} \; \text{x}):\text{ids}) \quad \text{++} \\
& \quad \text{layr} \; ((\text{i},\text{x}):\text{ids})
\end{align*}
\]

This definition takes linear time for each layout. Exactly the same template is used for the function pretty, which chooses a single best layout:

\[
\begin{align*}
\text{pretty} \; w \; \text{x} &= \text{best} \; w \; [(0,\text{x})] \\
\text{where} \\
\text{best} \; r \; [] &= "" \\
\text{best} \; r \; ((\text{i},x :\langle\rangle: y):\text{ids}) &= \text{best} \; r \; ((\text{i},x):(\text{i},y):\text{ids}) \\
\text{best} \; r \; ((\text{i},\text{Nil}):\text{ids}) &= \text{best} \; r \; \text{ids} \\
\text{best} \; r \; ((\text{i},\text{Line}):\text{ids}) &= [\backslash\text{n}:\text{replicate} \; \text{i} \; ′ ′ ++ \\
& \quad \text{best} \; (w-\text{i}) \; \text{ids} \\
\text{best} \; r \; ((\text{i},\text{Text} \; \text{s}):\text{ids}) &= \text{s} ++ \text{best} \; (\text{r-length} \; \text{s}) \; \text{ids} \\
\text{best} \; r \; ((\text{i},\text{Nest} \; \text{j} \; \text{x}):\text{ids}) &= \text{best} \; r \; ((\text{i}+\text{j},\text{x}):\text{ids}) \\
\text{best} \; r \; ((\text{i},\text{Group} \; \text{x}):\text{ids}) &= \text{better} \; r \\
\quad & \quad (\text{best} \; r \; ((\text{i},\text{flatten} \; \text{x}):\text{ids})) \\
\quad & \quad (\text{best} \; r \; ((\text{i},\text{x}):\text{ids}))
\end{align*}
\]
The first argument of \texttt{best} is the remaining space available on the current line. This function is made local to the definition of \texttt{pretty} to avoid having to carry around the maximum line width \( w \) as an additional argument.

That leaves us with the problem of computing \texttt{better \( r \) \( l x \) \( l y \)}. Here we can make use of the fact that the first line of \( l x \) is guaranteed to be at least as long as the first line of \( l y \). Thus it suffices to compare the length of the first line of \( l x \) with \( r \). If the former fits within the latter, we choose \( l x \); otherwise we choose \( l y \). We therefore define

\[
\text{better \( r \) \( l x \) \( l y \) = if \text{fits} \( r \) \( l x \) then \( l x \) else \( l y \)}
\]

But we don’t want to compute the length of the whole of the first line of \( l x \) since that looks ahead too far. Instead, we take a more miserly approach:

\[
\text{fits} \( r \) _ | \( r < 0 \) = \text{False} \\
\text{fits} \( r \) [] = \text{True} \\
\text{fits} \( r \) (c:cs) = if c == '\'n' then \text{True} \\
\text{else fits} \( r-1 \) cs
\]

For exactly the same reason it is essential that the second and third arguments to \texttt{better} are computed lazily, that is, the two layouts are evaluated just enough to determine which is the better one, and no further.

Let’s revisit our troublesome paragraph:

\begin{verbatim}
ghci> putStrLn $ pretty 30 $ para pg
This is a fairly short paragraph with just twenty-two words. The problem is that pretty-printing it takes time, in fact 31.32 seconds.
(0.00 secs, 1602992 bytes)
\end{verbatim}

Much better. Exercise L discusses what we can say about the running time of \texttt{pretty}.

The final task is to put our small library together as a module. Here is the main declaration:

\begin{verbatim}
module Pretty
    (Doc, Layout, nil, line, text, nest, (<>), group, layouts, pretty, layout) where
\end{verbatim}
The module name is Pretty and the file containing the above declaration and the definitions of the library functions has to be saved in a file called Pretty.lhs.

The module exports 11 entities. Firstly, there is the name Doc of the abstract type of documents. The constructors of this type are not exported. (By the way, if we did want to export all the constructors we can write Doc () in the export list, and if we wanted just, say, Nil and Text, we can write Doc (Nil, Text).) Secondly, there is the name Layout which is just a synonym for String. The next eight constants and functions are the ones we have defined above. The final function layout is used for printing a layout:

```
layout :: Layout -> IO ()
layout = putStrLn
```

And that’s it. Of course, in a really useful library a number of additional combinators could be provided. For example, we could provide

```
(<<+>>), (<|>) :: Doc -> Doc -> Doc
x <<+>> y = x <> text " " <> y
x <|> y = x <> line <> y

spread, stack :: [Doc] -> Doc
spread = foldr (<<+>>) nil
stack = foldr (<|>) nil
```

No doubt the reader can think of many others.

8.7 Exercises

**Exercise A**

A picky user of the library wants just three layouts for a certain document:

```
A B C     A B     A
   C       B C
```

Can the user do it with the given functions?

**Exercise B**

The layouts of a document are given as a list. But are they all different? Either prove that they are or give a counterexample.
By the way, is it obvious from the laws that each document has a nonempty set of layouts?

**Exercise C**

The next four exercises refer to the shallow embedding of Section 8.3. Prove, by equational reasoning, that

\[ \text{nest } i \cdot \text{nest } j = \text{nest } (i + j) \]

You will need a subsidiary result about \text{nest}1, which you don’t have to prove.

**Exercise D**

Continuing on from the previous question, prove that

\[ \text{nest } i \ (\text{group } x) = \text{group } (\text{nest } i \ x) \]

by equational reasoning (at the point level). Again, you will need a subsidiary result.

**Exercise E**

Continuing on, prove that \text{flatten} \cdot \text{group} = \text{flatten}. You will need a subsidiary result.

**Exercise F**

The final law is \text{flatten} \cdot \text{nest } i = \text{flatten}. And, yes, you will need yet another subsidiary result.

**Exercise G**

We said in the text that the prelude function \text{lines} breaks up a string on newline characters. In fact, \text{lines} treats a newline as a terminator character, so both \text{lines } "hello" and \text{lines } "hello\n" return the same result. It is arguable that a better definition treats newlines as separator characters, so there is always one more line than there are newlines. Define a function \text{lines} that has this behaviour. We will need the new definition below.

Now, the proof that \text{map shape} applied to the layouts of a document returns a lexicographically decreasing sequence of list of integers can be structured into the following steps. First, define

\[
\begin{align*}
\text{msl} & = \text{map shape} \cdot \text{layouts} \\
\text{shape} & = \text{map length} \cdot \text{lines}
\end{align*}
\]
8.7 Exercises

where lines refers to the revised version above. We have to prove that msl returns a decreasing sequence on every document. To this end, we can define functions nesty and groupy so that

\[
\text{nesty } i \ . \ \text{msl} = \text{msl} \ . \ \text{nest } i
\]
\[
\text{groupy } . \ \text{msl } = \text{msl} \ . \ \text{group}
\]

and an operation \(<+>\) so that

\[
\text{msl } x <+> \ \text{msl } y = \text{msl } (x <+> y)
\]

(It is this equation that requires the revised definition of lines.) The proof is then completed by showing that if xs and ys are decreasing, then so are nesty i xs and groupy xs and xs <+> ys. All this exercise asks though is that you construct definitions of nesty, groupy and <+>.

Exercise H

Write a function \textit{doc} :: \textit{Doc} \to \textit{Doc} that describes how to lay out elements of \textit{Doc} where \textit{Doc} is the abstract syntax tree representation in Section 8.6.

Exercise I

Consider a function prettybad that chooses a best layout from the list \textit{layouts} by taking the first layout all of whose lines fit within the given width, and the last layout if this is not possible. Does prettybad always compute the same layout as pretty? (Hint: think about paragraphs.)

Exercise J

Using the algebraic properties of the constructors of \textit{Doc}, calculate the efficient version of \textit{layouts}.

Exercise K

We have designed \textit{pretty \_w} to be \textit{optimal}, meaning that it chooses line breaks to avoid overflowing lines if at all possible. We also have that \textit{pretty \_w} is \textit{bounded}, meaning that it can make the choice about the next line break without looking at more than the next \_w characters of the input. Given that, what do you expect GHCi’s response would be to the commands

\[
\text{layout } $ \text{pretty } 5 \ $ \ \text{para } \text{pg}
\]
\[
\text{layout } $ \text{pretty } 10 \ $ \ \text{cexpr } \text{ce}
\]

where
pg = "Hello World!" ++ undefined
ce = If "happy" (Expr "great") undefined

Exercise L

We cannot relate the cost of pretty \( w\ x\) to the size of \( x\) without saying what the size of a document is. Here is a reasonable measure:

\[
\begin{align*}
\text{size} &: \text{Doc} \rightarrow \text{Int} \\
\text{size} \ \text{Nil} &= 1 \\
\text{size} \ \text{Line} &= 1 \\
\text{size} \ \text{Text} \ s &= 1 \\
\text{size} \ \text{Nest} \ i \ x &= 1 + \text{size} \ x \\
\text{size} \ (x :<>: y) &= 1 + \text{size} \ x + \text{size} \ y \\
\text{size} \ \text{Group} \ x &= 1 + \text{size} \ x
\end{align*}
\]

Under this definition both the documents

\[
\begin{align*}
\text{nest} \ 20 \ \text{(line <> text "!"}) \\
\text{nest} \ 40 \ \text{(line <> text "!"})
\end{align*}
\]

have size two. But it takes twice as long to produce the second layout, so the cost of pretty cannot be linear in the document size.

Instead of having pretty produce the final layout, a string, we can interpose an additional data type of layouts:

\[
\begin{align*}
data \ \text{Layout} &= \text{Empty} \\
& \mid \text{String} \ s \ \text{Layout} \\
& \mid \text{Break} \ i \ \text{Int} \ \text{Layout}
\end{align*}
\]

and define \( \text{layout} : : \text{Layout} \rightarrow \text{String} \) by

\[
\begin{align*}
\text{layout} \ \text{Empty} &= "" \\
\text{layout} \ \text{String} \ s \ x &= s ++ \text{layout} \ x \\
\text{layout} \ \text{Break} \ i \ x &= '\n' : \text{replicate} \ i \ ' ' ++ \text{layout} \ x
\end{align*}
\]

We have

\[
\text{pretty} \ w \ = \ \text{layout} \ . \ \text{prettyl} \ w
\]

where the new function \( \text{prettyl} \) produces a Layout rather than a string. Define \( \text{prettyl} \).

A fairer question to ask is whether \( \text{prettyl} \ w \ x \) takes linear time in the size of \( x \). Does it?
Answer to Exercise A

No. There is no way of allowing both $A<0>B<1>C$ and $A<1>B<0>C$ without also having both of $A<0>B<0>C$ and $A<1>B<1>C$. These four are given by the expression

\[ \text{group (A <> line <> B) <> group (line <> C)} \]

Answer to Exercise B

The layouts of a document are not necessarily all different. For example

\[ \text{layouts (group (text "hello")) = ["hello","hello"]} \]

Yes, it is obvious that each document has a nonempty set of layouts. Look at the laws of layouts. The basic documents have a nonempty list of layouts and this property is preserved by the other operations.

Answer to Exercise C

The calculation is:

\[
\text{nest } i \cdot \text{nest } j
= \{ \text{definition of nest} \}
\text{map (nestl } i) \cdot \text{map (nestl } j)
= \{ \text{functor law of map} \}
\text{map (nestl } i \cdot \text{nestl } j)
= \{ \text{claim} \}
\text{map (nestl } (i+j))
= \{ \text{definition of nest} \}
\text{nest } (i+j)
\]

The claim is that $\text{nestl } i \cdot \text{nestl } j = \text{nestl } (i+j)$, which follows – after a short calculation – from

\[ \text{indent } (i+j) = \text{concat} \cdot \text{map (indent } i) \cdot \text{indent } j \]

We omit the proof.
Answer to Exercise D

We reason:

\[
\begin{align*}
\text{nest } i \ (\text{group } x) &= \{\text{definition of group}\} \\
\text{nest } i \ (\text{flatten } x \ ++ \ x) &= \{\text{since nest } i = \text{map (nestl } i)\} \\
\text{nest } i \ (\text{flatten } x) \ + \ x &= \{\text{claim}\} \\
\text{flatten (nest } i \ x) \ + \ x &= \{\text{definition of group}\} \\
\text{group (nest } i \ x) \\
\end{align*}
\]

The claim follows from

\[
\begin{align*}
\text{nest } i \ . \ \text{flatten} &= \{\text{since there are no newlines in flatten } x\} \\
\text{flatten} &= \{\text{since flatten } \ . \ \text{nest } i = \text{flatten (Exercise F)}\} \\
\text{flatten} \ . \ \text{nest } i \\
\end{align*}
\]

Answer to Exercise E

We reason:

\[
\begin{align*}
\text{flatten } \ . \ \text{group} &= \{\text{definition of flatten and group}\} \\
\text{one} \ . \ \text{flattenl} \ . \ \text{flattenl} \ . \ \text{head} &= \{\text{claim}\} \\
\text{one} \ . \ \text{flattenl} \ . \ \text{head} &= \{\text{definition of flatten}\} \\
\text{flatten} \\
\end{align*}
\]

The claim is that flattenl is idempotent:

\[
\text{flattenl} \ . \ \text{flattenl} = \text{flattenl}
\]

This follows because flattenl returns a layout with no newlines.
By the way, it is the idempotence of \texttt{flattenl} that ensures all layouts of a document flatten to the same string. The only function that introduces multiple layouts is \texttt{group}, whose definition is

\[
\text{group } x = \text{flatten } x ++ x
\]

We have therefore to show that flattening the first element of this list gives the same string as flattening the second element. Thus we need to show

\[
\text{flattenl . head . flatten} = \text{flattenl . head}
\]

This follows at once from the definition of \texttt{flatten} and the idempotence of the function \texttt{flattenl}.

**Answer to Exercise F**

We reason:

\[
\begin{align*}
\text{flatten . nest } i &= \{\text{definitions}\} \\
&= \text{one . flattenl . head . map (nestl } i) \\
&= \{\text{since head . map } f = f \cdot \text{head}\} \\
&= \text{one . flattenl . nestl } i \cdot \text{head} \\
&= \{\text{claim}\} \\
&= \text{one . flattenl . head} \\
&= \{\text{definition of flatten}\} \\
&\quad \text{flatten}
\end{align*}
\]

The claim is that \(\text{flattenl . nestl } i = \text{flattenl}\).

**Answer to Exercise G**

We can define

\[
\text{lines } xs = \begin{cases} 
\text{ys} & \text{null } zs \\
\text{ys:lines } (\text{tail } zs) & \text{otherwise}
\end{cases}
\]

where \((ys,zs) = \text{break } (=romosome˜\text{n}) \) \text{xs}

The function \texttt{groupy} is defined by

\[
\text{groupy : : } [[[\text{Int}] \rightarrow [[\text{Int}]]
\text{groupy } (xs:xss) = [\text{sum } xs + \text{length } xs - 1]:xs:xss
\]

The function \texttt{nesty} is defined by
nesty :: :: Int -> [[Int]] -> [[Int]]
nesty i = map (add i)
    where add i (x:xs) = x:[i+x | x <- xs]

The function (+) is defined by

(+): [[Int]] -> [[Int]] -> [[Int]]
xss <+> yss = [glue xs ys | xs <- xss, ys <- yss]
    where glue xs ys = init xs ++ [last xs + head ys] ++
                  tail ys

Answer to Exercise H

One possibility, which no doubt can be improved on:

doc :: Doc -> Doc
doc Nil          = text "Nil"
doc Line         = text "Line"
doc (Text s)    = text ("Text " ++ show s)
doc (Nest i x)  = text ("Nest " ++ show i) <+>
    group (nest 2 (line <> paren (doc x)))
doc (x :+: y)   = doc x <+> text " :<>: " <+>
    group (line <> nest 3 (doc y))
doc (Group x)   = text "Group " <+>
    group (nest 2 (line <> paren (doc x)))

    paren x = text "(" <+ nest 1 x <+ text ")"

Answer to Exercise I

No. Consider a paragraph whose longest word is one character longer than the line width. In this case, prettybad will lay out each word on a single line, while pretty will still fill lines with groups of words provided they fit. For example:

ghci> putStrLn $ pretty 11 $ para pg4
A lost and
lonely
hippopotamus
went into a
bar.
Answer to Exercise J

First we show \( \text{layouts } x = \text{layr } [(0,x)] \):

\[
\text{layr } [(0,x)] \\
= \{\text{definition of layr}\} \\
\text{layouts } (\text{toDoc } [(0,x)]) \\
= \{\text{definition of toDoc}\} \\
\text{layouts } (\text{Nest 0 x :<>: Nil}) \\
= \{\text{laws of Doc}\} \\
\text{layouts } x 
\]

It remains to give a recursive definition of \( \text{layr} \). We will just give two clauses:

\[
\text{toDoc } ((i,\text{Nest } j \ x):\text{ids}) \\
= \{\text{definition of toDoc}\} \\
\text{Nest } i \ (\text{Nest } j \ x) :<>: \text{toDoc } \text{ids} \\
= \{\text{laws}\} \\
\text{Nest } (i+j) \ x :<>: \text{toDoc } \text{ids} \\
= \{\text{definition of toDoc}\} \\
\text{toDoc } ((i+j \ x):\text{ids}) 
\]

Hence \( \text{layr } ((i,\text{Nest } j \ x):\text{ids}) = \text{layr } ((i+j \ x):\text{ids}) \). Next:

\[
\text{toDoc } ((i,\text{x}:<>:\text{y}):\text{ids}) \\
= \{\text{definition of toDoc}\} \\
\text{Nest } i \ (\text{x} :<>: \text{y}) <> \text{toDoc } \text{ids} \\
= \{\text{laws}\} \\
\text{Nest } i \text{ x :<>: Nest } i \text{ y :<>: toDoc } \text{ids} \\
= \{\text{definition of toDoc}\} \\
\text{toDoc } ((i,\text{x}):(i,\text{y}):\text{ids}) 
\]

Hence \( \text{layr } ((i,\text{x}:<>:\text{y}):\text{ids}) = \text{layr } ((i,\text{x}):(i,\text{y}):\text{ids}) \).

Answer to Exercise K

ghci> layout $ pretty 5 $ para pg
Hello
World1*** Exception: Prelude.undefined
Pretty-printing

ghci> layout $ pretty 10 $ cexpr ce
if happy
then great
else *** Exception: Prelude.undefined

Answer to Exercise L

The definition is

prettyl :: Int -> Doc -> Layout
prettyl w x = best w [(0,x)]
    where
    best r [] = Empty
    best r ((i,Nil):ids) = best r ids
    best r ((i,Line):ids) = Break i (best (w-i) ids)
    best r ((i,Text s):ids) = String s (best (r-length s) ids)
    best r ((i,Nest j x):ids) = best r ((i+j,x):ids)
    best r ((i,x :<>: y):ids) = best r ((i,x):(i,y):ids)
    best r ((i,Group x):ids) = better r
        (best r ((i,flatten x):ids))
        (best r ((i,x):ids))

where better is changed to read

    better r lx ly = if fits r (layout lx) then lx else ly

The number of steps required to evaluate better r is proportional to r and thus at most w.

Now, prettyl takes linear time if best does. The second argument of best is a list of indentation-document pairs, and we can define the size of this list by

    isize ids = sum [size x | (i,x) <- ids]

For each of the inner five clauses in the definition of best, the size decreases by 1. For instance

    isize ((i,x :<>: y):ids)
    = size (x :<> y) + isize ids
    = 1 + size x + size y + isize ids
    = 1 + isize ((i,x):(i,y):ids)

It follows that if we let T(s) denote the running time of best r on an input of size s, then T(0) = Θ(1) from the first clause of best, and T(s+1) = Θ(1) + T(s) for
each of the five inner clauses, and
\[ T(s+1) = \Theta(w) + \text{maximum} \left[ T(k) + T(s-k) | k \leftarrow [1..s-1] \right] \]
for the last clause. And now we can deduce that \( T(s) = \Theta(ws) \).

In conclusion, our algorithm for pretty is linear, though not independently of \( w \).

8.9 Chapter notes

We referred to pretty-printing as a library, but another name for it is an embedded domain specific language (EDSL). It is a language for pretty-printing documents embedded in the host language Haskell. Many people believe that the growing success of Haskell is due to its ability to host a variety of EDSLs without fuss.

The detailed material in this chapter has been based closely on work by Philip Wadler, see ‘A prettier printer’, Chapter 11 in The Fun of Programming in Cornerstones of Computing Series (Palgrave MacMillan, 2003). The main difference is that Wadler used an explicit alternation operator in the term representation of Doc (though it was hidden from the user) rather than the constructor Group. Jeremy Gibbons suggested that the latter was a better fit with the idea of a deep embedding.

An earlier functional pretty-printing library based on a different set of combinators was described by John Hughes, ‘The design of a pretty-printer library’, in Johan Jeuring and Erik Meijer, editors, Advanced Functional Programming, volume 925 of LNCS, Springer, 1995. Hughes’ library was later reworked by Simon Peyton Jones and installed as a Haskell library

\text{Text.PrettyPrint.HughesPJ}

Another pretty-printing library, in an imperative rather than functional style, was constructed 30 years ago by Derek Oppen, ‘Pretty-printing’. ACM Transactions on Programming Languages and Systems 2(4), 465–483, 1980 and is widely used as the basis of pretty-printing facilities in a number of languages. More recently, efficient pretty-printing algorithms in a functional style have been described by Olaf Chitil, ‘Pretty printing with lazy dequeues’, ACM Transactions on Programming Languages and Systems 27(1), 163–184, 2005, and by Olaf Chitil and Doaitse Swierstra, ‘Linear, bounded, functional pretty-printing’, Journal of Functional Programming 19(1), 1–16, 2009. These algorithms are considerably more complicated than the one described in the text.
Chapter 9

Infinite lists

We have already met infinite lists in Chapter 4 and even given an induction principle for reasoning about them in Chapter 6. But we haven’t really appreciated what can be done with them. In this chapter we want to explain in more detail exactly what an infinite list is, and how they can be represented by cyclic structures. We also describe another useful method for reasoning about infinite lists, and discuss a number of intriguing examples in which infinite and cyclic lists can be used to good effect.

9.1 Review

Recall that \([m..]\) denotes the infinite list of all integers from \(m\) onwards:

ghci> [..]
[1,2,3,4,5,6,7,\{Interrupted\}
ghci> zip [..] "hallo"
[(1,'h'),(2,'a'),(3,'l'),(4,'l'),(5,'o')]

It would take forever to print \([1..]\), so we interrupt the first computation. The second example illustrates a simple but typical use of infinite lists in finite computations.

In Haskell, the arithmetic expression \([m..]\) is translated into \texttt{enumFrom m}, where \texttt{enumFrom} is a method in the \texttt{Enum} class, and defined by

\[
\text{enumFrom :: Integer -> [Integer]}
\]

\[
\text{enumFrom \(m = m:\text{enumFrom (m+1)}\)}
\]
Thus \([m.]\) is defined as an instance of a recursively defined function. The computation makes progress because (: ) is non-strict in its second argument.

It is important to bear in mind that infinite lists in computing do not have the same properties as infinite sets do in mathematics. For example, in set theory

\[
\{ x \mid x \in \{1, 2, 3, \ldots\}, x^2 < 10 \}
\]

denotes the set \(\{1, 2, 3\}\), but

ghci> [x | x <- [1..], x*x < 10]
[1,2,3]

After printing the first three values the computer gets stuck in an infinite loop looking for the next number after 3 whose square is less than 10. The value of the expression above is the partial list \(1:2:3:undefined\).

It is possible to have an infinite list of infinite lists. For example,

\[
multiples = [map (n*) [1..] | n <- [2..]]
\]
defines an infinite list of infinite lists of numbers, the first three being

\[
[2, 4, 6, 8, \ldots] \quad [3, 6, 9, 12, \ldots] \quad [4, 8, 12, 16, \ldots]
\]

Suppose we ask whether the above list of lists can be merged back into a single list, namely \([2..]\). We can certainly merge two infinite lists:

\[
merge :: Ord a => [a] -> [a] -> [a]
merge (x:xs) (y:ys) | x<y = x:merge xs (y:ys)
| x==y = x:merge xs ys
| x>y = y:merge (x:xs) ys
\]

This version of \texttt{merge} removes duplicates: if the two arguments are in strictly increasing order, so is the result. Note the absence of any clauses of \texttt{merge} mentioning the empty list. Now it seems that, if we define

\[
mergeAll = foldr1 merge
\]

then \texttt{mergeAll multiples} will return the infinite list \([2..]\). But it doesn’t. What happens is that the computer gets stuck in an infinite loop attempting to compute the first element of the result, namely

\[
\text{minimum (map head multiples)}
\]

It is simply not possible to compute the minimum element in an infinite list. Instead we have to make use of the fact that \texttt{map head multiples} is in strictly increasing order, and define
mergeAll = foldr1 xmerge
xmerge (x:xs) ys = x:merge xs ys

With this definition, mergeAll multiples does indeed return [2..].

Finally, recall the induction principle described in Chapter 6 for proving facts about infinite lists. Provided $P$ is a chain-complete assertion, we can prove that $P(xs)$ holds for all infinite lists $xs$ by showing that: (i) $P(\text{undefined})$ holds; and (ii) $P(xs)$ implies $P(x:xs)$ for all $x$ and $xs$. Using this principle, we proved in Chapter 6 that $xs++ys = xs$ for all infinite lists $xs$. But it’s not immediately clear how induction can be used to prove, for example,

$$\text{map fact [0..]} = \text{scanl (*) 1 [1..]}$$

The obvious result to prove is

$$\text{map fact [0..n]} = \text{scanl (*) 1 [1..n]}$$

for all $n$, but can one then assert the first identity holds?

9.2 Cyclic lists

Data structures, like functions, can be defined recursively. For instance

ones :: [Int]
ones = 1:ones

This is an example of a cyclic list, a list whose definition is recursive. Contrast this definition with ones = repeat 1, where

repeat x = x:repeat x

This definition of ones creates an infinite, not a cyclic list. We could define

repeat x = xs where xs = x:xs

Now the function repeat is defined in terms of a cyclic list. The second definition (call it repeat2) is faster to evaluate than the first (call it repeat1) because there is less overhead:

ghci> last $ take 10000000 $ repeat1 1
1
(2.95 secs, 800443676 bytes)
ghci> last $ take 10000000 $ repeat2 1
1
As another example, consider the following three definitions of the standard prelude function \(\text{iterate}\):

\[
\begin{align*}
\text{iterate1} \ f \ x &= x:\text{iterate1} \ f \ (f \ x) \\
\text{iterate2} \ f \ x &= x \text{ where } x = x:\text{map} \ f \ x \\
\text{iterate3} \ f \ x &= x:\text{map} \ f \ (\text{iterate3} \ f \ x)
\end{align*}
\]

All three functions have type \(\text{iterate} : (a \rightarrow a) \rightarrow a \rightarrow [a]\) and produce an infinite list of the iterates of \(f\) applied to \(x\). The three functions are equal, but the induction principle reviewed earlier doesn’t seem to be applicable in proving this assertion because there is no obvious argument on which to perform the induction. More on this later. The first definition is the one used in the standard prelude, but it does not create a cyclic list. The second definition does, and the third is obtained from the second by eliminating the where clause. Assuming \(f \ x\) can be computed in constant time, the first definition takes \(\Theta(n)\) steps to compute the first \(n\) elements of the result, but the third takes \(\Theta(n^2)\) steps:

\[
\begin{align*}
\text{iterate3} \ (2*) \ 1 &= 1:\text{map} \ (2*) \ (\text{iterate3} \ (2*1)) \\
&= 1:2:\text{map} \ (2*) \ (\text{map} \ (2*) \ (\text{iterate3} \ (2*1)))) \\
&= 1:2:4:\text{map} \ (2*) \ (\text{map} \ (2*) \ (\text{map} \ (2*) \ (\text{iterate3} \ (2*1))))
\end{align*}
\]

Evaluating the \(n\)th element requires \(n\) applications of \((2*)\), so it takes \(\Theta(n^2)\) to produce the first \(n\) elements.

That leaves the second definition. Does it take linear or quadratic time? The evaluation of \(\text{iterate2} \ (2*) \ 1\) proceeds as follows:

\[
\begin{align*}
x &= 1:\text{map} \ (2*) \ x \\
&= 1:y \text{ where } y = \text{map} \ (2*) \ (1:y) \\
&= 1:2:z \text{ where } z = \text{map} \ (2*) \ (2:z) \\
&= 1:2:4:t \text{ where } t = \text{map} \ (2*) \ (4:t)
\end{align*}
\]

Each element of the result is produced in constant time, so \(\text{iterate2} \ (2*) \ 1\) takes \(\Theta(n)\) steps to produce \(n\) elements.

Let us now develop a cyclic list to generate an infinite list of all the primes. To start with we define

\[
\begin{align*}
\text{primes} &= [2..] \ \setminus \ \text{composites} \\
\text{composites} &= \text{mergeAll} \ \text{multiples} \\
\text{multiples} &= [\text{map} \ (n*) \ [n..] \mid n < [2..]]
\end{align*}
\]
where (\ \) subtracts one strictly increasing list from another:

\[
(x:xs) \setminus (y:ys) \mid 
\begin{align*}
& x < y \quad = x:(xs \setminus (y:ys)) \\
& x = y \quad = xs \setminus ys \\
& x > y \quad = (x:xs) \setminus ys
\end{align*}
\]

Here, multiples consists of the list of all multiples of 2 from 4 onwards, all multiples of 3 from 9 onwards, all multiples of 4 from 16 onwards, and so on. Merging the list gives the infinite list of all the composite numbers, and taking its complement with respect to [2..] gives the primes. We saw the definition of mergeAll in the previous section.

So far, so good. But the algorithm can be made many times faster by observing that too many multiples are being merged. For instance, having constructed the multiples of 2 there is no need to construct the multiples of 4, or of 6, and so on. What we really would like to do is just to construct the multiples of the primes. That leads to the idea of ‘tying the recursive knot’ and defining

\[
primes = [2..] \setminus \text{composites}
\]

where

\[
\text{composites} = \text{mergeAll} [\text{map} (p*) [p..] \mid p <- \text{primes}]
\]

What we have here is a cyclic definition of primes. It looks great, but does it work? Unfortunately, it doesn’t: primes produces the undefined list. In order to determine the first element of primes the computation requires the first element of composites, which in turn requires the first element of primes. The computation gets stuck in an infinite loop. To solve the problem we have to pump-prime (!) the computation by giving the computation the first prime explicitly. We have to rewrite the definition as

\[
primes = 2:([3..] \setminus \text{composites})
\]

where

\[
\text{composites} = \text{mergeAll} [\text{map} (p*) [p..] \mid p <- \text{primes}]
\]

But this still doesn’t produce the primes! The reason is a subtle one and is quite hard to spot. It has to do with the definition

\[
\text{mergeAll} = \text{foldr1} \text{xmerge}
\]

The culprit is the function foldr1. Recall the Haskell definition:

\[
\text{foldr1} :: (a -> a -> a) -> [a] -> a
\]

\[
\text{foldr1} \ f \ [x] \quad = x
\]

\[
\text{foldr1} \ f \ (x:xs) = f \ x \ (\text{foldr1} \ xs)
\]
The order of the two defining equations is significant. In particular,

\[
\text{foldr1 f (x:undefined)} = \text{undefined}
\]
because the list argument is first matched against \(x:[]\), causing the result to be undefined. That means

\[
\text{mergeAll [map (p*) [p..] | p <- 2:undefined]} = \text{undefined}
\]

What we wanted was

\[
\text{mergeAll [map (p*) [p..] | p <- 2:undefined]} = 4:\text{undefined}
\]

To effect this change we have to define \text{mergeAll} differently:

\[
\text{mergeAll (xs:xss)} = \text{xmerge xs (mergeAll xss)}
\]

Now we have

\[
\begin{align*}
\text{mergeAll [map (p*) [p..] | p <- 2:undefined]} &= \text{xmerge (map (2*) [2..]) undefined} \\
&= \text{xmerge (4:map (2*) [3..]) undefined} \\
&= 4:\text{merge (map (2*) [3..]) undefined} \\
&= 4:\text{undefined}
\end{align*}
\]

This version of \text{mergeAll} behaves differently on finite lists from the previous one. Why?

With this final change we claim that \text{primes} does indeed get into gear and produces the primes. But how can the claim be proved? To answer this question we need to know something about the semantics of recursively defined functions and other values in Haskell, and how infinite lists are defined as limits of their partial approximations.

### 9.3 Infinite lists as limits

In mathematics, certain values are defined as *limits* of infinite sequences of approximations of simpler values. For example, the irrational number

\[
\pi = 3.14159265358979323846\ldots
\]

can be defined as the limit of the infinite sequence of rational approximations

\[
3, 3.1, 3.14, 3.141, 3.1415, \ldots
\]
The first element of the sequence, 3, is a fairly crude approximation to $\pi$. The next element, 3.1, is a little better; 3.14 is better still, and so on.

Similarly, an infinite list can also be regarded as the limit of a sequence of approximations. For example, the infinite list $[1..]$ is the limit of the infinite sequence of partial lists

$\bot, \ 1:\bot, \ 1:2:\bot, \ 1:2:3:\bot, \ ...$

Again, the sequence consists of better and better approximations to the intended limit. The first term, $\bot$, is the undefined element, and thus a very crude approximation: it tells us nothing about the limit. The next term, $1:\bot$, is a slightly better approximation: it tells us that the limit is a list whose first element is 1, but says nothing about the rest of the list. The following term, $1:2:\bot$, is a little better still, and so on. Each successively better approximation is derived by replacing $\bot$ with a more defined value, and thus gives more information about the limit.

Here is another sequence of approximations whose limit is $[1..]$:

$\bot, \ 1:2:\bot, \ 1:2:3:4:\bot, \ 1:2:3:4:5:6:\bot, \ ...$

This sequence is a subsequence of the one above but it converges to the same limit.

Here is a sequence of approximations that does not converge to a limit:

$\bot, \ 1:\bot, \ 2:1:\bot, \ 3:2:1:\bot, \ ...$

The problem with this sequence is that it gives conflicting information: the second term says that the limit begins with 1. However, the third term says that the limit begins with 2, and the fourth term says that it begins with 3, and so on. No approximation tells us anything about the intended limit and the sequence does not converge.

It should not be thought that the limit of a sequence of lists is necessarily infinite. For example, the sequence

$\bot, \ 1:\bot, \ 1:[] , \ 1:[], \ ...$

in which every element after the first two is $[1]$, is a perfectly valid sequence with limit $[1]$. Similarly,

$\bot, \ 1:\bot, \ 1:2:\bot, \ 1:2:\bot, \ ...$

is a sequence with limit $1:2:\bot$. Finite and partial lists are limits of sequences possessing only a finite number of distinct elements.

The way to formalise the property that an infinite sequence of partial lists converges to a limit is to introduce the notion of an approximation ordering $\sqsubseteq$ on the
elements of each type. The assertion \( x \sqsubseteq y \) means that \( x \) is an approximation to \( y \).

The ordering \( \sqsubseteq \) will be reflexive (\( x \sqsubseteq x \)), transitive (\( x \sqsubseteq y \) and \( y \sqsubseteq z \) implies \( x \sqsubseteq z \)), and anti-symmetric (\( x \sqsubseteq y \) and \( y \sqsubseteq x \) implies \( x = y \)). However, it is not the case that every pair of elements have to be comparable by \( \sqsubseteq \). Thus \( \sqsubseteq \) is what is known as a partial ordering. Note that \( \sqsubseteq \) is a mathematical operator (like \( = \)), and not a Haskell operator returning boolean results.

The approximation ordering for numbers, booleans, characters and any other enumerated type, is defined by

\[
x \sqsubseteq y \equiv (x = \bot) \lor (x = y).
\]

The first clause says that \( \bot \) is an approximation to everything. In other words, \( \bot \) is the bottom element of the ordering. This explains why \( \bot \) is pronounced ‘bottom’. The value \( \bot \) is the bottom element of \( \sqsubseteq \) for every type. The above ordering is flat. With a flat ordering one either knows everything there is to know about a value, or one knows absolutely nothing.

The approximation ordering on the type \((a, b)\) is defined by \( \bot \sqsubseteq (x, y) \) and

\[
(x, y) \sqsubseteq (x', y') \equiv (x \sqsubseteq x') \land (y \sqsubseteq y').
\]

The occurrences of \( \sqsubseteq \) on the right refer to the orderings on the types \( a \) and \( b \), respectively. The ordering \( \sqsubseteq \) on \((a, b)\) is not flat, even when the component orderings are. For example, in \((\text{Bool}, \text{Bool})\) we have the following chain of distinct elements:

\[
\bot \sqsubseteq (\bot, \bot) \sqsubseteq (\bot, \text{False}) \sqsubseteq (\text{True}, \text{False}).
\]

Note that in Haskell the pair \((\bot, \bot)\) is distinct from \( \bot \):

\[
\text{ghci> let f (a,b) = 1}\\
\text{ghci> f (undefined,undefined)}\\
1\\
\text{ghci> f undefined}\\
*** Exception: Prelude.undefined
\]

The ordering \( \sqsubseteq \) on \([a]\) is defined by \( \bot \sqsubseteq xs \) and \((x:xs) \not\sqsubseteq [] \) and

\[
[] \sqsubseteq xs \equiv xs = [],\\
(x:xs) \sqsubseteq (y:ys) \equiv (x \sqsubseteq y) \land (xs \sqsubseteq ys).
\]

These equations should be read as an inductive definition of a mathematical assertion, not as a Haskell definition. The second condition says that \( [] \) approximates only itself, and the third condition says that \((x:xs)\) is an approximation to \((y:ys)\) if and only if \( x \) is an approximation to \( y \) and \( xs \) is an approximation to \( ys \). The first
occurrence of $\sqsubseteq$ on the right-hand side refers to the approximation ordering on the type $a$.

As two examples, we have

$$[1, \bot, 3] \sqsubseteq [1, 2, 3] \quad \text{and} \quad 1 : 2 \cdot \bot \sqsubseteq [1, 2, 3].$$

However, $1 : 2 :: \bot$ and $[1, \bot, 3]$ are not related by $\sqsubseteq$.

The approximation ordering for each type $T$ is assumed to have another property in addition to those described above: each chain of approximations $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ has to possess a limit which is also a member of $T$. The limit, which we denote by $\lim_{n \to \infty} x_n$, is defined by two conditions:

1. $x_n \sqsubseteq \lim_{n \to \infty} x_n$ for all $n$. This condition states that the limit is an upper bound on the sequence of approximations.

2. If $x_n \sqsubseteq y$ for all $n$, then $\lim_{n \to \infty} x_n \sqsubseteq y$. This condition states that the limit is the least upper bound.

The definition of the limit of a chain of approximations applies to every type. Partial orderings possessing this property are called complete, and every Haskell type is a complete partial ordering (CPO for short). In particular, the property, introduced in Chapter 6, of a mathematical assertion $P$ being chain complete can now be formalised as

$$\left( \forall n : P(x_n) \right) \Rightarrow P\left( \lim_{n \to \infty} x_n \right).$$

In words, $P$ holds in the limit if it holds for each approximation to the limit.

For lists there is a useful Haskell function `approx`, which produces approximations to a given list. The definition is

```haskell
approx :: Integer -> [a] -> [a]
approx n [] | n>0 = []
approx n (x:xs) | n>0 = x:approx (n-1) xs
```

The definition of `approx` is very similar to that of `take` except that, by case exhaustion, we have $\text{approx } 0 \text{ xs } = \text{undefined}$ for all `xs`. For example,

- `approx 0 [1] = undefined`
- `approx 1 [1] = 1:undefined`
- `approx 2 [1] = 1:[]`

The crucial property of `approx` is that

$$\lim_{n \to \infty} \text{approx } n \text{ xs } = \text{xs}$$
9.3 Infinite lists as limits

for all lists \(xs\), finite, partial or infinite. The proof, an induction on \(xs\), is left as an exercise.

It follows that if \(\text{approx } n \ xs = \text{approx } n \ ys\) for all natural numbers \(n\), then \(xs = ys\). Thus we can prove that

\[
\text{iterate } f \ x = x : \text{map } f \ (\text{iterate } f \ x)
\]

by showing

\[
\text{approx } n \ (\text{iterate } f \ x) = \text{approx } n \ (x : \text{map } f \ (\text{iterate } f \ x))
\]

for all natural numbers \(n\). And, of course, we can use induction over the natural numbers to establish this fact. The details are left as an easy exercise.

As another example, consider the value \(\text{primes}\) defined in the previous section. Suppose we define

\[
\text{prs } n = \text{approx } n \ \text{primes}
\]

We would like to show that \(\text{prs } n = p_1 : p_2 : \cdots p_n : \bot\), where \(p_j\) is the \(j\)th prime. We claim that

\[
\text{prs } n = \text{approx } n \ (2 : ([3..] \ \setminus \ \text{crs } n))
\]

\[
\text{crs } n = \text{mergeAll} \ \text{map} \ (p*) \ [p..] \ | \ p < - \ \text{prs } n
\]

Given this, it is sufficient to show that \(\text{crs } n = c_1 : c_2 : \cdots c_m : \bot\), where \(c_j\) is the \(j\)th composite number (so \(c_1 = 4\)) and \(m = p_n^2\). Then the proof is completed by using the fact that \(p_{n+1} < p_n^2\), which is a non-trivial result in Number Theory. Details are in the exercises.

*Computable functions and recursive definitions*

One can describe many functions in mathematics, but only some of them are computable. There are two properties of computable functions not shared by arbitrary functions. Firstly, a computable function \(f\) is *monotonic* with respect to the approximation ordering. In symbols,

\[
x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)
\]

for all \(x\) and \(y\). Roughly speaking, monotonicity states that the more information you supply about the argument, the more information you get as a result. Secondly, a computable function \(f\) is *continuous*, which means that

\[
f \left( \lim_{n \to \infty} x_n \right) = \lim_{n \to \infty} f(x_n)
\]
for all chains of approximations \( x_0 \sqsubseteq x_1 \sqsubseteq \ldots \). Roughly speaking, continuity states that there are no surprises on passing to the limit.

Continuity appears similar to chain completeness but differs in two respects. One is that the chain completeness of \( P \) does not imply the converse property that if \( P \) is false for all approximations, then \( P \) is false for the limit. In other words, it does not imply that \( \neg P \) is chain complete. Secondly, \( P \) is a mathematical assertion, not a Haskell function returning a boolean value.

Although we won’t prove it, every monotonic and continuous function \( f \) has a least fixed point. A fixed point of a function \( f \) is a value \( x \) such that \( f(x) = x \). And \( x \) is a least fixed point if \( x \sqsubseteq y \) for any other fixed point \( y \). The least fixed point of a monotonic and continuous function \( f \) is given by \( \lim_{n \to \infty} x_n \) where \( x_0 = \bot \) and \( x_{n+1} = f(x_n) \). In functional programming, recursive definitions are interpreted as least fixed points.

Here are three examples. Consider the definition \( \text{ones} = 1: \text{ones} \). This definition asserts that \( \text{ones} \) is a fixed point of the function \( (1:) \). Haskell interprets it as the least fixed point, so \( \text{ones} = \lim_{n \to \infty} \text{ones}_n \), where \( \text{ones}_0 = \bot \) and \( \text{ones}_{n+1} = 1: \text{ones}_n \). It is easy to see that \( \text{ones}_n \) is the partial list consisting of \( n \) ones, so the limit is indeed an infinite list of ones.

Second, consider the factorial function

\[
\text{fact } n = \text{if } n=0 \text{ then } 1 \text{ else } n \ast \text{fact } (n-1)
\]

We can rewrite this definition in the equivalent form

\[
\text{fact } = (\lambda n \to \text{if } n=0 \text{ then } 1 \text{ else } n \ast \text{fact } (n-1)) \text{ fact }
\]

Again, this definition asserts that \( \text{fact} \) is a fixed point of a function. Here we have

\[
\text{fact}_0 n = \bot
\]
\[
\text{fact}_1 n = \text{if } n=0 \text{ then } 1 \text{ else } \bot
\]
\[
\text{fact}_2 n = \text{if } n\leq1 \text{ then } 1 \text{ else } \bot
\]

and so on. The value of \( \text{fact}_k n \) is the factorial of \( n \) if \( n \) is less than \( k \), and \( \bot \) otherwise.

Finally, consider the list \( \text{primes} \) once again. Here we have

\[
\text{primes}_0 = \bot
\]
\[
\text{primes}_{n+1} = 2: ([3..] \ \backslash \ \\text{mergeAll} \ [\text{map } (p \ast) [p..] \ \mid \ p \leftarrow \text{primes}_n])
\]
It is not the case that primes\(_n\) = approx n primes. In fact,

- \(\text{primes}_1 = 2:⊥\)
- \(\text{primes}_2 = 2:3:⊥\)
- \(\text{primes}_3 = 2:3:5:7:⊥\)
- \(\text{primes}_4 = 2:3:5:7:\ldots:47:⊥\)

The partial list \(\text{primes}_2\) produces all the primes less than 4, \(\text{primes}_3\) all the primes less than 9, and \(\text{primes}_4\) all the primes less than 49. And so on.

9.4 Paper–rock–scissors

Our next example of infinite lists is entertaining as well as instructive. Not only does it introduce the idea of using potentially infinite lists to model a sequence of interactions between processes, it also provides another concrete illustration of the necessity for formal analysis.

The paper–rock–scissors game is a familiar one to children, though it is known by different names in different places. The game is played by two people facing one another. Behind their backs, each player forms a hand in the shape of either a rock (a clenched fist), a piece of paper (a flat palm) or a pair of scissors (two fingers extended). At a given instant, both players bring their hidden hand forward. The winner is determined by the rule 'paper wraps rock, rock blunts scissors, and scissors cut paper'. Thus, if player 1 produces a rock and player 2 produces a pair of scissors, then player 1 wins because rock blunts scissors. If both players produce the same object, then the game is a tie and neither wins. The game continues in this fashion for a fixed number of rounds agreed in advance.

Our objective in this section is to write a program to play and score the game. We begin by introducing the types

```haskell
data Move = Paper | Rock | Scissors

type Round = (Move,Move)
```

To score a round we define

```haskell
score :: Round -> (Int,Int)
score (x,y) | x `beats` y = (1,0)
| y `beats` x = (0,1)
| otherwise = (0,0)
```

where
Infinite lists

\[
\begin{align*}
\text{Paper `beats` Rock} &= \text{True} \\
\text{Rock `beats` Scissors} &= \text{True} \\
\text{Scissors `beats` Paper} &= \text{True} \\
_ `\text{beats}` _ &= \text{False}
\end{align*}
\]

Each player in the game will be represented by a certain strategy. For instance, one simple strategy is, after the first round, always to produce what the opposing player showed in the previous round. This strategy will be called copy. Another strategy, which we will call smart, is to determine a move by analysing the number of times the opponent has produced each of the three possible objects, and calculating an appropriate response based on probabilities.

We will consider the details of particular strategies, and how they can be represented, in a moment. For now, suppose the type `Strategy` is given in some way. The function

\[
\text{rounds} :: (\text{Strategy},\text{Strategy}) \rightarrow [\text{Round}]
\]

takes a pair of strategies and returns the infinite list of rounds that ensue when each player follows his or her assigned strategy. The function

\[
\text{match} :: \text{Int} \rightarrow (\text{Strategy},\text{Strategy}) \rightarrow (\text{Int},\text{Int})
\]

\[
\text{match} n = \text{total} \cdot \text{map score} \cdot \text{take} n \cdot \text{rounds}
\]

\[
\text{where total} \; rs = (\text{sum} \; (\text{map} \; \text{fst} \; rs), \text{sum} \; (\text{map} \; \text{snd} \; rs))
\]

determines the total score after playing a given number of rounds.

The instructive aspect of the game is how to represent strategies. We are going to consider two ways, calling them `Strategy1` and `Strategy2`. The obvious idea is to take

\[
\text{type} \; \text{Strategy1} = [\text{Move}] \rightarrow \text{Move}
\]

Here, a strategy is a function which takes the (finite) list of moves made by the opponent so far and returns an appropriate move for the subsequent round. For efficiency in processing lists, we suppose that the list of moves is given in reverse order, with the last move first.

For example, the `copy1` strategy is implemented by

\[
\text{copy1} :: \text{Strategy1}
\]

\[
\text{copy1} \; ms = \text{if null} \; ms \; \text{then Rock else head} \; ms
\]

The first move is an arbitrary choice of Rock. The second strategy `smart1` is implemented by
smart1 :: Strategy1
smart1 ms = if null ms then Rock
               else pick (foldr count (0,0,0) ms)

count :: Move -> (Int,Int,Int) -> (Int,Int,Int)
count Paper (p,r,s) = (p+1,r,s)
count Rock (p,r,s) = (p,r+1,s)
count Scissors (p,r,s) = (p,r,s+1)

pick :: (Int,Int,Int) -> Move
pick (p,r,s)
  | m < p = Scissors
  | m < p+r = Paper
  | otherwise = Rock
  where m = rand (p+r+s)

This strategy counts the number of times each move has been made, and uses the
results to pick a move. The value of rand applied to \( n \) is some integer \( m \) in the
range \( 0 \leq m < n \). (Note that rand is never applied to the same integer.) Thus the
choice of move depends on whether \( m \) falls in one of the three ranges

\[
0 \leq m < p \quad \text{or} \quad p \leq m < p+r \quad \text{or} \quad p+r \leq m < p+r+s.
\]

For example, if \( p \) is large, then Scissors will be chosen with high probability
(because scissors cuts paper); and if \( r \) is large, then Paper will be chosen with
high probability (because paper wraps rock); and so on.

To define rand we can make use of two functions in the library System.Random:

\[
\text{rand} :: \text{Int} \rightarrow \text{Int}
\]
\[
\text{rand} n = \text{fst} \$ \text{randomR} (0,n-1) (\text{mkStdGen} n)
\]

The function mkStdGen takes an integer and returns a random number generator,
likely to be different for different integers. The choice of argument to mkStdGen
is arbitrary, and we have simply chosen \( n \). The function randomR takes a range
\((a,b)\) and a random number generator, and returns a pseudo-random integer \( r \) in
the range \( a \leq r \leq b \) and a new random number generator.

We can now define rounds1:

\[
\text{rounds1} :: (\text{Strategy1},\text{Strategy1}) \rightarrow [\text{Round}]
\]
\[
\text{rounds1} (p1,p2)
  = \text{map head} \$ \text{tail} \$ \text{iterate} (\text{extend} (p1,p2)) []
\]
\[
\text{extend} (p1,p2) \text{rs} = (p1 \ (\text{map snd} \text{rs}),p2 \ (\text{map fst} \text{rs})):\text{rs}
\]
The function `extend` adds a new pair of moves to the front of the list of existing rounds, and `rounds1` generates the infinite list of rounds by repeatedly applying `extend` to the initially empty list. It is more efficient to add something to the front of a list than to the end, which is why we keep the list of moves in reverse order.

Nevertheless `rounds1` is inefficient. Suppose a strategy takes time proportional to the length of its input to compute the next move. It follows that `extend` takes $\Theta(n)$ steps to update a game of $n$ rounds with a new round. Therefore, it takes $\Theta(N^2)$ steps to compute a game of $N$ rounds.

For comparison, let’s consider another way we might reasonably represent strategies. This time we take

```haskell
type Strategy2 = [Move] -> [Move]
```

In the new representation, a strategy is a function that takes the potentially infinite list of moves made by the opponent and returns the potentially infinite list of replies. For example, the copy strategy is now implemented by

```haskell
copy2 :: Strategy2
copy2 ms = Rock:ms
```

This strategy returns Rock the first time, and thereafter returns just the move made by the opponent in the previous round. The smart strategy is reprogrammed as

```haskell
smart2 :: Strategy2
smart2 ms = Rock:map pick (stats ms)
    where stats = tail . scanl (flip count) (0,0,0)
```

The function `stats` computes the running counts of the three possible moves. This strategy, like `copy2`, is also efficient in that it produces each successive output with constant delay.

With this new model of strategies we can redefine the function `rounds`:

```haskell
rounds2 :: (Strategy2,Strategy2) -> [Round]
rounds2 (p1,p2) = zip xs ys
    where xs = p1 ys
          ys = p2 xs
```

Here, `xs` is the list of replies computed by the first player in response to the list `ys` which, in turn, is the list of replies made by the second player in response to the list of moves `xs`. Thus `rounds2` is defined by two cyclic lists and we are obliged to show that it does indeed generate an infinite list of well-defined moves. More on this below. If the two players do encapsulate legitimate strategies, then
rounds2 computes the first $n$ moves of the game in $\Theta(n)$ steps, assuming that both players compute each new move with constant delay. Thus the second method for modelling strategies leads to a more efficient program than the earlier one.

Unfortunately, there is a crucial flaw with the second representation of strategies: it offers no protection against someone who cheats! Consider the strategy

\[
\text{cheat } ms = \text{map } \text{trump } ms
\]

\[
\begin{align*}
\text{trump Paper} &= \text{Scissors} \\
\text{trump Rock} &= \text{Paper} \\
\text{trump Scissors} &= \text{Rock}
\end{align*}
\]

The first reply of cheat is the move guaranteed to beat the opponent’s first move; similarly for subsequent moves. To see that cheat cannot be prevented from subverting the game, consider a match in which it is played against copy2, and let $xs = \text{cheat } ys$ and $ys = \text{copy2 } xs$. The lists $xs$ and $ys$ are the limits of the two chains $\{xs_n | 0 \leq n\}$ and $\{ys_n | 0 \leq n\}$, where $xs_0 = \bot$ and $xs_{n+1} = \text{cheat } ys_n$, and $ys_0 = \bot$ and $ys_{n+1} = \text{copy2 } xs_n$. Now, we have

\[
\begin{align*}
xs_1 &= \text{cheat } \bot &= \bot \\
y_1 &= \text{copy2 } \bot &= \text{Rock: } \bot \\
x_2 &= \text{cheat } (\text{Rock: } \bot) &= \text{Paper: } \bot \\
y_2 &= \text{copy2 } \bot &= \text{Rock: } \bot \\
x_3 &= \text{cheat } (\text{Rock: } \bot) &= \text{Paper: } \bot \\
y_3 &= \text{copy2 } (\text{Paper: } \bot) &= \text{Rock: } \text{Paper: } \bot
\end{align*}
\]

Continuing in this way, we see that the limits of these sequences are indeed infinite lists of well-defined moves. Moreover, cheat always triumphs. Another cheating strategy is given by

\[
\text{devious :: Int } \rightarrow \text{Strategy2} \\
\text{devious } n \text{ ms} = \text{take } n \text{ (copy2 ms) } ++ \text{cheat } \text{(drop } n \text{ ms)}
\]

This strategy behaves like copy for $n$ moves then starts to cheat.

Can we find a way to protect against cheats? To answer this question, we need to take a closer look at what constitutes an honest strategy. Informally speaking, a strategy is honest if its first move is computed in the absence of any information about the opponent’s first move, the second move is computed without any information about the opponent’s second move, and so on. Moreover, each of these moves should be well-defined, given that the opponent’s moves are well-defined. More precisely, let $\text{wdf}(n, ms)$ denote the assertion that the first $n$ elements in the
Infinite lists

(possibly partial) list of moves $ms$ are well-defined. Then a strategy $f$ is honest if

$$wdf(n, ms) \Rightarrow wdf(n+1, f(ms))$$

for all $n$ and $ms$. It is easy to show that $copy2$ is honest. On the other hand, $cheat$ is not honest because $wdf(0, \bot)$ is true but $wdf(1, cheat \ \bot)$ is false. The strategy $dozy$, where

$$dozy \ ms = \text{repeat undefined}$$

is also dishonest according to this definition although it doesn’t actually cheat.

Having identified the source of criminal or lackadaisical behaviour, can we ensure that only honest strategies are admitted to the game? The answer is a qualified yes: although it is not possible for a mechanical evaluator to recognise cheating (in the same way that it is not possible to recognise $\bot$, or strategies that do not return well-defined moves), it is possible to define a function $police$ so that if $p$ is an honest player and $ms$ is an infinite sequence of well-defined moves, then $police \ p \ ms = p \ ms$. On the other hand, if $p$ is dishonest at some point, then the game ends at that point in $\bot$. Operationally speaking, $police$ works by forcing $p$ to return the first (well-defined!) element of its output before it gives $p$ the first element of its input. Similarly for the other elements. The definition is

$$police \ p \ ms = ms' \ \text{where} \ ms' = p \ (\text{synch} \ ms \ ms')$$

$$\text{synch} \ (x:xs) \ (y:ys) = (y \ `\text{seq}` \ x):\text{synch} \ xs \ ys$$

Recall from Chapter 7 that $x \ `\text{seq}` \ y$ evaluates $x$ before returning the value of $y$. The proof that this implementation meets its specification is rather involved, so we are not going into details. It follows from the above analysis that to prevent cheating we must rewrite the definition of $rounds2$ to read

$$rounds2 \ (p1,p2) = \text{zip} \ xs \ ys$$

$$\text{where} \ xs = police \ p1 \ ys$$

$$ys = police \ p2 \ xs$$

9.5 Stream-based interaction

In the paper–rock–scissors game we modelled interaction by a function that took an infinite list of moves and returned a similar list. The same idea can be used to provide a simple model of input–output interaction. It’s called stream-based interaction because infinite lists are also called streams. Haskell provides a function

$$\text{interact} :: \ ([\text{Char}]) \rightarrow \text{IO} \ ()$$
for interacting with the world. The argument to \texttt{interact} is a function that takes a potentially infinite list of characters from the standard input channel, and returns a potentially infinite list of characters to be typed on the standard output channel.

For example,

\begin{verbatim}
ghci> import Data.Char
ghci> interact (map toUpper)
hello world!
HELLO WORLD!
Goodbye, cruel world!
GOODBYE, CRUEL WORLD!
{Interrupted}
\end{verbatim}

We imported the library \texttt{Data.Char} to make \texttt{toUpper} available, and then created an interaction that capitalised each letter. Each time a line of input was typed (and echoed) the interaction produced the same line in capital letters. The process continues until we interrupt it.

We can also design an interactive program that terminates. For example,

\begin{verbatim}
interact (map toUpper . takeWhile (/= '.'))
\end{verbatim}

will interact as above but terminate as soon as a line containing a period is typed:

\begin{verbatim}
ghci> interact (map toUpper . takeWhile (/= '.'))
Goodbye. Forever
GOODBYE
\end{verbatim}

Finally, here is a stand-alone program that takes a literate Haskell file as input and returns a file in which all nonempty lines not beginning with > are removed. The remaining lines are modified by removing the > character, so the result is a legitimate .hs file (a Haskell script not using the literate style):

\begin{verbatim}
main  = interact replace
replace = unlines . map cleanup . filter code . lines
code xs = null xs || head xs == '>'
cleanup xs = if null xs then [] else tail xs
\end{verbatim}

The program is the computation associated with the identifier \texttt{main}, and there always has to be a definition associated with this name if we want to compile a program. The function \texttt{lines} splits a text into lines, and \texttt{unlines} reassembles the text by putting a single newline between lines. If we store the program in \texttt{lhs2hs.lhs}, we can compile it and then run it:
$ ghc lhs2hs.lhs
$ lhs2hs <myscript.lhs >myscript.hs

In the second line, the input is taken from `myscript.lhs` and the output is directed to `myscript.hs`.

Stream-based interaction was the main method for interacting with the outside world in early versions of Haskell. However, the model presented above is too simple for most practical purposes. In a serious application one wants to do other things than reading and printing characters to a screen. For example, one also wants to open and read files, to write to or delete files, and in general to interact with all the mechanisms that are available in the world outside the confines of a functional programming language. Interaction takes place in time, and the order in which events occur has to be managed correctly by the programmer. In the stream-based approach, this ordering of events is represented by the order of the elements in a list; in other words, it is represented in the data and not reflected primarily in the way the program is composed. In the following chapter we will consider another approach to interaction, indeed, a general method for writing programs that have to control an orderly sequence of events. In this approach, the order is made explicit in the way the program is composed.

### 9.6 Doubly-linked lists

We end with another application of cyclic lists. Imagine reading a book consisting of a nonempty list of pages. To navigate around the book we need some way of moving on to the next page and moving back to the previous page. Other navigation tools would be useful, but we’ll stick with these two. Here is an interactive session with a particularly boring book consisting of three pages:

```
ghci> start book
"Page 1"
ghci> next it
"Page 2"
ghci> prev it
"Page 1"
ghci> next it
"Page 2"
ghci> next it
"Page 3"
```
In GHCi the variable it is bound to the expression just typed at the prompt. We started a book and what was printed was the first page. We turned to the next page, and then returned to the previous one. The interesting question is what should happen when we turn to the next page after the last one. Should the navigation report an error, just deliver the last page again or go to the first page? Suppose we decide on the last alternative, namely that the next page after the last one should be the first page, and the previous page before the first one should be the last page. In other words, our book is an instance of a cyclic doubly-linked list.

Here is the relevant datatype declaration:

```haskell
data DList a = Cons a (DList a) (DList a)

elem :: DList a -> a
    elem (Cons a p n) = a

prev, next :: DList a -> DList a
    prev (Cons a p n) = p
    next (Cons a p n) = n
```

We print a doubly-linked list by displaying the current entry:

```haskell
instance Show a => Show (DList a)
    where show d = show (elem d)
```

Our book is then a list \([p1,p2,p3]\) of three pages, where

\[
p1 = \text{Cons } "\text{Page 1}" \ p3 \ p2 \\
p2 = \text{Cons } "\text{Page 2}" \ p1 \ p3 \\
p3 = \text{Cons } "\text{Page 3}" \ p2 \ p1
\]

This example suggests that the function \(\text{mkCDList} :: [a] \rightarrow \text{DList } a\) for converting a (nonempty) list \(a\) into a doubly-linked list can be specified as the first element in a finite list \(xs\) of doubly-linked lists satisfying the following three properties:

\[
\begin{align*}
\text{map elem } xs &= as \\
\text{map prev } xs &= \text{rotr } xs \\
\text{map next } xs &= \text{rotl } xs
\end{align*}
\]

Here, \(\text{rotr}\) and \(\text{rotl}\) (short for \(\text{rotate right}\) and \(\text{rotate left}\)), are defined by

\[
\begin{align*}
\text{rotr } xs &= [\text{last } xs] ++ \text{init } xs \\
\text{rotl } xs &= \text{tail } xs ++ [\text{head } xs]
\end{align*}
\]
Observe now that for any list \(xs\) of doubly-linked lists we have

\[
x = \text{zipWith3 Cons} \\
(\text{map elem } xs) (\text{map prev } xs) (\text{map next } xs)
\]

where \(\text{zipWith3}\) is like \(\text{zipWith}\) except that it takes three lists instead of two. The standard prelude definition is:

\[
\text{zipWith3 } f \ (x:xs) \ (y:ys) \ (z:zs) = f \ x \ y \ z : \text{zipWith3 } f \ xs \ ys \ zs \\
\text{zipWith3 } _ \ _ \ _ \ = \ []
\]

We will see another definition in a moment. We can prove the claim above by induction. It clearly holds for the undefined and empty lists. For the inductive case we reason:

\[
x:xs = \\
\{\text{since } xs \text{ is a doubly-linked list}\} \\
\text{Cons (elem } x) \ (\text{prev } x) \ (\text{next } x):xs = \\
\{\text{induction}\} \\
\text{Cons (elem } x) \ (\text{prev } x) \ (\text{next } x): \\
(\text{zipWith3 Cons} \\
(\text{map elem } xs) (\text{map prev } xs) (\text{map next } xs)) = \\
\{\text{definition of } \text{zipWith3 and map}\} \\
\text{zipWith3 Cons} \\
(\text{map elem } (x:xs)) (\text{map prev } (x:xs)) (\text{map next } (x:xs))
\]

Putting this result together with our specification of doubly-linked lists, we arrive at

\[
\text{mkCDList as} = \text{head } xs \\
\text{where } xs = \text{zipWith3 Cons } (\text{rotr } xs) (\text{rotl } xs)
\]

This definition involves a cyclic list \(xs\). Does it work? The answer is: No, it doesn’t. The reason is that \(\text{zipWith3}\) as defined above is too eager. We need to make it lazier by not demanding the values of the second two lists until they are really needed:

\[
\text{zipWith3 } f \ (x:xs) \ ys \ zs = f \ x \ (\text{head } ys) \ (\text{head } zs): \\
\text{zipWith3 } f \ xs \ (\text{tail } ys) \ (\text{tail } zs) \\
\text{zipWith3 } _ \ _ \ _ \ = \ []
\]
An equivalent way to define this function is to make use of Haskell’s irrefutable patterns:

\[
\text{zipWith3 } f (x:xs) \sim (y:ys) \sim (z:zs) \\
= f x y z : \text{zipWith3 } f xs ys zs \\
\text{zipWith3 } _ _ _ = []
\]

An irrefutable pattern is introduced using a tilde, and \((x:xs)\) is matched lazily, meaning that no matching is actually performed until either \(x\) or \(xs\) is needed.

Just to convince ourselves that the above definition of \(\text{mkCDList}\) with the revised definition of \(\text{zipWith3}\) does make progress, let \(xs_0 = \bot\) and

\[
xs_{n+1} = \text{zipWith3 } \text{Cons } "A" \text{ (rotr } xs_n \text{) (rotl } xs_n \text{)}
\]

Then \(xs_1\) is given by

\[
\text{zipWith3 } \text{Cons } "A" \bot \bot \\
= [\text{Cons } 'A' \bot \bot]
\]

and \(xs_2\) by

\[
\text{zipWith3 } \text{Cons } "A" \\
[\text{Cons } 'A' \bot \bot] [\text{Cons } 'A' \bot \bot] \\
= [\text{Cons } 'A' (\text{Cons } 'A' \bot \bot) \text{ (Cons } 'A' \bot \bot)]
\]

and so on.

9.7 Exercises

Exercise A

Given three lists \(xs\), \(ys\) and \(zs\) in strictly increasing order, we have

\[
\text{merge } (\text{merge } xs \text{ ys}) \text{ zs} = \text{merge } xs \text{ (merge } ys \text{ zs)}
\]

Thus \(\text{merge}\) is associative. Assuming in addition that the first elements of \(xs\), \(ys\) and \(zs\) are in strictly increasing order, we also have

\[
\text{xmerge } (\text{xmerge } xs \text{ ys}) \text{ zs} = \text{xmerge } xs \text{ (xmerge } ys \text{ zs)}
\]

Does it follow that in the expression \(\text{foldr1 } \text{xmerge } \text{multiples}\) we could replace \(\text{foldr1}\) by \(\text{foldl1}\)?
Exercise B

The standard prelude function \( \text{cycle} :: [a] \to [a] \) takes a list \( xs \) and returns a list consisting of an infinite number of repetitions of the elements of \( xs \). If \( xs \) is the empty list, then \( \text{cycle} [] \) returns an error message. For instance

\[
\text{cycle } "\text{hallo}" = "\text{hallohallohallo}"
\]

Define \( \text{cycle} \) using a cyclic list. Ensure that your definition works on empty, finite and infinite lists.

Exercise C

The fibonacci function is defined by

\[
\begin{align*}
\text{fib} 0 & = 0 \\
\text{fib} 1 & = 1 \\
\text{fib} n & = \text{fib} (n-1) + \text{fib} (n-2)
\end{align*}
\]

Write down a one-line definition of the list \( \text{fibs} \) that produces the infinite list of Fibonacci numbers.

Exercise D

A well-known problem, due to the mathematician W.R. Hamming, is to write a program that produces an infinite list of numbers with the following properties: (i) the list is in strictly increasing order; (ii) the list begins with the number 1; (iii) if the list contains the number \( x \), then it also contains the numbers \( 2x, 3x \) and \( 5x \); (iv) the list contains no other numbers. Thus, the required list begins with the numbers

\[
1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, \ldots
\]

Write a definition of \( \text{hamming} \) that produces this list.

Exercise E

Prove that \( \text{approx} \ n \ xs \subseteq xs \) for all \( n \). Now prove that if \( \text{approx} \ n \ xs \subseteq ys \) for all \( n \), then \( xs \subseteq ys \). Hence conclude that

\[
\lim_{n \to \infty} \text{approx} \ n \ xs = xs.
\]

Exercise F

Give a counter-example to the claim that \( xs = ys \) if \( xs !! n = ys !! n \) for all \( n \).
Exercise G
Prove that \( \text{iterate } f \ x = x : \text{map } f \ (\text{iterate } f \ x) \).

Exercise H
In the definition of \( \text{primes} \) as a cyclic list, could we have defined
\[
\text{mergeAll} = \text{foldr } \text{xmerge } []
\]
as an alternative to the definition in the text?

Exercise I
Recall that
\[
\text{prs } n = \text{approx } n \ (2 : (\mathbb{N} \setminus \text{crs } n))
\]
\[
\text{crs } n = \text{mergeAll } [\text{map } (\text{p} \ast) \ [\text{p}..] \mid \text{p} \leftarrow \text{prs } n]
\]
Given that \( \text{prs } n = p_1 : p_2 : \cdots p_n : \bot \), where \( p_j \) is the \( j \)th prime, sketch how to show that \( \text{crs } n = c_1 : c_2 : \cdots c_m : \bot \), where \( c_j \) is the \( j \)th composite number (so \( c_1 = 4 \)) and \( m = p_n^2 \). Hence show that \( \text{primes} \) does produce the infinite list of primes.

We said in the text that it is not the case that the \( n \)th approximation \( \text{primes}_n \) of \( \text{primes} \) is equal to \( \text{approx } n \ \text{primes} \). In fact
\[
\text{primes}_4 = 2 : 3 : 5 : 7 : \cdots : 47 : \bot
\]
What list does \( \text{primes}_5 \) produce?

Exercise J
Another way of generating the primes is known as the Sieve of Sundaram, after its discoverer S.P. Sundaram in 1934:
\[
\text{primes} = 2 : [2 \ast n + 1 \mid n \leftarrow [1..] \setminus \text{sundaram}]
\]
\[
\text{sundaram} = \text{mergeAll } [[i + j + 2 \ast i \ast j \mid j \leftarrow [i..]] \mid i \leftarrow [1..]]
\]
To show that the list comprehension in the definition of \( \text{primes} \) generates exactly the odd primes, it is sufficient to prove that the term \( 2 \ast n + 1 \) is never composite, which is to say that it never factorises into \( (2 \ast i + 1) \ast (2 \ast j + 1) \) for positive integers \( i \) and \( j \). Why is this so?

Exercise K
Is the function \( f \), defined by \( f(\bot) = 0 \) and \( f(x) = 1 \) for \( x \neq \bot \), computable? How about the function that returns \( \bot \) on all finite or partial lists, and \( 1 \) on all infinite lists?
Exercise L

By definition, a torus is a doubly-cyclic, doubly-doubly-linked list. It is a cyclic doubly-linked list in the left/right direction, and also in the up/down direction. Given a matrix represented as a list of length \( m \) of lists, all of length \( n \), construct a definition of

\[
\text{mkTorus :: Matrix a -> Torus a}
\]

where

\[
\text{data Torus a = Cell a (Torus a) (Torus a) (Torus a) (Torus a)}
\]

\[
\begin{align*}
\text{elem (Cell a u d l r) } &= a \\
\text{up (Cell a u d l r) } &= u \\
\text{down (Cell a u d l r) } &= d \\
\text{left (Cell a u d l r) } &= l \\
\text{right (Cell a u d l r) } &= r
\end{align*}
\]

That looks tricky, but the answer is short enough to be tweeted.

9.8 Answers

Answer to Exercise A

No, since \( \text{foldl1 f xs = undefined} \) for any infinite list \( xs \).

Answer to Exercise B

The definition is

\[
\text{cycle [] = error "empty list"}
\]

\[
\text{cycle xs = ys where ys = xs ++ ys}
\]

Note that if \( xs \) is infinite, then \( xs ++ ys = xs \), so \( \text{cycle} \) is the identity function on infinite lists.

Answer to Exercise C

The one-liner is:

\[
\begin{align*}
\text{fibs :: [Integer]} \\
\text{fibs = 0:1:zipWith (+) fibs (tail fibs)}
\end{align*}
\]
Answer to Exercise D

```haskell
hamming :: [Integer]
hamming = 1: merge (map (2*) hamming)
       (merge (map (3*) hamming)
       (map (5*) hamming))
```

Answer to Exercise E

The proof of \( \text{approx } n \; \text{xs} \sqsubseteq \text{xs} \) is by induction on \( n \). The base case is easy but the induction step involves a sub-induction over \( \text{xs} \). The base cases (the empty list and the undefined list) of the sub-induction are easy and the inductive case is

\[
\text{approx} \; (n+1) \; (x:xs) = \begin{cases} 
\text{definition} \\
 x: \text{approx} \; n \; \text{xs} \\
\sqsubseteq \begin{cases} 
\text{induction and monotonicity of } (x:) \\
 x: \text{xs}.
\end{cases}
\end{cases}
\]

The proof of

\[
(\forall n : \text{approx } n \; \text{xs} \sqsubseteq \text{ys}) \Rightarrow \text{xs} \sqsubseteq \text{ys}
\]

is by induction on \( \text{xs} \). The claim is immediate for the undefined and empty lists, and for the inductive case we have

\[
(\forall n : \text{approx } n \; (x:xs) \sqsubseteq \text{ys}) \\
\Rightarrow \text{xs} \sqsubseteq \text{head} \; \text{ys} \land (\forall n : \text{approx } n \; \text{xs} \sqsubseteq \text{tail} \; \text{ys})
\]

by the definitions of \text{approx} and the approximation ordering on lists. By induction we therefore have

\[
x: \text{xs} \sqsubseteq \text{head} \; \text{ys}: \text{tail} \; \text{ys} = \text{ys}.
\]

It follows that

\[
\lim_{n \to \infty} \text{approx } n \; \text{xs} = \text{xs}
\]

by the definition of limit.

Answer to Exercise F

The two lists \text{repeat undefined} and \text{undefined} are not equal, but

\[
(\text{repeat undefined})!!n = \text{undefined}!!n
\]

for all \( n \) because both sides are \( \bot \).
Answer to Exercise G

We have to show that

\[ \text{approx } n \text{ (iterate } f \text{ x)} = \text{approx } n \text{ (x:map } f \text{ (iterate } f \text{ x))} \]

for all natural numbers \( n \). This claim follows from

\[ \text{approx } n \text{ (iterate } f \text{ (f x))} = \text{approx } n \text{ (map } f \text{ (iterate } f \text{ x))} \]

which we establish by induction on \( n \). For the inductive step we simplify each side. For the left-hand side:

\[
\begin{align*}
\text{approx } (n+1) \text{ (iterate } f \text{ (f x))} &= \{\text{definition of iterate}\} \\
&= \text{approx } (n+1) \text{ (f x:iterate } f \text{ (f (f x)))} \\
&= \{\text{definition of approx}\} \\
&= f x: \text{approx } n \text{ (iterate } f \text{ (f (f x)))} \\
&= \{\text{induction}\} \\
&= f x: \text{approx } n \text{ (map } f \text{ (iterate } f \text{ (f x)))}
\end{align*}
\]

For the right-hand side:

\[
\begin{align*}
\text{approx } (n+1) \text{ (map } f \text{ (iterate } f \text{ x))} &= \{\text{definition of iterate and map}\} \\
&= \text{approx } (n+1) \text{ (f x:map } f \text{ (iterate } f \text{ (f x)))} \\
&= \{\text{definition of approx}\} \\
&= f x: \text{approx } n \text{ (map } f \text{ (iterate } f \text{ (f x)))}
\end{align*}
\]

Answer to Exercise H

Yes, since

\[ \text{foldr } xmerge \text{ [] (xs:undefined) = xmerge xs undefined} \]

and the right-hand side begins with the first element of \( xs \).

Answer to Exercise I

The proof is by induction. We have first to show that \( \text{crs (n+1)} \) is the result of merging \( c_1 : c_2 : \cdots c_m : \bot \), where \( m = p_2^2 \) with the infinite list of multiples
That gives the partial list of all composite numbers up to \( p_{n+1}^2 \). Finally, we need the result that \( p_{n+2} < p_{n+1}^2 \).

The partial list \( \text{primes}_5 \) produces all the primes smaller than \( 2209 = 47 \times 47 \).

**Answer to Exercise J**

Because an odd integer is excluded from the final list if it takes the form \( 2n + 1 \) where \( n \) is of the form \( i + j + 2ij \). But

\[
2(i+j+2ij) + 1 = (2i+1)(2j+1).
\]

**Answer to Exercise K**

No, \( f \) is not monotonic: \( \bot \sqsubseteq 1 \) but \( f(\bot) \not\sqsubseteq f(1) \). For the second function (call it \( g \)) we have \( \text{xss} \sqsubseteq \text{ys} \) implies \( g(\text{xss}) \sqsubseteq g(\text{ys}) \), so \( g \) is monotonic. But \( g \) is not continuous, so it’s not computable.

**Answer to Exercise L**

The definition is

\[
\text{mkTorus ass} = \text{head} (\text{head} \ \text{xss})
\]

where \( \text{xss} = \text{zipWith5} (\text{zipWith5 Cell}) \)

\[
\text{ass} (\text{rotr xss}) (\text{rotl xss})
\]

\[
(\text{map rotr xss}) (\text{map rotl xss})
\]

Whereas \( \text{rotr} \) and \( \text{rotl} \) rotate the rows of a matrix, \( \text{map rotr} \) and \( \text{map rotl} \) rotate the columns. The definition of \( \text{zipWith5} \) has to be made non-strict in its last four arguments.

**9.9 Chapter notes**

Melissa O’Neill has written a nice pearl on sieve methods for generating primes; see ‘The genuine sieve of Eratosthenes’, *Journal of Functional Programming* 19 (1), 95–106, 2009. Ben Sijtsma’s thesis *Verification and derivation of infinite-list programs* (University of Groningen, the Netherlands, 1988) studies various aspects of infinite-list programs and gives a number of techniques for reasoning about them. One chapter is devoted to the proof of fairness in the paper–rock–scissors game.

My paper, ‘On building cyclic and shared data structures in Haskell’, *Formal Aspects of Computing* 24(4–6), 609–621, July 2012, contains more examples of the uses of infinite and cyclic lists. See also the article on ‘Tying the knot’ at
Infinite lists

haskell.org/haskellwiki/Tying_the_Knot

Hamming’s problem has been used as an illustration of cyclic programs since the early days of functional programming.
Back in Chapter 2 we described the function putStrLn as being a Haskell command, and IO a as being the type of input–output computations that interact with the outside world and deliver values of type a. We also mentioned some syntax, called do-notation, for sequencing commands. This chapter explores what is really meant by these words, and introduces a new style of programming called monadic programming. Monadic programs provide a simple and attractive way to describe interaction with the outside world, but are also capable of much more: they provide a simple sequencing mechanism for solving a range of problems, including exception handling, destructive array updates, parsing and state-based computation. In a very real sense, a monadic style enables us to write functional programs that mimic the kind of imperative programs one finds in languages such as Python or C.

10.1 The IO monad

The type IO a is an abstract type in the sense described in the previous chapter, so we are not told how its values, which are called actions or commands, are represented. But you can think of this type as being

    type IO a = World -> (a,World)

Thus an action is a function that takes a world and delivers a value of type a and a new world. The new world is then used as the input for the next action. Having changed the world with an input–output action, you can’t go back to the old world. You can’t duplicate the world or inspect its components. All you can do is operate on the world with given primitive actions, and put such actions together in a sequence.
One primitive action is to print a character:

```haskell
putChar :: Char -> IO ()
```

When executed, this action prints a character on the standard output channel, usually the computer screen. For example,

```haskell
ghci> putChar 'x'
x
```

The character \(x\) is printed, but nothing else, so the next GHCi prompt follows without additional spaces or newlines. Performing this action produces no value of interest, so the return value is the null tuple (\()\).

Another primitive action is `done :: IO ()`, which does nothing. It leaves the world unchanged and also returns the null tuple (\()\).

One simple operation to sequence actions is denoted by `>>` and has type

```haskell
(>>) :: IO () -> IO () -> IO ()
```

Given actions \(p\) and \(q\), the action \(p >> q\) first performs action \(p\) and then performs action \(q\). For example,

```haskell
ghci> putChar 'x' >> putChar '\n'
x
```

This time a newline is printed. Using `>>` we can define the function `putStrLn`:

```haskell
putStrLn :: String -> IO ()
putStrLn xs = foldr (>>) done (map putChar xs) >>
        putChar '\n'
```

This action prints all the characters in a string, and then finishes up with an additional newline character. Note that `map putChar xs` is a list of actions. We are still in the universe of functional programming and its full expressive power, including uses of `map` and `foldr`, is still available to us.

Here is another primitive action:

```haskell
getChar :: IO Char
```

When performed, this operation reads a character from the standard input channel. This channel is fed by you typing at the keyboard, so `getChar` returns the first character you type. For example,
10.1 The IO monad

ghci> getChar
x
'x'

After typing getChar and pressing return, GHCi waits for you to type a character. We typed the character 'x' (and what we typed was echoed), and then that character was read and printed.

The generalisation of done is an action that does nothing and returns a named value:

\[
\text{return} :: \text{a} \rightarrow \text{IO a}
\]

In particular, done = return (). The generalisation of (>>) has type

\[
(\gg) :: \text{IO a} \rightarrow \text{IO b} \rightarrow \text{IO b}
\]

Given actions p and q, the action \(p \gg q\) first does p, and then throws the return value away, and then does q. For example,

ghci> return 1 >> return 2
2

It is clear that this action is useful only when the value returned by p is not interesting since there is no way that q can depend on it. What is really wanted is a more general operator (\(\gg\)) with type

\[
(\gg) :: \text{IO a} \rightarrow (\text{a} \rightarrow \text{IO b}) \rightarrow \text{IO b}
\]

The combination \(p \gg f\) is an action that, when performed, first does p, returning a value x of type a, then does action \(f \ x\) returning a final value y of type b. It is easy to define (\(\gg\)) in terms of (\(\gg\)) and we leave this as an exercise. The operator (\(\gg\)) is often referred to as bind, though one can also pronounce it as ‘then apply’.

Using (\(\gg\)), we can define a function getline for reading a line of input, more precisely, the list of characters up to but not including the first newline character:

\[
\text{getLine} :: \text{IO String}
\]

\[
\text{getLine} = \text{getChar} \gg f
\]

where

\[
f x = \text{if } x = \text{\'\n\'} \text{ then return [] else getline} \gg g
\]

where

\[
g xs = \text{return (x:xs)}
\]

This has a straightforward reading: get the first character x; stop if x is a newline and return the empty list; otherwise get the rest of the line and add x to the front. Though the reading is straightforward, the use of nested where clauses makes the
definition a little clumsy. One way to make the code smoother is to use anonymous lambda expressions and instead write:

```haskell
getLine = getChar >>= \x ->
  if x == '\n'
  then return []
  else getLine >>= \xs ->
    return (x:xs)
```

Another, arguably superior solution is to use do-notation:

```haskell
getLine = do x <- getChar
  if x == '\n'
  then return []
  else do xs <- getLine
    return (x:xs)
```

The right-hand side makes use of the Haskell layout convention. Note especially the indentation of the conditional expression, and the last return to show it is part of the inner do. Better in our opinion is to use braces and semicolons to control the layout explicitly:

```haskell
getLine = do {x <- getChar;
  if x == '\n'
  then return []
  else do {xs <- getLine;
    return (x:xs)}}
```

We return to do-notation below.

The Haskell library System.IO provides many more actions than just putChar and getChar, including actions to open and read files, to write and close files, to buffer output in various ways and so on. We will not go into details in this book. But perhaps two more things need to be said. Firstly, there is no function of type `IO a -> a` \(^1\). Once you are in a room performing input–output actions, you stay in the room and can’t come out of it. To see one reason this has to be the case, suppose there is such a function, `runIO` say, and consider

```haskell
int :: Int
int = x - y
  where x = runIO readInt
       y = runIO readInt
```

\(^1\) Actually there is, and it’s called `unsafePerformIO`, but it is a very unsafe function.
readInt = do {xs <- getLine; return (read xs :: Int)}

The action readInt reads a line of input and, provided the line consists entirely of digits, interprets it as an integer. Now, what is the value of int? The answer depends entirely on which of x and y gets evaluated first. Haskell does not prescribe whether or not x is evaluated before y in the expression x-y. Put it this way: input–output actions have to be sequenced in a deterministic fashion, and Haskell is a lazy functional language in which it is difficult to determine the order in which things happen. Of course, an expression such as x-y is a very simple example (and exactly the same undesirable phenomenon arises in imperative languages) but you can imagine all sorts of confusion that would ensue if we were provided with runIO.

The second thing that perhaps should be said is in response to a reader who casts a lazy eye over an expression such as

```
undefined >> return 0 :: IO Int
```

Does this code raise an error or return zero? The answer is: an error. IO is strict in the sense that IO actions are performed in order, even though subsequent actions may take no heed of their results.

To return to the main theme, let us summarise. The type IO a is an abstract type on which the following operations, at least, are available:

```
return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b
putChar :: Char -> IO ()
getChar :: IO Char
```

The second two functions are specific to input and output, but the first two are not. Indeed they are general sequencing operations that characterise the class of types called monads:

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

The two monad operations are required to satisfy certain laws, which we will come to in due course. As to the reason for the name ‘monad’, it is stolen from philosophy, in particular from Leibniz, who in turn borrowed it from Greek philosophy. Don’t read anything into the name.
10.2 More monads

If that’s all a monad is, then surely lots of things form a monad? Yes, indeed. In particular, the humble list type forms a monad:

```haskell
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
```

Of course, we don’t yet know what the laws governing the monad operations are, so maybe this instance isn’t correct (it is), but at least the operations have the right types. Since do-notation can be used with any monad we can, for example, define the cartesian product function `cp :: [[a]] -> [[a]]` (see Section 7.3) using the new notation:

```haskell
cp [] = return []
cp (xs:xss) = do {x <- xs;
                 ys <- cp xss;
                 return (x:ys)}
```

Comparing the right-hand side of the second clause to the list comprehension

```
[x:ys | x <- xs, ys <- cp xss]
```

one can appreciate that the two notations are very similar; the only real difference is that with do-notation the result appears at the end rather than at the beginning. If monads and do-notation had been made part of Haskell before list comprehensions, then maybe the latter wouldn’t have been needed.

Here is another example. The `Maybe` type is a monad:

```haskell
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f = f x
```

To appreciate what this monad can bring to the table, consider the Haskell library function

```haskell
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

The value of `lookup x alist` is `Just y` if `(x,y)` is the first pair in `alist` with first component `x`, and `Nothing` if there is no such pair. Imagine looking up `x` in `alist`, then looking up the result `y` in a second list `blist`, and then looking up the result `z` in yet a third list `clist`. If any of these lookups return `Nothing`, then
Nothing is the final result. To define such a function we would have to write its defining expression as something like

```haskell
case lookup x alist of
  Nothing -> Nothing
  Just y  -> case lookup y blist of
             Nothing -> Nothing
             Just z  -> lookup z clist
```

With a monad we can write

```haskell
do {y <- lookup x alist;
    z <- lookup y blist;
    return (lookup z clist)}
```

Rather than having to write an explicit chain of computations, each of which may return Nothing, and explicitly passing Nothing back up the chain, we can write a simple monadic expression in which handling Nothing is done implicitly under a monadic hood.

### do-notation

Just as list comprehensions can be translated into expressions involving `map` and `concat`, so do-expressions can be translated into expressions involving `return` and `bind`. The three main translation rules are:

```haskell
do {p} = p
do {p;stmts} = p >> do {stmts}
do {x <- p;stmts} = p >>= \x -> do {stmts}
```

In these rules `p` denotes an action, so the first rule says that a `do` round a single action can be removed. In the second and third rules `stmts` is a nonempty sequence of statements, each of which is either an action or a statement of the form `x <- p`. The latter is not an action; consequently an expression such as

```haskell
do {x <- getChar}
```

is not syntactically correct. Nor, by the way, is an empty `do`-expression `do { }`. The last statement in a `do`-expression must be an action.

On the other hand, the following two expressions are both fine:

```haskell
do {putStrLn "hello "; name <- getLine; putStrLn name}
do {putStrLn "hello "; getLine; putStrLn "there"}
```
The first example prints a greeting, reads a name and completes the greeting. The second prints a greeting, reads a name but immediately forgets it, and then completes the greeting with a ‘there’. A bit like being introduced to someone in real life.

Finally, there are two rules that can be proved from the translation rules above:

\[
\begin{align*}
\text{do } \{\text{do } \{\text{stmts}\}\} &= \text{do } \{\text{stmts}\} \\
\text{do } \{\text{stmts1; do } \{\text{stmts2}\}\} &= \text{do } \{\text{stmts1; stmts2}\}
\end{align*}
\]

But one has to be careful; the nested dos in

\[
\begin{align*}
\text{do } \{\text{stmts1; if } p \text{ then do } \{\text{stmts2}\} \text{ else do } \{\text{stmts3}\}\}
\end{align*}
\]

are necessary if \text{stmts2} and \text{stmts3} contain more than one action.

\textit{Monad laws}

The monad laws say nothing much more than that expressions involving \texttt{return} and (\texttt{>>=}) simplify in just the way one would expect. There are three laws and we are going to state them in three different ways. The first law states that \texttt{return} is a right identity element of (\texttt{>>=}):

\[
(p \ggp \texttt{return}) = p
\]

In do-notation the law reads:

\[
\text{do } \{x \leftarrow p; \texttt{return } x\} = \text{do } \{p\}
\]

The second law says that \texttt{return} is also a kind of left identity element:

\[
(\texttt{return } e \ggp f) = f\ e
\]

In do-notation the law reads:

\[
\text{do } \{x \leftarrow \texttt{return } e; f\ x\} = \text{do } \{f\ e\}
\]

The third law says that (\texttt{>>=}) is kind of associative:

\[
((p \ggp f) \ggp g) = p \ggp (\lambda x \rightarrow (f\ x \ggp g))
\]

In do-notation the law reads:
\[
\text{do \{y <- do \{x <- p; f x\}; g y\}}
\]
\[
= \text{do \{x <- p; do \{y <- f x; g y\}\}}
\]
\[
= \text{do \{x <- p; y <- f x; g y\}}
\]

The last line makes use of the un-nesting property of do-notation.

For the third way of stating the monad laws, consider the operator \((\Rightarrow\Rightarrow)\) defined by

\[
(\Rightarrow\Rightarrow) : \text{Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)}
\]
\[
(f \Rightarrow\Rightarrow g) x = f x >\Rightarrow g
\]

This operator is just like function composition except that the component functions each have type \(x \to m y\) for appropriate \(x\) and \(y\), and the order of composition is from left to right rather than from right to left. This operator, which is called \((left to right)\) Kleisli composition, is defined in the Haskell library \texttt{Control.Monad}. There is a dual version, \((right to left)\) Kleisli composition,

\[
(\Leftarrow\Leftarrow) : \text{Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)}
\]

whose definition we leave as an easy exercise.

The point is that we can define \((\Rightarrow\Rightarrow)\) in terms of \((\Rightarrow\Rightarrow)\):

\[
(p >\Rightarrow f) = (id >\Rightarrow f) p
\]

More briefly, \((\Rightarrow\Rightarrow) = \text{flip (id >\Rightarrow)}\). We also have the leapfrog rule:

\[
(f >\Rightarrow g) . h = (f . h) >\Rightarrow g
\]

The proof is left as an exercise.

In terms of \((\Rightarrow\Rightarrow)\) the three monad laws say simply that \((\Rightarrow\Rightarrow)\) is associative with identity \texttt{return}. Any set of values with an associative binary operation and an identity element is called a monoid, and the word ‘monad’ was probably adopted because of the pun with monoid. Be that as it may, this is certainly the shortest way of stating the monad laws.

One additional and instructive way of describing the monad laws is considered in the exercises.

10.3 The State monad

If it wasn’t for the problem of how to sequence input–output actions correctly, monads probably wouldn’t have appeared in Haskell. But once it was appreciated
what they could do, all kinds of other uses quickly followed. We have seen with the Maybe monad how chains of computations that involve passing information back up the chain can be simplified with monadic notation. Another primary use of monads is a way to handle mutable structures, such as arrays, that rely for their efficiency on being able to update their values, destroying the original structure in the process.

Mutable structures are introduced through the State-Thread monad ST s which we will consider in a subsequent section. Before getting on to the particular properties of this monad, we start by considering a simpler monad, called State s, for manipulating an explicit state s. You can think of the type State s a as being

\[
\text{type State s a = s -> (a,s)}
\]

An action of type State s a takes an initial state and returns a value of type a and a new state. It is tempting, but wrong, to think of IO a as synonymous with State World a. The state component s in State s a can be exposed and manipulated, but we can't expose and manipulate the world.

Specifically, as well as the monad operations return and (>>=), five other functions are provided for working with the state monad:

\[
\begin{align*}
\text{put} & : s \rightarrow \text{State } s () \\
\text{get} & : \text{State } s s \\
\text{state} & : (s \rightarrow (a,s)) \rightarrow \text{State } s a \\
\text{runState} & : \text{State } s a \rightarrow (s \rightarrow (a,s)) \\
\text{evalState} & : \text{State } s a \rightarrow s \rightarrow a
\end{align*}
\]

The function put puts the state into a given configuration, while get returns the current state. Each of these two operations can be defined in terms of state:

\[
\begin{align*}
\text{put } s & = \text{state } (\_ \rightarrow (((),s)) \\
\text{get } & = \text{state } (\_s \rightarrow (s,s))
\end{align*}
\]

On the other hand, state can also be defined using put and get:

\[
\text{state } f = \text{do } \{ s \leftarrow \text{get}; \text{let } (a,s') = f s; \text{put } s'; \text{return } a\}
\]

Haskell permits an abbreviated form of let expressions in do expressions (and also in list comprehensions). We have

\[
\text{do } \{ \text{let } \text{decls; } \text{stmts} \} = \text{let } \text{decls in do } \{ \text{stmts} \}
\]

The function runState is the inverse of state: it takes both an action and an
10.3 The State monad

The State monad initial state and returns the final value and the final state after performing the action (something the IO monad cannot do). The function `evalState` is defined by

\[
\text{evalState } m \ s = \text{fst} \ (\text{runState } m \ s)
\]

and returns just the value of the stateful computation.

Here is an example of the use of State. In Section 7.6 we constructed the following program for building a binary tree out of a given nonempty list of values:

```
build :: [a] -> BinTree a
build xs = fst (build2 (length xs) xs)
build2 1 xs = (Leaf (head xs), tail xs)
built2 n xs = (Fork u v, xs'')
    where (u,xs') = built2 m xs
        (v,xs'') = built2 (n-m) xs'
        m = n \ `div` 2
```

The point to appreciate here is that `build2` is essentially a function that manipulates a state of type `[a]`, returning elements of `BinTree a` as its result. Another way of writing `build` is as follows:

```
build xs = evalState (build2 (length xs)) xs

build2 :: Int -> State [a] (BinTree a)
built2 1 = do {x:xs <- get;
    put xs;
    return (Leaf x)}
built2 n = do {u <- built2 m;
    v <- built2 (n-m);
    return (Fork u v)}
    where m = n `div` 2
```

All the work in manipulating the state explicitly is done when building a leaf. The state is accessed and its first element is chosen as the label associated with a Leaf; the remaining list then is installed as the new state. Whereas the first version of `build2 n` threads the state explicitly, the second version hides this machinery under a monadic hood.

Notice in the first line of `build2` we have a statement `x:xs <- get` in which the left-hand side is a pattern rather than a simple variable. If the current state happens to be the empty list, the action fails with a suitable error message. For example,

```
ghci> runState (do {x:xs <- get; return x}) ""
```

Of course this behaviour cannot arise with \texttt{build2 1} because the definition only applies when the state is a singleton list. We leave it as an exercise to say what \texttt{build []} does.

As another example, consider the problem of producing a pseudo-random integer in a specified interval. Imagine we have a function

\begin{verbatim}
random :: (Int,Int) \rightarrow Seed \rightarrow (Int,Seed)
\end{verbatim}

that takes a pair of integers as the specified interval and then a seed, and calculates a random integer and a new seed. The new seed is used for obtaining further random values. Rather than be explicit about what a seed is, suppose there is a function

\begin{verbatim}
mkSeed :: Int \rightarrow Seed
\end{verbatim}

that makes a seed from a given integer. Now if we wanted to roll a pair of dice, we could write

\begin{verbatim}
diceRoll :: Int \rightarrow (Int,Int)
diceRoll n = (x,y)
  where (x,s1) = random (1,6) (mkSeed n)
        (y,s2) = random (1,6) s1
\end{verbatim}

But we could also write

\begin{verbatim}
diceRoll n = evalState (do {x <- randomS (1,6);
  y <- randomS (1,6);
  return (x,y)}
  ) (mkSeed n)
  where randomS = state . random
\end{verbatim}

The function \texttt{randomS :: (Int,Int) \rightarrow State Seed Int} takes an interval and returns an action. The second version of \texttt{diceRoll} is a little longer than the first, but is arguably more easy to write. Imagine that instead of two dice we had five, as in liar dice. The first method would involve a chain of where-clauses expressing the linkage between five values and five seeds, something that would be easy to mistype, but the second version is easily extended and harder to get wrong.

One final point. Consider

\begin{verbatim}
evalState (do {undefined; return 0}) 1
\end{verbatim}

Does this raise an exception, or does it return zero? In other words, is the monad
State strict, as the IO monad is, or is it lazy? The answer is that it can be both. There are two variants of the state monad, one of which is lazy and the other of which is strict. The difference lies in how the operation ($\gg=$) is implemented. Haskell provides the lazy variant by default, in Control.Monad.State.Lazy, but you can ask for the strict variant, in Control.Monad.State.Strict if you want.

10.4 The ST monad

The state-thread monad, which resides in the library Control.Monad.ST, is a different kettle of fish entirely from the state monad, although the kettle itself looks rather similar. Like State $s$ a you can think of this monad as the type

\[
\text{type ST } s \text{ a } = s \rightarrow (a, s)
\]

but with one very important difference: the type variable $s$ cannot be instantiated to specific states, such as Seed or [Int]. Instead it is there only to name the state. Think of $s$ as a label that identifies one particular state thread. All mutable types are tagged with this thread, so that actions can only affect mutable values in their own state thread.

One kind of mutable value is a program variable. Unlike variables in Haskell, or mathematics for that matter, program variables in imperative languages can change their values. They can be thought of as references to other values, and in Haskell they are entities of type STRef $s$ a. The $s$ means that the reference is local to the state thread $s$ (and no other), and the $a$ is the type of value being referenced. There are operations, defined in Data.STRef, to create, read from and write to references:

\[
\begin{align*}
\text{newSTRef} & : : \text{a } \rightarrow \text{ST } s \text{ (STRef } s \text{ a)} \\
\text{readSTRef} & : : \text{STRef } s \text{ a } \rightarrow \text{ST } s \text{ a} \\
\text{writeSTRef} & : : \text{STRef } s \text{ a } \rightarrow \text{a } \rightarrow \text{ST } s \text{ ()}
\end{align*}
\]

Here is an example. Recall Section 7.6 where we gave the following definition of the Fibonacci function:

\[
\begin{align*}
fib & : : \text{Int } \rightarrow \text{Integer} \\
fib \text{ n} & = \text{fst} (\text{fib2 } n) \\
fib2 \text{ 0} & = (0, 1) \\
fib2 \text{ n} & = (b, a+b) \text{ where } (a, b) = \text{fib2 } (n-1)
\end{align*}
\]
Evaluating \( \text{fib} \) takes linear time, but the space involved is not constant (even ignoring the fact that arbitrarily large integers cannot be stored in constant space): each recursive call involves fresh variables \( a \) and \( b \). By contrast, here is a definition of \( \text{fib} \) in the imperative language Python:

```python
def fib(n):
    a, b = 0, 1
    for i in range(0, n):
        a, b = b, a + b
    return a
```

The definition manipulates two program variables \( a \) and \( b \), and runs in constant space (at least, for small integers). We can translate the Python code almost directly into Haskell:

```haskell
fibST :: Int -> ST s Integer
fibST n = do {a <- newSTRef 0;
               b <- newSTRef 1;
               repeatFor n
                  (do {x <- readSTRef a;
                        y <- readSTRef b;
                        writeSTRef a y;
                        writeSTRef b $! (x+y)});
               readSTRef a}
```

Note the use of the strict application operator \( ($) ! \) to force evaluation of the sum. The action `repeatFor` repeats an action \( a \) a given number of times:

```haskell
repeatFor :: Monad m => Int -> m a -> m ()
repeatFor n = foldr (>>) done . replicate n
```

All well and good, but we end up with an action \( \text{ST} s \text{ Integer} \) when what we really want is an integer. How do we escape from the monad back into the world of Haskell values?

The answer is to provide a function similar to `runState` for the state monad, Here it is, with its type:

```haskell
runST :: (forall s. ST s a) -> a
```

This type is unlike any other Haskell type we have met so far. It is what is called a \textit{rank 2 polymorphic type}, while all previous polymorphic types have had rank 1. What it says is that the argument of \( \text{runST} \) must be universal in \( s \), so it can’t
depend on any information about \( s \) apart from its name. In particular, every \texttt{STRef} declared in the action has to carry the same thread name \( s \).

To amplify a little on rank 2 types, consider the difference between the two lists

\[
\text{list1} :: \text{forall } a. \ [a \rightarrow a] \\
\text{list2} :: [\text{forall } a. \ a \rightarrow a]
\]

The type of \text{list1} is just what we would have previously written as \([a \rightarrow a]\) because in ordinary rank 1 types universal quantification at the outermost level is assumed. For example, \([\sin, \cos, \tan]\) is a possible value of \text{list1} with the instantiation \texttt{Float} for \( a \). But there are only two functions that can be elements of \text{list2}, namely \texttt{id} and the undefined function \texttt{undefined}, because these are the only two functions with type \([\text{forall } a. \ a \rightarrow a]\). If you give me an element \( x \) of a type \( a \) about which absolutely nothing is known, the only things I can do if I have to give you back an element of \( a \), is either to give you \( x \) or \( \bot \).

Why have a rank 2 type for \texttt{runST}? Well, it prevents us from defining things like

```
let v = runST (newSTRef True)
in runST (readSTRef v)
```

This code is not well-typed because

\texttt{newSTRef True :: ST s (STRef s Bool)}

and in the expression \texttt{runST (newSTRef Bool)} the Haskell type checker cannot match \texttt{STRef s a} with \( a \), the expected result type of \texttt{runST}. Values of type \texttt{STRef s a} cannot be exported from \texttt{ST s}, but only entities whose types do not depend on \( s \). If the code were allowed, then the reference allocated in the first \texttt{runST} would be usable inside the second \texttt{runST}. That would enable reads in one thread to be used in another, and hence the result would depend on the evaluation order used to execute the threads, leading to mayhem and confusion. It is just the same problem that we prevented from occurring in the \texttt{IO} monad.

But we can safely define

```
fib :: Int \rightarrow \text{Integer} 
fib n = runST (fibST n)
```

This version of \texttt{fib} runs in constant space.

For our purposes the main use of the \texttt{ST} monad resides in its ability to handle mutable arrays. The whole question of arrays deserves a section to itself.
It sometimes surprises imperative programmers who meet functional programming for the first time that the emphasis is on lists as the fundamental data structure rather than arrays. The reason is that most uses of arrays (though not all) depend for their efficiency on the fact that updates are destructive. Once you update the value of an array at a particular index the old array is lost. But in functional programming, data structures are persistent and any named structure continues to exist. For instance, `insert x t` may insert a new element `x` into a tree `t`, but `t` continues to refer to the original tree, so it had better not be overwritten.

In Haskell a mutable array is an entity of type `STArray s i e`. The `s` names the state thread, `i` the index type and `e` the element type. Not every type can be an index; legitimate indices are members of the type class `Ix`. Instances of this class include `Int` and `Char`, things that can be mapped into a contiguous range of integers.

Like `STRefs` there are operations to create, read from and write to arrays. Without more ado we consider an example, explaining the actions as we go along. Recall the Quicksort algorithm from Section 7.7:

```haskell
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [y | y < -x s ,y < x] ++ [x] ++
                  qsort [y | y <- xs, x <= y]
```

There we said that when Quicksort is implemented in terms of arrays rather than lists, the partitioning phase can be performed in place without using any additional space. We now have the tools to write just such an algorithm. We begin with

```haskell
qsort :: (Ord a) => [a] -> [a]
qsort xs = runST $
  do {xa <- newListArray (0,n-1) xs;
     qsortST xa (0,n);
     getElems xa}
  where n = length xs
```

First we create a mutable array with bounds `(0,n-1)` and fill it with the elements of `xs`. Sorting the array is done with the action `qsortST xa (0,n)`. At the end, the list of elements of the sorted array is returned. In the code above, the action `newListArray` has type

```
Ix i => (i, i) -> [e] -> ST s (STArray s i e)
```
and getElems has type

\[ \text{Ix } i \Rightarrow \text{STArray } s \ i \ e \Rightarrow \text{ST } s \ [e] \]

The first constructs a mutable array from a list of elements, and the second returns
a list of the elements in a mutable array.

The purpose of \( \text{qsortST } xa \ (a,b) \) is to sort the elements in the sub-array of \( xa \)
in the interval \((a,b)\), where by definition such an interval includes the lower bound
but excludes the upper bound; in other words \([a \ldots b-1]\). Choosing intervals that
are closed on the left but open on the right is almost always the best policy when
processing arrays. Here is the definition of \( \text{qsortST} \):

\[
\text{qsortST} :: \text{Ord } a \Rightarrow \text{STArray } s \ \text{Int} \ a \rightarrow \\
(\text{Int},\text{Int}) \rightarrow \text{ST } s \ ()
\]

\[
\text{qsortST } xa \ (a,b) \\
\mid a == b \quad = \text{return } ()
\mid \text{otherwise } = \text{do } \{ m <- \text{partition } xa \ (a,b); \\
\quad \text{qsortST } xa \ (a,m); \\
\quad \text{qsortST } xa \ (m+1,b) \}
\]

If \( a==b \) we have an empty interval and there is nothing to do. Otherwise we rear-
range the array so that for some suitable element \( x \) in the array all elements in the
interval \((a,m)\) are less than \( x \), and all elements in the interval \((m+1,b)\) are at least
\( x \). The element \( x \) itself is placed in the array at position \( m \). Sorting is then completed
by sorting both sub-intervals.

It remains to define partition. The only way to find a suitable definition is by
formal development using pre- and post-conditions and loop invariants. But this is
a book on functional programming, not on the formal development of imperative
programs, so we are going to cop out and just record one version:

\[
\text{partition } xa \ (a,b) \\
\quad = \text{do } \{ x <- \text{readArray } xa \ a; \\
\quad \text{let } \text{loop } (j,k) \quad = \text{if } j==k \\
\quad \quad \text{then do } \{ \text{swap } xa \ a \ (k-1); \\
\quad \quad \quad \text{return } (k-1) \} \\
\quad \quad \text{else do } \{ y <- \text{readArray } xa \ j; \\
\quad \quad \quad \text{if } y < x \text{ then } \text{loop } (j+1,k) \\
\quad \quad \quad \text{else do } \{ \text{swap } xa \ j \ (k-1); \\
\quad \quad \quad \quad \text{loop } (j,k-1) \} \} \\
\quad \text{in } \text{loop } (a+1,b) \}
\]
The action `swap` is defined by

```haskell
swap :: STArray s Int a -> Int -> Int -> ST s ()
swap xa i j = do {v <- readArray xa i;
                 w <- readArray xa j;
                 writeArray xa i w;
                 writeArray xa j v}
```

Here is a brief and certainly inadequate explanation of how `partition` works. We begin by taking the first element \( x \) in the interval \((a, b)\) as pivot. We then enter a loop that processes the remaining interval \((a+1, b)\), stopping when the interval becomes empty. We pass over elements that are less than \( x \), shrinking the interval from the left. Encountering a \( y \) not less than \( x \), we swap it with the element at the rightmost position in the interval, shrinking the interval from the right. When the interval becomes empty, we place the pivot in its final position, returning that position as a result.

Note that `loop` is defined as a local procedure within the monad. We could have defined it as a global procedure, though we would have had to add three extra parameters, namely the array \( xa \), the pivot \( x \) and the starting position \( a \).

---

### Hash tables

A purely functional Quicksort has the same asymptotic time efficiency as one based on mutable arrays, but there are one or two places where mutable arrays seem to play a crucial role in achieving an asymptotically faster algorithm. One such place is the use of hash tables for an efficient representation of sets.

But let us approach the use of hash tables in the context of a particular problem. Consider a typical puzzle defined in terms of two finite sets, a set of positions and a set of moves. Given are the following functions:

```haskell
moves :: Position -> [Move]
moves :: Position -> Move -> Position
solved :: Position -> Bool
```

The function `moves` describes the set of possible moves that can be made in a given position, `move` makes a move, and `solved` determines those positions that are a solution to the puzzle. Solving the puzzle means finding some sequence of moves, preferably a shortest such sequence, that leads from a given starting position to a solved position:
solve :: Position -> Maybe [Move]

The value \( \text{solve} \ p \) is \text{Nothing} if there is no sequence of moves starting in position \( p \) that leads to a solved position, and \text{Just} \ ms \ otherwise, where

\[
\text{solve} \ (\text{foldl} \ \text{move} \ p \ ms)
\]

We are going to implement \text{solve} \ by carrying out a \textit{breadth-first} search. What this means is that we examine all positions one move away from the starting position to see if there is a solution, then all positions two moves away, and so on. Breadth-first will therefore find a shortest solution if one exists. To implement the search we need

\[
\text{type} \ \text{Path} \ = \ (\text{[Move]},\text{Position}) \\
\text{type} \ \text{Frontier} \ = \ \text{[Path]}
\]

A path consists of a sequence of moves made from the starting position (in reverse order), and the position that results after making the moves. A frontier is a list of paths waiting to be extended into longer paths. A breadth-first search is then implemented by

\[
\text{solve} \ p \ = \ \text{bfs} \ \text{[]} \ \text{[([],p)]}
\]

\[
\text{bfs} \ :\ : \ [\text{Position}] \ \rightarrow \ \text{Frontier} \ \rightarrow \ \text{Maybe} \ [\text{Move}]
\]

\[
\text{bfs} \ \text{ps} \ \text{[]} \ = \ \text{Nothing} \\
\text{bfs} \ \text{ps} \ \text{((ms,p):mps)} \ | \ \text{solved} \ p \ = \ \text{Just} \ (\text{reverse} \ \text{ms}) \\
\text{p `elem` ps} \ = \ \text{bfs} \ \text{ps} \ \text{mps} \\
\text{otherwise} \ = \ \text{bfs} \ (\text{p:ps}) \ (\text{mps} \ \text{++} \ \text{successes} \ (\text{ms,p}))
\]

\[
\text{successes} \ :\ : \ \text{Path} \ \rightarrow \ \text{[Path]}
\]

\[
\text{successes} \ (\text{ms,p}) \ = \ [(\text{m:ms},\text{move} \ p \ \text{m}) \ | \ \text{m} \ \text{<-} \ \text{moves} \ p]
\]

The first argument \( \text{ps} \) of \text{bfs} \ represents the set of positions that have already been explored. The second argument is the frontier, which is managed in a queue-like fashion to ensure that paths of the same length are inspected before their successors. Inspecting a path means accepting it if the final position is a solution, rejecting it if the end position has already been explored, and otherwise adding its successors to the end of the current frontier for future exploration. The moves in a successful path are reversed before being returned as the final result of \text{bfs}. simply because, for efficiency, \text{successes} \ adds a new move to the front of the list rather than at the end.

There are two major sources of inefficiency with \text{bfs}, one concerning the use of
(++) and the other concerning \texttt{elem}. Firstly, the size of a frontier can grow exponentially and so concatenating successors to the end of the frontier is slow. Better is the following alternative to \texttt{bfs}:

\begin{verbatim}
  bfs :: [Position] -> Frontier -> Frontier ->
    Maybe [Move]
  bfs ps [] [] = Nothing
  bfs ps [] mqs = bfs ps mqs []
  bfs ps ((ms,p):mps) mqs
    | solved p = Just (reverse ms)
    | p `elem` ps = bfs ps mps mqs
    | otherwise = bfs (p:ps) mps (succs (ms,p) ++ mqs)
\end{verbatim}

The additional argument is a temporary frontier used to store successors. When the first frontier is exhausted the contents of the temporary frontier are installed as the new frontier. Adding successors to the front of the temporary frontier takes time proportional to the number of successors, not to the size of the frontier, and that leads to a faster algorithm. On the other hand, the new version of \texttt{bfs} is not the same as the old one because successive frontiers are traversed alternately from left to right and from right to left. Nevertheless a shortest solution will still be found if one exists.

The second source of inefficiency is the membership test. Use of a list to store previously explored positions is slow because the membership test can take time proportional to the number of currently explored positions. It would all be easier if positions were integers in the range $[0..n-1]$ for some $n$, for then we could use a boolean array with bounds $(0,n-1)$ to tick off positions as they arise. The membership test would then consist of a single array lookup.

One can imagine coding positions as integers, but not as integers in an initial segment of the natural numbers. For instance, a Sudoku position (see Chapter 5) can be expressed as an integer consisting of 81 digits. So suppose we have a function

\begin{verbatim}
  encode :: Position -> Integer
\end{verbatim}

that encodes positions as integers. To reduce the range we can define

\begin{verbatim}
  hash :: Position -> Int
  hash p = fromInteger (encode p) `mod` n
\end{verbatim}

for some suitable $n : : \text{Int}$. The result of \texttt{hash} is then an integer in the range $[0..n-1]$.

The one hitch, and it’s a big one, is that two distinct positions may hash to the
same integer. To solve this problem we abandon the idea of having an array of booleans, and instead have an array of lists of positions. The positions in the array at index $k$ are all those whose hash value is $k$. There is no guarantee that any of this will improve efficiency in the worst case, but if we allow $n$ to be reasonably large, and trust that the hash function assigns integers to positions in a reasonably evenly distributed way, then the complexity of a membership test is reduced by a factor of $n$.

With this hashing scheme the revised code for `solve` is:

```haskell
solve :: Maybe [Move]
solve = runST $ do {pa <- newArray (0,n-1) []; 
    bfs pa [[[],start]] []}

bfs :: STArray s Int [Position] -> Frontier -> Frontier -> ST s (Maybe [Move])
bfs pa [] [] = return Nothing 
bfs pa [] mqs = bfs pa mqs []
bfs pa ((ms,p):mps) mqs = if solved p then return (Just (reverse ms))
    else do {ps <- readArray pa k; 
    if p `elem` ps 
        then bfs pa mps mqs 
        else 
            do {writeArray pa k (p:ps); 
                 bfs pa mps (succe (ms,p) ++ mqs))}
    where k = hash p
```

10.6 Immutable arrays

We cannot leave the subject of arrays without mentioning a very nice Haskell library `Data.Array` that provides purely functional operations on immutable arrays. The operations are implemented using mutable arrays, but the interface is purely functional.

The type `Array i e` is an abstract type of arrays with indices of type `i` and elements of type `e`. One basic operation for constructing arrays is

```haskell
array :: Ix i => (i,i) -> ((i,e)) -> Array i e
```
This function take a pair of bounds, the lowest and highest indices in the array, and a list of index-element pairs specifying the array entries. The result is an array with the given bounds and entries. Any entry missing from the association list is deemed to be the undefined entry. If two entries have the same index, or one of the indices is out of bounds, the undefined array is returned. Because of these checks, array construction is strict in the indices, though lazy in the elements. Building the array takes linear time in the number of entries.

A simple variant of `array` is `listArray` which takes just a list of elements:

\[\text{listArray :: Ix i => (i,i) -> [e] -> Array i e}\]

\[\text{listArray (l,r) xs = array (l,r) (zip [l..r] xs)}\]

Finally, there is another way of building arrays called `accumArray` whose type appears rather daunting:

\[\text{Ix i => (e -> v -> e) -> e -> (i,i) -> [(i,v)] -> Array i e}\]

The first argument is an ‘accumulating’ function for transforming array entries and new values into new entries. The second argument is an initial entry for each index. The third argument is a pair of bounds, and the fourth and final argument is an association list of index–value pairs. The result is an array built by processing the association list from left to right, combining entries and values into new entries using the accumulating function. The process takes linear time in the length of the association list, assuming the accumulating function takes constant time.

That’s what `accumArray` does in words. In symbols,

\[\text{elems (accumArray f e (l,r) ivs)}\]

\[\text{= [foldl f e [v | (i,v) <- ivs, i==j | j <- [l..r]]}\]

where `elems` returns the list of elements of an array in index order. Well, the identity above is not quite true: there is an additional restriction on `ivs`, namely that every index should lie in the specified range. If this condition is not met, then the left-hand side returns an error while the right-hand side does not.

Complicated as `accumArray` seems, it turns out to be a very useful tool for solving certain kinds of problem. Here are two examples. First, consider the problem of representing directed graphs. Directed graphs are usually described in mathematics in terms of a set of vertices and a set of edges. An edge is an ordered pair \((j,k)\) of vertices signifying that the edge is directed from \(j\) to \(k\). We say that \(k\) is adjacent to \(j\). We will suppose that vertices are named by integers in the range \(1\) to \(n\) for some \(n\). Thus

\[\text{type Vertex = Int}\]
type Edge = (Vertex, Vertex)
type Graph = ([Vertex], [Edge])

vertices g = fst g
edges g = snd g

In computing, directed graphs are often described in terms of adjacency lists:

adjs :: Graph -> Vertex -> [Vertex]
adjs g v = [k | (j, k) <- edges g, j == v]

The problem with this definition of \textit{adjs} is that it takes time proportional to the number of edges to compute the adjacency list of any particular vertex. Better is to implement \textit{adjs} as an array:

adjArray :: Graph -> Array Vertex [Vertex]

Then we have

adjs g v = (adjArray g)!(v)

where (!) denotes the operation of array-indexing. For reasonably sized arrays this operation takes constant time.

The specification of \textit{adjArray} is that

elems (adjArray g) = [[k | (j, k) <- edges g, j == v] \mid v \leftarrow vertices g]

Using this specification we can calculate a direct definition of \textit{adjArray}. To keep each line short, abbreviate \textit{edges g} to \textit{es} and \textit{vertices g} to \textit{vs}, so

elems (adjArray g) = [[k | (j, k) <- es, j == v] \mid v \leftarrow vs]

Concentrating on the right-hand side, the first step is to rewrite it using the law \textit{foldr} (\textit{\_\_\_}) [] = id. That gives the expression

\[\text{[foldr (\textit{\_\_\_}) [] [k \mid (j, k) \leftarrow es, j == v] \mid v \leftarrow vs]}\]

Next we use the law \textit{foldr} \textit{f} \textit{e} \textit{x} = \textit{foldl} (\textit{flip f}) \textit{e} (reverse \textit{x} \textit{s}) for all finite lists \textit{x}s. Abbreviating \textit{flip} (\textit{\_\_\_}) to (\textit{\_\_\_}), we obtain

\[\text{[foldl (\_\_\_ e) [] (reverse [k \mid (j, k) \leftarrow es, j == v]) \mid v \leftarrow vs]}\]

Distributing \textit{reverse} we obtain the expression

\[\text{[foldl (\_\_\_ e) [] [k \mid (j, k) \leftarrow reverse es, j == v] \mid v \leftarrow vs]}\]

Next we use \textit{swap} (\textit{j}, \textit{k}) = (\textit{k}, \textit{j}) to obtain
Imperative functional programming

\[
\text{foldl} \ (\@) \ [] \ [j \mid (k,j) \leftarrow es', \ j==v] \mid v \leftarrow vs
\]

where \( es' = \text{map} \ \text{swap} \ (\text{reverse} \ es) \). Finally, using \( n = \text{length} \ \text{vs} \) and the specification of \( \text{accumArray} \), we obtain

\[
\text{elems} \ (\text{adjArray} \ g) \\
= \text{elems} \ (\text{accumArray} \ (\text{flip} \ :) \ [] \ (1,n) \ es')
\]

That means we can define

\[
\text{adjArray} \ g = \text{accumArray} \ (\text{flip} \ :) \ [] \ (1,n) \ es
\]

where \( n = \text{length} \ (\text{vertices} \ g) \)
\[
es = \text{map} \ \text{swap} \ (\text{reverse} \ (\text{edges} \ g))
\]

This definition of \( \text{adjArray} \ g \) computes the successors in time proportional to the number of edges.

Here is the second example of the use of \( \text{accumArray} \). Suppose we are given a list of \( n \) integers, all in the range \((0,m)\) for some \( m \). We can sort this list in \( \Theta(m+n) \) steps by counting the number of times each element occurs:

\[
\text{count} :: [\text{Int}] \rightarrow \text{Array} \ \text{Int} \ \text{Int} \\
\text{count} \ \text{xs} = \text{accumArray} \ (\text{+}) \ 0 \ (0,m) \ (\text{zip} \ \text{xs} \ (\text{repeat} \ 1))
\]

The value \( \text{repeat} \ 1 \) is an infinite list of 1s. Counting takes \( \Theta(n) \) steps. Having counted the elements, we can now sort them:

\[
\text{sort} \ \text{xs} = \text{concat} \ [\text{replicate} \ c \ x \mid (x,c) \leftarrow \text{assocs} \ (\text{count} \ \text{xa})]
\]

The function \( \text{assocs} \) is yet another library function and returns the list of index–element pairs of an array in index order. The sorting is completed in \( \Theta(m) \) steps.

As well as the above operations \( \text{Data.Array} \) contains one or two more, including the update operation (\( // \)):

\[
(//) :: \text{Ix} \ i \Rightarrow \text{Array} \ i \ e \rightarrow [(i,e)] \rightarrow \text{Array} \ i \ e
\]

For example, if \( \text{xa} \) is an \( n \times n \) matrix, then

\[
\text{xa} \ // [(i,i),0] \mid i \leftarrow [1..n]
\]

is the same matrix except with zeros along the diagonal. The downside of (\( // \)) is that it takes time proportional to the size of the array, even for an update involving a single element. The reason is that a completely new array has to be constructed because the old array \( \text{xa} \) continues to exist.

We have ended the chapter back in the world of pure functional programming,
where equational reasoning can be used both to calculate definitions and to optimise them. Although the monadic style is attractive to programmers who are used to imperative programming, there remains the problem of how to reason about monadic programs. True, equational reasoning is still possible in certain situations (see Exercise F for an example), but it is not so widely applicable as it is in the pure functional world (witness the correctness of the partition phase of Quicksort). Imperative programmers have the same problem, which they solve (if they bother to) by using predicate calculus, preconditions, postconditions and loop invariants. How to reason directly with monadic code is still a topic of ongoing research.

Our best advice is to use the monadic style sparingly and only when it is really useful; otherwise the most important aspect of functional programming, the ability to reason mathematically about its constructs, is lost.

10.7 Exercises

**Exercise A**

Recall that

\[
\text{putStr} = \text{foldr} \ (\gg) \ \text{done} \ . \ \text{map} \ \text{putChar}
\]

What does

\[
\text{foldl} \ (\gg) \ \text{done} \ . \ \text{map} \ \text{putChar}
\]

do? Justify your answer by expressing \(\gg\) in terms of \((\gg=)\) and appealing to the monad laws.

**Exercise B**

Using a pattern-matching style, define a function

\[
\text{add3} :: \text{Maybe Int} \to \text{Maybe Int} \to \text{Maybe Int} \to \text{Maybe Int}
\]

that adds three numbers, provided all of them exist. Now rewrite \text{add3} using the \text{Maybe} monad.

**Exercise C**

The monadic definition of \text{cp} in Section 10.1 is still inefficient. We might prefer to write
Imperative functional programming

\[ cp\ (xs:xss) = do\ \{ ys \leftarrow cp\ xss; \]
\[ x \leftarrow xs; \]
\[ return\ (x:ys)\]\n
By definition a **commutative** monad is one in which the equation

\[ do\ \{ x \leftarrow p; \ y \leftarrow q; \ f\ x\ y\} \]
\[ = do\ \{ y \leftarrow q; \ x \leftarrow p; \ f\ x\ y\}\]

holds. The IO monad is certainly not commutative, while some other monads are. Is the Maybe monad commutative?

**Exercise D**

Every monad is a functor. Complete the definition

\[
\text{instance}\ \text{Monad}\ m\Rightarrow\ \text{Functor}\ m\ \text{where}
\]
\[
fmap\ :\ (a\rightarrow b)\rightarrow\ m\ a\rightarrow\ m\ b
\]
\[
fmap\ f\ =\ ...\]

Currently Haskell does not insist that the Monad class should be a subclass of Functor, though there are plans to change this in future releases. Instead, Haskell provides a function \text{liftM} equivalent to \text{fmap} for monads. Give a definition of \text{liftM} in terms of \text{return} and \text{>>=}.

The function \text{join} :: \text{m (m a)} -> \text{m a} flattens two layers of monadic structure into one. Define \text{join} in terms of \text{>>=}.

What familiar functions do \text{join} and \text{liftM} give for the list monad?

Finally, using \text{join} and \text{liftM}, define \text{>>(=)}. It follows that instead of defining monads in terms of \text{return} and \text{>>=} we can also define them in terms of \text{return}, \text{liftM} and \text{join}.

**Exercise E**

A number of useful monadic functions are provided in the \text{Control.Monad} library. For instance:

\[
\text{sequence} x :: \text{Monad}\ m\Rightarrow\ [m\ a]\rightarrow\ m\ ()
\]
\[
\text{sequence} x = \text{foldr} (\text{>>=})\ \text{done}
\]

(The underscore convention is used in a number of places in Haskell to signify that the result of the action is the null tuple.) Define the related function

\[
\text{sequence} :: \text{Monad}\ m\Rightarrow\ [m\ a]\rightarrow\ m\ [a]
\]

Using these two functions, define
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM :: Monad m => (a -> m b) -> [a] -> m [b]

Also, define

foldM :: Monad m => (b -> a -> m b) -> b -> [a] -> m b

In the text we made use of a function repeatFor n that repeated an action n times.
Generalise this function to

for_ :: Monad m => [a] -> (a -> m b) -> m ()

**Exercise F**

Here is an exercise in monadic equational reasoning. Consider the function

add :: Int -> State Int ()
add n = do {m <- get; put (m+n)}

The task is to prove that

sequence_ . map add = add . sum

where sequence_ was defined in the previous exercise and sum sums a list of integers. You will need the fusion law of foldr, some simple laws of put and get, and the monad law

do {stmts1} >>= do {stmts2} = do {stmts1;stmts2}

which is valid provided the variables in stmts1 and stmts2 are disjoint.

**Exercise G**

Prove the leapfrog rule: (f ==> g) . h = (f . h) ==> g.

Using this rule, prove: (return . h) ==> g = g . h.

**Exercise H**

Prove that

liftM f = id ==> (return . f)
join = id ==> id

A fourth way of describing the monad laws is in terms of the two functions liftM and join of Exercise D. There are seven laws governing these two functions, all of which have a familiar ring:
Imperative functional programming

liftM id = id
liftM (f . g) = liftM f . liftM g

liftM f . return = return . f
liftM f . join = join . liftM (liftM f)

join . return = id
join . liftM return = id
join . liftM join = join . join

Prove the fourth rule.

**Exercise I**

What does \( \text{build} [] \) do (see Section 10.3)?

**Exercise J**

Write an interactive program to play hangman. An example session:

```
ghci> hangman
I am thinking of a word:
-----
Try and guess it.
guess: break
-a---
guess: parties
Wrong number of letters!
guess: party
-appy
guess: happy
You got it!
Play again? (yes or no)
no
Bye!
```

Assume that a list of secret words is stored in a file called `Words`, so that the action \( \text{xs <- readFile "Words"} \) reads the file as a list of characters. By the way, `readFile` is lazy in that its contents are read on demand.

**Exercise K**

Write another version of \( \text{fib} \) in terms of a `fibST` that uses a single `STRef`. 
Exercise L

One way of defining the greatest common divisor (gcd) of two positive integers is:

\[
\text{gcd}(x,y) \begin{cases} 
  x == y & \Rightarrow x \\
  x < y & \Rightarrow \text{gcd}(x, y-x) \\
  x > y & \Rightarrow \text{gcd}(x-y, y) 
\end{cases}
\]

Translate this definition into two other programs, one of which uses the \texttt{State} monad and the other the \texttt{ST} monad.

Exercise M

Here is a concrete puzzle you can solve using breadth-first search. A cut-down version of Sam Loyd’s famous 15 puzzle is the 8 puzzle. You are given a 3 × 3 array containing tiles numbered from 1 to 8 and one blank space. You move by sliding an adjacent tile into the blank space. You move by sliding an adjacent tile into the blank space. Depending on where the blank space is, you can slide tiles upwards, downwards, to the left or to the right. At the start the blank space is in the top left corner and the tiles read from 1 to 8. At the end the blank space is in the bottom right corner, but the tiles are still neatly arranged in the order 1 to 8.

Your mission, should you choose to accept it, is to settle on a suitable representation of positions and moves, and to define the functions \texttt{moves}, \texttt{move}, \texttt{solved} and \texttt{encode}.

Answer to Exercise A

We claim that \( (>>) :: \text{IO }() \to \text{IO }() \to \text{IO }() \) is associative with identity element \texttt{done}. That means

\[
\text{putStr } xs = \text{foldl } (>>) \text{ done } (\text{map putChar } xs)
\]

for all finite strings \( xs \)

We concentrate on the proof of associativity. Firstly, for actions in \( \text{IO }() \) we have

\[
p >> q = p >>= \text{ const } q
\]
where \( \text{const } x y = x \). Now we can reason:

\[
(p >> q) >> r = \{ \text{definition of } (>>) \} \quad (p >>= \text{const } q) >>= \text{const } r = \{ \text{third monad law} \} \quad p >>= \text{const } (q >>= \text{const } r) = \{ \text{definition of } (>>) \} \quad p >>= \text{const } (q >> r) = \{ \text{definition of } (>>) \} \quad p >> (q >> r)
\]

**Answer to Exercise B**

The direct version uses pattern matching with a wild-card:

\[
\text{add3 Nothing } _ _ _ = \text{Nothing} \\
\text{add3 (Just } x) \text{Nothing } _ _ _ = \text{Nothing} \\
\text{add3 (Just } x) \text{ (Just } y) \text{Nothing } = \text{Nothing} \\
\text{add3 (Just } x) \text{ (Just } y) \text{ (Just } z) = \text{Just } (x+y+z)
\]

This definition ensures that \( \text{add Nothing undefined } = \text{Nothing} \).

The monadic version reads:

\[
\text{add3 } mx \text{ my mz } = \text{do } \{ x <- mx; y <- my; z <- mz; \text{return } (x + y + z) \}
\]

**Answer to Exercise C**

Yes. The commutative law states that

\[
p >>= \lambda x \to q >>= \lambda y \to f x y = q >>= \lambda y \to p >>= \lambda x \to f x y
\]

In the \texttt{Maybe} monad there are four possible cases to check. For example, both sides simplify to \texttt{Nothing} if \( p = \text{Nothing} \) and \( q = \text{Just } y \),. The other cases are similar.

**Answer to Exercise D**

We have
\[ \text{fmap } f \ p = p >>= (\text{return } . f) \]
\[ \text{join } p = p >>= \text{id} \]

For the list monad we have \( \text{liftM} = \text{map} \) and \( \text{join} = \text{concat} \).

In the other direction
\[ p >>= f = \text{join} (\text{liftM } f \ p) \]

**Answer to Exercise E**

The function \( \text{sequence} \) is defined by

\[
\text{sequence} :: \text{Monad } m \Rightarrow [m a] \rightarrow m [a] \\
\text{sequence} = \text{foldr } k (\text{return } []) \\
\text{where } k p q = \text{do } \{ x \leftarrow p; \ xs \leftarrow q; \text{return } (x:xs) \}
\]

The two new map functions are:

\[
\text{mapM}_- f = \text{sequence}_- . \text{map } f \\
\text{mapM } f = \text{sequence } . \text{map } f
\]

The function \( \text{foldM} \) is defined by

\[
\text{foldM} :: \text{Monad } m \Rightarrow (b \rightarrow a \rightarrow m b) \rightarrow b \rightarrow [a] \rightarrow m b \\
\text{foldM } f \ e \ [] = \text{return } e \\
\text{foldM } f \ e \ (x:xs) = \text{do } \{ y \leftarrow f \ e \ x; \text{foldM } f \ y \ xs \}
\]

Note that \( \text{foldM} \) is analogous to \( \text{foldl} \) in that it works from left to right. Finally \( \text{for} = \text{flip } \text{mapM}_- \).

**Answer to Exercise F**

The first thing to note is that

\[
\text{sequence}_- . \text{map } \text{add} \\
= \text{foldr } (\gg) \text{ done } . \text{map } \text{add} \\
= \text{foldr } ((\gg) . \text{add}) \text{ done}
\]

using the fusion law of \( \text{foldr} \) and \( \text{map} \) given in Section 6.3. Moreover,

\[(\gg) . \text{add} \ n \ p = \text{add } n \gg p\]

Since \( \text{sum} = \text{foldr } (+) 0 \) that means we have to prove

\[ \text{foldr } (\backslash n \ p \rightarrow \text{add } n \gg p) = \text{add } . \text{foldr } (+) 0 \]
That looks like an instance of the fusion law of `foldr`. We therefore have to show that `add` is strict (which it is), and

\[
\begin{align*}
\text{add } 0 &= \text{done} \\
\text{add } (n + n') &= \text{add } n \gg \text{add } n'
\end{align*}
\]

Here goes:

\[
\begin{align*}
\text{add } 0 &= \\
&= \{\text{definition}\} \\
&\quad \begin{aligned}
&\begin{aligned}
&\begin{aligned}
&\text{do } \{ m \leftarrow \text{get}; \text{put } (m+0) \}
&= \{\text{arithmetic}\} \\
&\quad \begin{aligned}
&\begin{aligned}
&\text{do } \{ m \leftarrow \text{get}; \text{put } m \}
&= \{\text{simple law of put and get}\}
&\text{done}
\end{aligned}
\end{aligned}
\end{aligned}
\end{aligned}
\end{aligned}
\end{align*}
\]

That disposes of the first condition. For the second we start with the more complicated side and reason:

\[
\begin{align*}
\text{add } n \gg \text{add } n' &= \\
&= \{\text{definition}\} \\
&\quad \begin{aligned}
&\begin{aligned}
&\begin{aligned}
&\text{do } \{ l \leftarrow \text{get}; \text{put } (l + n) \} \gg
&\quad \begin{aligned}
&\begin{aligned}
&\text{do } \{ m \leftarrow \text{get}; \text{put } (m + n') \}
&= \{\text{monad law}\} \\
&\quad \begin{aligned}
&\begin{aligned}
&\text{do } \{ l \leftarrow \text{get}; \text{put } (l + n); m \leftarrow \text{get}; \text{put } (m + n') \}
&= \{\text{simple law of put and get}\} \\
&\quad \begin{aligned}
&\begin{aligned}
&\text{do } \{ l \leftarrow \text{get}; \text{put } ((l + n) + n') \}
&= \{\text{associativity of (+); definition of add}\} \\
&\text{add } (n + n')
\end{aligned}
\end{aligned}
\end{aligned}
\end{aligned}
\end{aligned}
\end{aligned}
\end{aligned}
\end{aligned}
\end{align*}
\]

**Answer to Exercise G**

We can reason:

\[
\begin{align*}
(f \gg g)(h \ x) &= \\
&= \{\text{definition of } (\gg\gg)\} \\
f \ (h \ x) \gg g &= \\
&= \{\text{definition of } (\gg\gg)\} \\
(f \ . \ h) \gg g \ x &=
\end{align*}
\]
For the second part:

\[
\begin{align*}
\text{(return . h) >>= g} \\
= & \{\text{leapfrog rule}\} \\
\text{(return >>= g) . h} \\
= & \{\text{monad law}\} \\
\text{g . h}
\end{align*}
\]

**Answer to Exercise H**

For the fourth rule we simplify both sides. For the left-hand side:

\[
\begin{align*}
\text{liftM f . join} \\
= & \{\text{definitions}\} \\
\text{(id >>= (return . f)) . (id >>= id)} \\
= & \{\text{leapfrog rule and id . f = f}\} \\
\text{(id >>= id) >>= (return . f)}
\end{align*}
\]

For the right-hand side:

\[
\begin{align*}
\text{join . liftM (liftM f)} \\
= & \{\text{definitions}\} \\
\text{(id >>= id) . (id >>= return . (id >>= (return . f)))} \\
= & \{\text{leapfrog rule, and associativity of (>>=)}\} \\
\text{id >>= (return . (id >>= (return . f))) >>= id} \\
= & \{\text{since (return . h) >>= g = g . h}\} \\
\text{id >>= id >>= (return . f)}
\end{align*}
\]

The two sides are equal because (>>=) is associative.

**Answer to Exercise I**

build [] causes an infinite loop, so its value is \(\perp\).

**Answer to Exercise J**

For the main function we can define

```haskell
hangman :: IO ()
hangman = do {xs <- readFile "Words";
    play (words xs)}
```
The function `play` plays as many rounds of the game as desired with different words from the file (which we quietly suppose always has enough words):

```haskell
def play (w:ws) = do 
  putStrLn "I am thinking of a word:";
  putStrLn (replicate (length w) '-');
  putStrLn "Try and guess it."
  guess w ws
```

The function `guess` deals with a single guess, but keeps the remaining words for any subsequent round of play:

```haskell
def guess w ws = do 
  putStrLn "guess: ";
  w' <- getLine;
  if length w' /= length w then 
    do {putStrLn "Wrong number of letters!";
         guess w ws}
  else if w' == w then 
    do 
      putStrLn "You got it!"
      putStrLn "Play again? (yes or no)"
      ans <- getLine;
      if ans == "yes" 
        then play ws 
        else putStrLn "Bye!"
  else do 
    putStrLn (match w' w);
    guess w ws
```

Finally we program `match`:

```haskell
match w' w = map check w
  where
    check x = if x `elem` w' then x else '-'
```

**Answer to Exercise K**

The following program is correct but doesn’t run in constant space:

```haskell
fib n = fst $ runST (fibST n)

fibST :: Int -> ST s (Integer,Integer)
fibST n = do 
  ab <- newSTRef (0,1);
repeatFor n
  (do {(a,b) <- readSTRef ab;
       writeSTRef ab $! (b,a+b));
    readSTRef ab}

The reason is that \((b,a+b)\) is already in head-normal form, so strict-apply has no effect. The penultimate line needs to be changed to

\[
b \ `seq` (a+b) `seq` writeSTRef ab (b,a+b)
\]

in order to force evaluation of the components.

**Answer to Exercise L**

The version that uses the \textit{State} monad:

\[
gcd (x,y) = \text{fst} \$ \text{runState} \text{loop} (x,y)
\]

\[
\text{loop :: State (Int,Int) Int}
\]

\[
\text{loop = do \{(x,y) <- get;
    if x == y
    then return x
    else if x < y
    then do \{put (x,y-x); loop\}
    else do \{put (x-y,y); loop\}\}
\]

The version that uses the \textit{ST} monad:

\[
gcd (x,y) = \text{runST} \$
\]

\[
\text{do \{a <- newSTRef x;
    b <- newSTRef y;
    loop a b\}}
\]

\[
\text{loop :: STRef s Int -> STRef s Int -> ST s Int}
\]

\[
\text{loop a b}
\]

\[
\text{= do \{x <- readSTRef a;
    y <- readSTRef b;
    if x==y
    then return x
    else if x<y
    then do \{writeSTRef b (y-x);loop a b\}
    else do \{writeSTRef a (x-y);loop a b\}\}}
Answer to Exercise M

There are, of course, many possible answers. The one I chose was to represent the array of tiles by a list of nine digits \([0..8]\) with zero representing the space. To avoid recalculation, a position is represented by a pair \((j,ks)\) with \(j\) as the position of the zero in \(ks\), where \(ks\) was some permutation of \([0..8]\). Thus:

```haskell
type Position = (Int,[Int])
data Move = Up | Down | Left | Right

encode :: Position -> Integer
encode (j,ks) = foldl op 0 ks
  where op x d = 10*x + fromIntegral d

start :: Position
start = (0,[0..8])
```

The function `moves` can be defined by

```haskell
moves :: Position -> [Move]
moves (j,ks)
  = [Up | j `notElem` [6,7,8]] ++
    [Down | j `notElem` [0,1,2]] ++
    [Left | j `notElem` [2,5,8]] ++
    [Right | j `notElem` [0,3,6]]
```

Up moves are allowed except for a blank in the bottom row; down moves except for a blank in the top row, left moves except for a blank in the rightmost column, and right moves except for a blank in the leftmost column.

The function `move` can be defined by:

```haskell
move :: Position -> Move -> Position
move (j,ks) Up = (j+3,swap (j,j+3) ks)
move (j,ks) Down = (j-3,swap (j-3,j) ks)
move (j,ks) Left = (j+1,swap (j,j+1) ks)
move (j,ks) Right = (j-1,swap (j-1,j) ks)

swap (j,k) ks = ks1 ++ y:ks3 ++ x:ks4
  where (ks1,x:ks2) = splitAt j ks
        (ks3,y:ks4) = splitAt (k-j-1) ks2
```

Finally,
solved :: Position -> Bool
solved p = p == (8,[1,2,3,4,5,6,7,8,0])

My computer produced:

ghci> solve start
Just [Left,Up,Right,Up,Left,Left,Down,
    Right,Right,Up,Left,Down,Down,Left,
    Up,Up,Right,Right,Down,Left,Left,Up]
(4.84 secs, 599740496 bytes)

10.9 Chapter notes

Read *The History of Haskell* to see how monads came to be an integral part of
Haskell, and why this idea has been mainly responsible for the increasing use of
Haskell in the real world. Monads are used to structure GHC, which itself is written
in Haskell. Each phase of the compiler uses a monad for book-keeping information.
For instance, the type checker uses a monad that combines state (to maintain a
current substitution), a name supply (for fresh type variable names) and exceptions.

Use of do-notation in preference to (>>=) was suggested by John Launchbury in
1993 and was first implemented by Mark Jones in Gofer.

The number of tutorials on monads has increased steadily over the years; see

haskell.org/haskellwiki/Monad_tutorials

for a reasonably comprehensive list.

The example (in Exercise F) of monadic equational reasoning can be found in the
paper ‘Unifying theories of programming with monads’. (UTP Symposium, Au-
gust 2012) by Jeremy Gibbons. For additional material on reasoning equationally
with monads, read ‘Just do it: simple monadic equational reasoning’ by Jeremy
Gibbons and Ralf Hinze, which appeared in the proceedings of the 2011 Interna-
tional Conference of Functional Programming. Both papers can be found at

www.cs.ox.ac.uk/people/jeremy.gibbons/publications/
A parser is a function that analyses a piece of text to determine its logical structure. The text is a string of characters describing some value of interest, such as an arithmetic expression, a poem or a spreadsheet. The output of a parser is a representation of the value, such as a tree of some kind for an arithmetic expression, a list of verses for a poem, or something more complicated for a spreadsheet. Most programming tasks involve decoding the input in some way, so parsing is a pervasive component of computer programming. In this chapter we will describe a monadic approach to parsing, mainly designing simple parsers for expressions of various kinds. We will also say a little more about the converse process of encoding the output as a string; in other words, more about the type class Show. This material will be used in the final chapter.

11.1 Parsers as monads

Parsers return different values of interest, so as a first cut we can think of a parser as a function that takes a string and returns a value:

\[
\text{type Parser } a = \text{String } \rightarrow \ a
\]

This type is basically the same as that of the standard prelude function

\[
\text{read} :: \text{Read } a \Rightarrow \text{String } \rightarrow \ a
\]

Indeed, read is a parser, though not a very flexible one. One reason is that all the input must be consumed. Thus:

\[
\text{ghci} > \text{read } "123" :: \text{Int}
123
\]
With `read` there is no obvious way of reading two or more things in sequence. For example, in a parser for arithmetic expressions we may want to look in the input stream for a numeral, then an operator and then another numeral. The first parser for a numeral will consume some prefix of the input, the parser for an operator some prefix of the remaining input, and the third parser yet more input. A better idea is to define a parser as a function that consumes a prefix of the input and returns both a value of interest and the unconsumed suffix:

```haskell
type Parser a = String -> (a, String)
```

We are not quite there yet. It can happen that a parser may fail on some input. It is not a mistake to construct parsers that can fail. For example, in a parser for arithmetic expressions, we may want to look for either a numeral or an opening parenthesis. One or either of these subsidiary parsers will certainly fail. Failure should not be thought of as an error that terminates the parsing process; rather it acts like an identity element for an operation that chooses between alternatives. More generally, a parser may find a number of different ways that some prefix of the input can be structured. Failure then corresponds to the particular case of the empty sequence of parses. In order to handle these various possibilities, we change our definition yet again and define

```haskell
type Parser a = String -> [(a, String)]
```

The standard prelude provides exactly this type synonym, except that it is called `ReadS`, not `Parser`. And it also provides a function

```haskell
reads :: Read a => ReadS a
```

as a subsidiary method in the type class `Read`. For example,

```haskell
ghci> reads "-123+51" :: [(Int, String)]
[(-123, "+51")]
ghci> reads "+51" :: [(Int, String)]
[]
```

As with the function `read` you have to tell `reads` the type you are expecting. The second example fails, returning no parses, because a Haskell integer can be preceded by an optional minus sign but not by an optional plus sign. By definition, a parser is deterministic if it returns an empty or singleton list of parses in all possible cases. In particular, instances of `reads` ought to be deterministic parsers.
There is one further change we have to make to the definition of \texttt{Parser}. We would like to install this type as an instance of the \texttt{Monad} class, but that is not possible. The reason is that \texttt{Parser} is declared as a type synonym, and type synonyms cannot be made members of any type class: they inherit whatever instances are declared for the underlying type. A type synonym is there simply to improve readability in type declarations; no new types are involved and we cannot construct two different type class instances for what is essentially the same type.

One way to construct a new type is by a data declaration:

\begin{verbatim}
data Parser a = Parser (String -> [(a,String)])
\end{verbatim}

The identifier \texttt{Parser} on the right is a constructor, while on the left it is the name of a new type. Most people are happy with the pun; others would rename the constructor as something like \texttt{MkParser} or just \texttt{P}.

There is a better way to create a new type for \texttt{Parser} and that is to use a \texttt{newtype} declaration:

\begin{verbatim}
newtype Parser a = Parser (String -> [(a,String)])
\end{verbatim}

We have not needed \texttt{newtype} declarations up to now, so let us digress a little to explain them. The price paid for using a data declaration for \texttt{Parser} is that operations to examine parsers have to be constantly unwrapped and rewrapped with the constructor \texttt{Parser}, and this adds to the running time of parser operations. In addition there is an unwanted element of \texttt{Parser}, namely \texttt{Parser undefined}. In other words, \texttt{Parser a} and \texttt{String \to [(a,String)]} are not isomorphic types. Recognising this, Haskell allows a \texttt{newtype} declaration for types defined with a single constructor taking a single argument. It differs from a type synonym in that it creates a genuinely new type whose values must be expressed using the \texttt{Parser} wrapper. But these coercions, though they have to appear in the program text, do not add to the execution time of the program because the Haskell compiler eliminates them before evaluation begins. The values of the new type are systematically replaced by the values in the underlying type. Consequently, \texttt{Parser a} and \texttt{String \to [(a,String)]} describe isomorphic types, and \texttt{Parser undefined} and \texttt{undefined} are isomorphic values sharing the same representation. New types, as distinct from synonym types, can be made members of type classes in different ways from the underlying type.

With either kind of declaration we have to provide some way of applying the parsing function, so we define

\begin{verbatim}
apply :: Parser a -> String -> [(a,String)]
apply (Parser p) s = p s
\end{verbatim}
The functions `apply` and `Parser` are mutual inverses and witness the isomorphism.

We also define

```haskell
parse :: Parser a -> String -> a
parse p = fst . head . apply p
```

The function `parse p` returns the first object of the first parse, causing an error if the parser `p` fails. This is the only place an error might occur.

Now we can define

```haskell
instance Monad Parser where
  return x = Parser \s -> [(x,s)]
  p >>= q = Parser \s -> [(y,s'') |
    (x,s') <- apply p s,
    (y,s'') <- apply (q x) s']
```

In the definition of `p >>= q` the parser `p` is applied to an input string, producing a list of possible parses each of which is paired with the corresponding unconsumed portion of the input. The parser `q` is then applied to each parse to produce a list of results whose concatenation provides the final answer. One should also show that the three monad laws hold, a task we will leave as an exercise.

---

11.2 Basic parsers

Perhaps the simplest basic parser is

```haskell
getc :: Parser Char
cetc = Parser f
  where f [] = []
        f (c:cs) = [(c,cs)]
```

This parser returns the first character of the input if there is one. It plays exactly the same role for parsers as `getChar` does for the input–output monad of the previous chapter.

Next, here is a parser for recognising a character that satisfies a given condition:

```haskell
sat :: (Char -> Bool) -> Parser Char
sat p = do {c <- getc;
           if p c then return c
           else fail}
```
where \texttt{fail} is defined by

\[
\texttt{fail} = \texttt{Parser} (\lambda s \to [])
\]

The parser \texttt{fail} is another basic parser that returns no parses. The parser \texttt{sat} \ p reads a character and, if it satisfies \ p, returns the character as the result. The definition of \texttt{sat} can be written more briefly by using a little combinator called \texttt{guard}:

\[
\texttt{sat} \ p = \texttt{do} \{ c \leftarrow \texttt{getc}; \ \texttt{guard} \ (p \ c); \ \texttt{return} \ c \} \\
\]

\[
\texttt{guard} :: \texttt{Parser} () \\
\texttt{guard} \ True = \texttt{return} () \\
\texttt{guard} \ False = \texttt{fail}
\]

To see that these two definitions are the same, observe that if \texttt{p c} is false, then

\[
\texttt{guard} \ (p \ c) >> \texttt{return} \ c = \texttt{fail} >> \texttt{return} \ c = \texttt{fail}
\]

Note the use of the law \texttt{fail} >> \texttt{p} = \texttt{fail}, whose proof we leave as an exercise. If \texttt{p c} is true, then

\[
\texttt{guard} \ (p \ c) >> \texttt{return} \ c = \texttt{return} () >> \texttt{return} \ c = \texttt{return} c
\]

Using \texttt{sat} we can define a number of other parsers; for instance

\[
\texttt{char} :: \texttt{Char} \to \texttt{Parser} () \\
\texttt{char} \ x = \texttt{do} \{ c \leftarrow \texttt{sat} (==x); \ \texttt{return} () \}
\]

\[
\texttt{string} :: \texttt{String} \to \texttt{Parser} () \\
\texttt{string} [] = \texttt{return} () \\
\texttt{string} (x:xs) = \texttt{do} \{ \texttt{char} \ x; \ \texttt{string} \ xs; \ \texttt{return} () \}
\]

\[
\texttt{lower} :: \texttt{Parser} \texttt{Char} \\
\texttt{lower} = \texttt{sat} \texttt{isLower}
\]

\[
\texttt{digit} :: \texttt{Parser} \texttt{Int} \\
\texttt{digit} = \texttt{do} \{ d \leftarrow \texttt{sat} \texttt{isDigit}; \ \texttt{return} \ (\texttt{cvt} \ d) \} \\
\texttt{where} \ \texttt{cvt} \ d = \texttt{fromEnum} \ d - \texttt{fromEnum} \ '0'
\]

The parser \texttt{char} \ x looks for the specific character \texttt{x} as the next item in the input string, while \texttt{string \ xs} looks for a specific string; both parsers return () if successful. For example,
The parser `digit` looks for a digit character and returns the corresponding integer if successful. The parser `lower` looks for a lowercase letter, returning such a letter if found.

11.3 Choice and repetition

In order to define more sophisticated parsers we need operations for choosing between alternative parsers and for repeating parsers. One such alternation operator is `( <| > )`, defined by

\[
(\text{p} \mid \text{q}) :\!\!: \text{Parser a} \to \text{Parser a} \to \text{Parser a}
\]

\[
p \mid \mid q = \text{Parser } f
\]

\[
\text{where } f \ s = \text{let } ps = \text{apply } p \ s \text{ in }
\]

\[
\text{if null } ps \text{ then apply } q \ s
\]

\[
\text{else } ps
\]

Thus `p <|> q` returns the same parses as `p` unless `p` fails, in which case the parses of `q` are returned. If both `p` and `q` are deterministic, then so is `p <|> q`. For another choice of `|>` see the exercises. We claim that `|>` is associative with `fail` as its identity element, but again we relegate the proof as an exercise.

Here is a parser for recognising a string of lowercase letters:

\[
\text{lowers} :\!\!: \text{Parser String}
\]

\[
\text{lowers } = \text{do } \{ c \leftarrow \text{lower} ; \ cs \leftarrow \text{lowers} ; \ \text{return } (c:cs) \}
\]

\[
\mid \mid \text{return } ""
\]

To see how this parser works, suppose the input is the string ‘Upper’. In this case the parser on the left of `|>` fails because ‘U’ is not a lowercase letter. However, the parser on the right succeeds, so

\[
\text{ghci} > \ \text{apply lowers } "\text{Upper}"
\]

\[
["\text{"},"\text{Upper}\text{"}]
\]

With input string ‘isUpper’, the left-hand parser succeeds, so

\[
\text{ghci} > \ \text{apply lowers } "\text{isUpper}"
\]

\[
["\text{is"},"\text{Upper"}]
\]
Use of the choice operator `|>` requires care. For example, consider a very simple form of arithmetic expression that consists of either a single digit or a digit followed by a plus sign followed by another digit. Here is a possible parser:

```haskell
wrong :: Parser Int
wrong = digit <|> addition

addition :: Parser Int
addition = do {m <- digit; char '+'; n <- digit;
  return (m+n)}
```

We have

```
ghci> apply wrong "1+2"
[(1,"+2")]
```

The parser `digit` succeeds, so `addition` is not invoked. But what we really wanted was to return `[(3,"")]`, absorbing as much of the input as possible. One way to correct `wrong` is to rewrite it in the form

```haskell
better = addition <|> digit
```

Then on `1+2` the parser `addition` succeeds, returning the result we want. What is wrong with `better` is that it is inefficient: applied to the input `1` it parses the digit but fails to find a subsequent plus sign, so parser `addition` fails. As a result `digit` is invoked and the input is parsed again from scratch. Not really a problem with a single digit, but the repetition of effort could be costly if we were parsing for a numeral that could contain many digits.

The best solution is to factor the parser for digits out of the two component parsers:

```haskell
best = digit >>= rest
rest m = do {char '+'; n <- digit; return (m+n)} <|> return m
```

The argument to `rest` is just an accumulating parameter. We saw essentially the same solution in the chapter on pretty-printing. Factoring parsers to bring out common prefixes is a Good Idea to improve efficiency.

Generalising from the definition of `lowers`, we can define a parser combinator that repeats a parser zero or more times:

```haskell
many :: Parser a -> Parser [a]
many p = do {x <- p; xs <- many p; return (x:xs)} <|> none
```
The value `none` is different from `fail` (why?). We can now define

```haskell
lowers = many lower
```

In many applications, so-called *white space* (sequences of space, newline and tab characters) can appear between *tokens* (identifiers, numbers, opening and closing parentheses, and so on) just to make the text easier to read. The parser `space` recognises white space:

```haskell
space :: Parser ()
space = many (sat isSpace) >>= return ()
```

The function `isSpace` is defined in the library `Data.Char`. The function

```haskell
symbol :: String -> Parser ()
symbol xs = space >>= string xs
```

ignores white space before recognising a given string. More generally we can define

```haskell
token :: Parser a -> Parser a
token p = space >>= p
```

for ignoring white space before invoking a parser. Note that

```haskell
token p <|> token q = token (p <|> q)
```

but the right-hand parser is more efficient as it does not look for white space twice if the first parser fails.

Sometimes we want to repeat a parser one or more times rather than zero or more times. This can be done by a combinator which we will call `some` (it is also called `many1` in some parser libraries):

```haskell
some :: Parser a -> Parser [a]
some p = do {x <- p; xs <- many p; return (x:xs)}
```

This definition repeats that of the first parser in the definition of `many`, a fact we can take into account by redefining `many` in terms of `some`:

```haskell
many :: Parser a -> Parser [a]
many p = optional (some p)
```

```haskell
optional :: Parser [a] -> Parser [a]
optional p = p <|> none
```
The parsers many and some are now mutually recursive.

Here is a parser for natural numbers, one that allows white space before the number:

```haskell
natural :: Parser Int
natural = token nat
nat = do {ds <- some digit;
          return (foldl1 shiftl ds)}
  where shiftl m n = 10*m+n
```

The subsidiary parser `nat` does not allow white space before the number.

Consider now how to define a parser for an integer numeral, which by definition is a nonempty string of digits possibly prefixed by a minus sign. You might think that the parser

```haskell
int :: Parser Int
int = do {symbol "-"; n <- natural; return (-n)}
       <|> natural
```

does the job, but it is inefficient (see Exercise H) and may or may not be what we want. For example,

```haskell
ghci> apply int " -34"
[(-34,"")]
ghci> apply int " - 34"
[(-34,"")]
```

Whereas we are quite happy with white space before a numeral, we may not want any white space to appear between the minus sign and the ensuing digits. If that is the case, then the above parser will not do. It is easy to modify the given definition of `int` to give what we want:

```haskell
int :: Parser Int
int = do {symbol "-"; n <- nat; return (-n)}
       <|> natural
```

This parser is still inefficient, and a better alternative is to define

```haskell
int :: Parser Int
int = do {space; f <- minus; n <- nat; return (f n)}
  where
    minus = (char '-' >> return negate) <|> return id
```

The parser `minus` returns a function, either `negate` if the first symbol is a minus sign, or the identity function otherwise.
Next, let us parse a list of integers, separated by commas and enclosed in square brackets. White space is allowed before and after commas and brackets though not of course between the digits of the integers. Here is a very short definition:

\[
\text{ints} :: \text{Parser \[Int\]}
\text{ints} = \text{bracket \{(manywith (symbol ",") \text{int})\}}
\]

The subsidiary parser \text{bracket} deals with the brackets:

\[
\text{bracket} :: \text{Parser a -> Parser a}
\text{bracket} p = \text{do \{symbol \"[\";}
\text{x <- p;}
\text{symbol \"]\";}
\text{return x}\}
\]

The function \text{manywith \ sep \ p} acts a bit like \text{many \ p} but differs in that the instances of \text{p} are separated by instances of \text{sep} whose results are ignored. The definition is

\[
\text{manywith :: Parser b -> Parser a -> Parser \[a\]}
\text{manywith \ q \ p = \text{optional \{(somewith \ q \ p)\}}}
\]

\[
\text{somewith :: Parser b -> Parser a -> Parser \[a\]}
\text{somewith \ q \ p = \text{do \{x <- p;}
\text{xs <- many (q >> p);}
\text{return (x:xs)\}}}
\]

For example,

\[
\text{ghci}\text{> apply ints \"[2, -3, 4]" [([2,-3,4],")\]}
\text{ghci}\text{> apply ints \"[2, -3, +4]\" []}
\text{ghci}\text{> apply ints \"[]\" [([],")\]}
\]

Integers cannot be preceded by a plus sign, so parsing the second expression fails.

11.4 Grammars and expressions

The combinators described so far are sufficiently powerful for translating a structural description of what is required directly into a functional parser. Such a struc-
natural description is provided by a grammar. We will illustrate some typical grammars by looking at parsers for various kinds of arithmetic expression.

Let us start by building a parser for the type Expr, defined by

```haskell
data Expr = Con Int | Bin Op Expr Expr
data Op = Plus | Minus
```

Here is a grammar for fully parenthesised expressions, expressed in what is known as Backus-Naur form, or BNF for short:

```
expr ::= nat | '(' expr op expr ')'
op ::= '+' | '-'
nat ::= {digit}+
digit ::= '0' | '1' | ... | '9'
```

This grammar defines four syntactic categories. Symbols enclosed in quotes are called terminal symbols and describe themselves; these are symbols that actually occur in the text. There are ten possible characters for a digit, and a nat is defined as a sequence of one or more digits. The meta-symbol {-}+ describes a non-zero repetition of a syntactic category. Note that we do not allow an optional minus sign before a sequence of digits, so constants are natural numbers, not arbitrary integers. The grammar states that an expression is either a natural number or else a compound expression consisting of an opening parenthesis, followed by an expression, followed by either a plus or minus sign, followed by another expression, and finally followed by a closing parenthesis. It is implicitly understood in the description that white space is ignored between terminal symbols except between the digits of a number. The grammar translates directly into a parser for expressions:

```
expr ::= Parser Expr
expr = token (constant <|> paren binary)
constant = do {n <- nat; return (Con n)}
binary = do {e1 <- expr;
          p <- op;
          e2 <- expr;
          return (Bin p e1 e2)}
op = (symbol "+" >> return Plus) <|>
     (symbol "-" >> return Minus)
```

For readability we have made use of a subsidiary parser binary; the parser paren is left as an exercise.

Now suppose we want a parser that also works for expressions that are not fully parenthesised, things like 6-2-3 and 6-(2-3) and (6-2)-3. In such a case, (+)
and \((-\) should associate to the left in expressions, as is normal with arithmetic. One way to express such a grammar in BNF is to write

\[
\begin{align*}
\text{expr} &::= \text{expr} \text{ op} \text{ term} | \text{ term} \\
\text{term} &::= \text{ nat} | '(' \text{ expr } ')' \\
\end{align*}
\]

This grammar says that an expression is a sequence of one or more terms separated by operators. A term is either a number or a parenthesised expression. In particular, 6-2-3 will be parsed as the expression 6-2 followed by a minus operator, followed by the term 3. In other words, the same as \((6-2)-3\), as required. This grammar also translates directly into a parser:

\[
\begin{align*}
\text{expr} & = \text{token } (\text{binary } <|> \text{ term}) \\
\text{binary} & = \text{do } \{ \text{e1 } <- \text{ expr}; \\
 & \text{ p } <- \text{ op}; \\
 & \text{ e2 } <- \text{ term}; \\
 & \text{ return } (\text{Bin p e1 e2}) \} \\
\text{term} & = \text{token } (\text{constant } <|> \text{ paren } \text{ expr}) \\
\end{align*}
\]

However, there is a fatal flaw with this parser: it falls into an infinite loop. After ignoring initial white space the first action of \text{expr} is to invoke the parser \text{binary}, whose first action is to invoke the parser \text{expr} again. Whoops!

Furthermore, it will not do to rewrite \text{expr} as

\[
\text{expr} = \text{token } (\text{term } <|> \text{ binary})
\]

because, for example,

Main*> apply \text{expr} "3+4"
\[(\text{Con 3,"+4"})\]

Only the first term is parsed. The problem is called the \textit{left recursion} problem and is a difficulty with all recursive parsers, functional or otherwise.

One solution is to rewrite the grammar in the following equivalent form:

\[
\begin{align*}
\text{expr} ::= \text{term} \{\text{op} \text{ term}\}*
\end{align*}
\]

The meta-symbol \{-\}* indicates a syntactic category that can be repeated zero or more times. The new parser then takes the form

\[
\begin{align*}
\text{expr} & = \text{token } (\text{term } >>= \text{ rest}) \\
\text{rest e1 } = \text{ do } \{ \text{p } <- \text{ op}; \\
 & \text{ e2 } <- \text{ term}; \\
 & \text{ rest } (\text{Bin p e1 e2}) \} <|> \text{ return e1}
\end{align*}
\]
The parser rest corresponds to the category \{op term\}* and takes an argument (an accumulating parameter) whose value is the expression parsed so far.

Finally, let us design a parser for arithmetic expressions that may contain multiplication and division, changing the definition of Op to

\[
\text{data } \text{Op} = \text{Plus} \mid \text{Minus} \mid \text{Mul} \mid \text{Div}
\]

The usual rules apply in that multiplication and division take precedence over addition and subtraction, and operations of the same precedence associate to the left. Here is a grammar:

\[
\begin{align*}
\text{expr} & ::= \text{term} \{\text{addop} \text{ term}\} \\
\text{term} & ::= \text{factor} \{\text{mulop} \text{ factor}\} \\
\text{factor} & ::= \text{nat} \mid (\text{ expr })' \\
\text{addop} & ::= '+' \mid '-' \\
\text{mulop} & ::= '*' \mid '/'
\end{align*}
\]

And here is the parser:

\[
\begin{align*}
\text{expr} &= \text{token } (\text{term} >>= \text{rest}) \\
\text{rest } e1 &= \text{ do } \{p <- \text{addop}; \\
& \quad e2 <- \text{term}; \\
& \quad \text{rest } (\text{Bin } p \ e1 \ e2)\} \\
& \langle |\rangle \text{ return } e1 \\
\text{term} &= \text{token } (\text{factor} >>= \text{more}) \\
\text{more } e1 &= \text{ do } \{p <- \text{mulop}; \\
& \quad e2 <- \text{factor}; \\
& \quad \text{more } (\text{Bin } p \ e1 \ e2)\} \\
& \langle |\rangle \text{ return } e1 \\
\text{factor} &= \text{token } (\text{constant } <|\rangle \text{ paren } \text{expr})
\end{align*}
\]

The definitions of addop and mulop are left as exercises.

11.5 Showing expressions

Our final question is: how can we install \text{Expr} as a member of the type class \text{Show} so that the function show is the inverse of parsing? More precisely, we want to define show so that

\[
\text{parse } \text{expr } (\text{show } e) = e
\]

Recall that parse p extracts the first parse returned by apply p.
As a warm-up, here is the instance of Show when expr is the parser for fully parenthesised expressions involving addition and subtraction only:

```haskell
instance Show Expr where
    show (Con n) = show n
    show (Bin op e1 e2) =
        "(" ++ show e1 ++
        " " ++ showop op ++
        " " ++ show e2 ++ ")"

    showop Plus = "+"
    showop Minus = "-"
```

Clear enough, but there is a problem with efficiency. Because (++) has time complexity linear in the length of its left argument, the cost of evaluating show is, in the worst case, quadratic in the size of the expression.

The solution, yet again, is to use an accumulating parameter. Haskell provides a type synonym ShowS:

```haskell
type ShowS = String -> String
```

and also the following subsidiary functions

```haskell
showChar :: Char -> ShowS
showString :: String -> ShowS
showParen :: Bool -> ShowS -> ShowS
```

These functions are defined by

```haskell
    showChar = (:)
    showString = (++)
    showParen p x = if b then
                    showChar '(' . p . showChar ')'
                    else p
```

Now we can define show for expressions by

```haskell
    show e = shows e ""
where
    shows (Con n) = showString (show n)
    shows (Bin op e1 e2) = showParen True (shows e1 . showSpace .
                            showsop op . showSpace . shows e2)
    showsop Plus = showChar '+'
    showsop Minus = showChar '-'
```
showSpace     = showChar ' '

This version, which contains no explicit concatenation operations, takes linear time in the size of the expression.

Now suppose we want to display expressions that are not fully parenthesised. There is no need for parentheses around left-hand expressions, but we do need parentheses around right-hand expressions. That leads to

\[
\text{show} = \text{shows False e ""}
\]

where
\[
\begin{align*}
\text{shows b (Con n)} &= \text{showString (show n)} \\
\text{shows b (Bin op e1 e2)} &= \text{showParen p (showsFalse e1 . showSpace . showsop op . showSpace . showsTrue e2)}
\end{align*}
\]

This definition takes no account of associativity; for example, \(1+(2+3)\) is not shown as \(1+2+3\).

Finally, let’s tackle expressions involving all four arithmetic operations. The difference here is that:

1. With expressions \(e1 + e2\) or \(e1 - e2\) we will never need parentheses around \(e1\) (just as above), nor will we need parentheses around \(e2\) if \(e2\) is a compound expression with a multiplication or division at the root.

2. On the other hand, with expressions \(e1 \times e2\) or \(e1 / e2\) we will need parentheses around \(e1\) if \(e1\) is a compound expression with a plus or minus at the root, and we will always need parentheses around \(e2\).

One way to codify these rules is to introduce precedence levels (for another way, see Exercise L). Define

\[
\begin{align*}
\text{prec :: Op -> Int} \\
\text{prec Mul} &= 2 \\
\text{prec Div} &= 2 \\
\text{prec Plus} &= 1 \\
\text{prec Minus} &= 1
\end{align*}
\]

Consider now how to define a function \text{showsPrec} with type

\[
\text{showsPrec :: Int -> Expr -> ShowS}
\]

such that \text{showsPrec p e} shows the expression \(e\) assuming that the parent of \(e\) is a compound expression with an operator of precedence \(p\). We will define \text{show} by
show e = showsPrec 0 e ""
so the enclosing context of e is an operator with fictitious precedence 0. We can at once define

showsPrec p (Con n) = showString (show n)
because constants are never enclosed in parentheses. The interesting case is when we have a compound expression. We give the definition first and explain it afterwards:

showsPrec p (Bin op e1 e2)
  = showParen (p>q) (showsPrec q e1 . showSpace .
               showsop op . showSpace . showsPrec (q+1) e2)
  where q = prec op

We put parentheses around an expression if the parent operator has greater precedence than the current one. To display the expression e1 it is therefore sufficient to pass the current precedence as the new parent precedence. But we need parentheses around e2 if the root operator of e2 has precedence less than or equal to q; so we have to increment q in the second call.

Admittedly, the above definition of showsPrec requires a little thought, but there is a payoff. The type class Show has a second method in it, namely showsPrec. Moreover, the default definition of show is just the one above. So to install expressions as a member of Show we merely have to give the definition of showsPrec.

11.6 Exercises

Exercise A

Consider the synonym

    type Angle = Float

Suppose we want to define equality on angles to be equality modulo a multiple of $2\pi$. Why can’t we use (==) for this test? Now consider

    newtype Angle = Angle Float

Install Angle as a member of Eq, thereby allowing (==) as an equality test between angles.
Exercise B

We could have defined

```haskell
newtype Parser a = Parser (String -> Maybe (a, String))
```

Give the monad instance of this kind of parser.

Exercise C

Prove that \( \text{fail} >> p = \text{fail} \).

Exercise D

Could we have defined \(<|>\) in the following way?

```haskell
p <|> q = Parser (\s -> parse p s ++ parse q s)
```

When is the result a deterministic parser? Define a function

```haskell
limit :: Parser a -> Parser a
```

such that \( \text{limit} (p <|> q) \) is a deterministic parser, even if \( p \) and \( q \) are not.

Exercise E

Parsers are not only instances of monads, they can also be made instances of a more restricted class, called \( \textbf{MonadPlus} \), a class we could have introduced in the previous chapter. Basically, these are monads that support choice and failure. The Haskell definition is

```haskell
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

As examples, both [] and Maybe can be made members of MonadPlus:

```haskell
instance MonadPlus [] where
  mzero = []
  mplus = (++)

instance MonadPlus Maybe where
  mzero = Nothing
  Nothing `mplus` y = y
  Just x `mplus` y = Just x
```

Install Parser as an instance of MonadPlus.
Exercise F

Continuing from the previous exercise, the new methods `mzero` and `mplus` are expected to satisfy some equational laws, as is usually the case with the methods of a type class. But currently the precise set of rules that these methods should obey is not agreed on by the Haskell designers! Uncontroversial are the laws that `mplus` should be associative with identity element `mzero`. That’s three equations. Another reasonable law is the left-zero law

\[
mzero >>= f = mzero
\]

The corresponding right-zero law, namely

\[
p >>= mzero = mzero
\]

can also be imposed. Does the `MonadPlus` instance of the list monad satisfy these five laws? How about the `Maybe` monad?

Finally, the really contentious law is the following one:

\[
(p \{mplus\} q >>= f = (p >>= f) \{mplus\} (q >>= f)
\]

This law is call the left-distribution law. Why can’t `Maybe` be installed as a member of `MonadPlus` if the left-distribution is imposed?

Exercise G

Design a parser for recognising Haskell floating-point numbers. Bear in mind that .314 is not a legitimate number (no digits before the decimal point) and that 3 . 14 is not legitimate either (because no spaces are allowed before or after the decimal point).

Exercise H

Why are the first and second definitions of `int` given in the text inefficient, compared to the third definition?

Exercise I

Is "(3)" a fully parenthesised expression? Is it a non-fully parenthesised expression? Haskell allows parenthesised constants:

```
ghci> (3)+4
7
```

Design a parser for fully parenthesised expressions that allows parentheses around constants.
Exercise J

Consider the grammar \( expr ::= term \{ op \ term \}* \). Define pair and shunt so that the following parser is legitimate:

\[
expr = do \{ e1 \leftarrow \text{term};
        pes \leftarrow \text{many (pair op term)};
        \text{return (foldl shunt e1 pes)}\}\
\]

Exercise K

Define the parsers addop and mulop.

Exercise L

Consider again the showing of expressions with all four arithmetic operations. The rules for putting in parentheses come down to: we need parentheses around \( e_1 \) in \( e_1 \ op \ e_2 \) if \( op \) is a multiplication operator, and the root of \( e_1 \) isn’t. Dually we will need parentheses around \( e_2 \) if either \( op \) is a multiplication operator or the root of \( e_2 \) isn’t. Defining

\[
\text{isMulOp Mul} = \text{True} \\
\text{isMulOp Div} = \text{True} \\
\text{isMulOp _} = \text{False}\
\]

construct an alternative definition of show involving a subsidiary function

\[
\text{showsF :: (Op -> Bool) -> Expr -> ShowS}\
\]

11.7 Answers

Answer to Exercise A

Because (==) is the equality test on floating-point numbers, and different numbers cannot be equal.

instance Eq Angle where

\[
\text{Angle x == Angle y} = \text{reduce x == reduce y}\
\]

where

\[
\text{reduce x} | x < 0 = \text{reduce (x + r)} \\
| x > r = \text{reduce (x - r)} \\
| \text{otherwise} = x \\
\text{where r = 2*pi}\
\]
11.7 Answers

Answer to Exercise B

instance Monad Parser where
  return x = Parser ($s -> Just (x,s))
  P >>= q = Parser ($s -> case apply p s of
     Nothing -> apply q s
     Just (x,s') -> Just (x,s'))

Answer to Exercise C

fail >>= p
= fail >>= const p
= fail

The fact that fail >>= p = fail is immediate from the definition of fail and
the definition of p >>= q.

Answer to Exercise D

Yes, but the result is only a deterministic parser when either p or q is fail. The
function limit can be defined by

    limit p = Parser (take 1 . apply p)

Answer to Exercise E

mzero = fail
mplus = (<!|>)

Answer to Exercise F

Yes, both the list monad and the Maybe monad satisfy the five laws. For example,
in the list monad

    mzero >>= f = concat (map f []) = [] = mzero
    xs >>= mzero = concat (map (const []) xs) = [] = mzero

With Maybe the left-distribution law doesn’t hold. We have

    (Just x `mplus` q) >>= (\x -> Nothing)
= Just x >>= (\x -> Nothing)
= Nothing

but
(Just x >> \x -> Nothing) `mplus`
(q >>= \x -> Nothing)
= Nothing `mplus` (q >>= \x -> Nothing)
= q >>= \x -> Nothing

The two resulting expressions are not equal (take q = undefined).

**Answer to Exercise G**

```haskell
float :: Parser Float
float = do {ds <- some digit;
             char '.';
             fs <- some digit;
             return (foldl shiftl 0 ds +
                     foldr shiftr 0 fs)}
  where shiftl n d = 10*n + fromIntegral d
        shiftr f x = (fromIntegral f+x)/10
```

The parser `digit` returns an `Int`, which has to be converted to a number (in this case a `Float`).

**Answer to Exercise H**

White space is parsed twice. For example, calling the first version `int1` and the third `int3` we have

ghci> apply int3 $ replicate 100000 ' ' ++ "3"
[(3,"")]
(1.40 secs, 216871916 bytes)
ghci> apply int1 $ replicate 100000 ' ' ++ "3"
[(3,"")]
(2.68 secs, 427751932 bytes)

**Answer to Exercise I**

No, according to the first grammar for `expr`, only binary expressions can be parenthesised. Yes, according to the second grammar as arbitrary expressions can be parenthesised.

The revised grammar is

```
expr ::= term | '( expr op expr ')
term ::= nat | '( expr ')
```

The corresponding parser is
expr = token (term <|> paren binary)
where
term = token (constant <|> paren expr)
binary = do {e1 <- expr;
    p <- op;
    e2 <- expr;
    return (Bin p e1 e2)}

Answer to Exercise J

pair :: Parser a -> Parser b -> Parser (a,b)
pair p q = do {x <- p; y <- q; return (x,y)}

shunt e1 (p,e2) = Bin p e1 e2

Answer to Exercise K

addop = (symbol "+" >> return Plus) <|>
      (symbol "-" >> return Minus)
mulop = (symbol "*" >> return Mul) <|>
      (symbol "/" >> return Div)

Answer to Exercise L

show e = showsF (const False) e ""
where
showsF f (Con n) = showString (show n)
showsF f (Bin op e1 e2)
  = showParen (f op) (showsF f1 e1 . showSpace .
               showsop op . showSpace . showsF f2 e2)
  where f1 x = isMulOp op && not (isMulOp x)
       f2 x = isMulOp op || not (isMulOp x)

The design of functional parsers in a monadic setting has long been a favourite
application of functional programming. Our presentation follows that of ‘Monadic
parsing in Haskell’ by Graham Hutton and Erik Meijer, which appears in The Jour-
Chapter 12

A simple equational calculator

This final chapter is devoted to a single programming project, the design and implementation of a simple calculator for carrying out point-free equational proofs. Although the calculator provides only a small subset of the facilities one might want in an automatic proof assistant, and is highly restrictive in a number of other ways, it will nevertheless be powerful enough to prove many of the point-free laws described in previous chapters – well, provided we are prepared to give it a nudge in the right direction if necessary. The project is also a case study in the use of modules. Each component of the calculator, its associated types and functions, is defined in an appropriate module and linked to other modules through explicit import and export lists.

12.1 Basic considerations

The basic idea is to construct a single function \texttt{calculate} with type

\[
\texttt{calculate :: [Law] \rightarrow Expr \rightarrow Calculation}
\]

The first argument of \texttt{calculate} is a list of laws that may be applied. Each law consists of a descriptive name and an equation. The second argument is an expression and the result is a calculation. A calculation consists of a starting expression and a sequence of steps. Each step consists of the name of a law and the expression that results by applying the left-hand side of the law to the current expression. The calculation ends when no more laws can be applied, and the final expression is the conclusion. The entire process is automatic, requiring no intervention on the part of the user.
12.1 Basic considerations

Laws, expressions and calculations are each elements of appropriate data types to be defined in the following sections. But for now let us plunge straight in with an example to show the framework we have in mind.

Here are some laws (we use a smaller font to avoid breaking lines):

definition filter: filter p = concat . map (box p)
definition box: box p = if p one nil
if after dot: if p f g . h = if (p . h) (f . h) (g . h)
dot after if: h . if p f g = if p (h . f) (h . g)
nil constant: nil . f = nil
map after nil: map f . nil = nil
map after one: map f . one = one . f
map after concat: map f . concat = concat . map (map f)

map functor: map f . map g = map (f . g)
map functor: map id = id

Each law consists of a name and an equation. The name of the law is terminated by a colon sign, and an equation consists of two expressions separated by an equals sign. Each expression describes a function; our calculator will be one that simplifies functional expressions only (yes, it’s a pointless calculator). Expressions are built from constants, like one and map, and variables, like f and g. The precise syntax will be given in due course. Note that there are no conditional laws, equations that are valid only if some subsidiary conditions are met. That will limit what we can do with the calculator, but it still leaves enough to be interesting.

Suppose we want to simplify the expression filter p . map f. Here is one possible calculation:

filter p . map f
= {definition filter} concat . map (box p) . map f
= {map functor} concat . map (box p . f)
= {definition box} concat . map (if p one nil . f)
= {if after dot} concat . map (if (p . f) (one . f) (nil . f))
= {nil constant} concat . map (if (p . f) (one . f) nil)

The steps of the calculation are displayed in the conventional format with the name of the law being invoked printed in braces between the two expressions to which
it applies. No more laws apply to the final expression, so that is the result of the calculation. It is certainly not simpler than the expression we started out with.

The calculator could have applied some of the laws in a different order; for example, the definition of box could have been applied at the second step rather than at the third. But the conclusion would have been the same. It is also possible, though not with this particular set of laws, that an expression could be simplified to different conclusions by different calculations. However, at the outset we make the decision that calculate returns just one calculation, not a tree of possible calculations.

Notice what is happening at each step. Some left-hand side of some law is matched against some subexpression of the current expression. If a match is successful the result is a substitution for the variables occurring in the law. For example, in the second step, the subexpression map (box p) . map f is successfully matched with the first map functor law, resulting in a substitution in which the variable f of the functor law is bound to the expression box p, and the variable g is bound to f. The result of the step involves rewriting the subexpression with the corresponding instance of the right-hand side of the law in which each variable is replaced by its binding expression. Matching, substitutions and rewriting are all fundamental components of the calculator.

Now suppose that with the same set of laws as above we want to simplify the expression map f . filter (p . f). Here is the calculation:

```
map f . filter (p . f)
= {definition filter}
  map f . concat . map (box (p . f))
= {map after concat}
  concat . map (map f) . map (box (p . f))
= {map functor}
  concat . map (map f . box (p . f))
= {definition box}
  concat . map (map f . if (p . f) one nil)
= {dot after if}
  concat . map (if (p . f) (map f . one) (map f . nil))
= {map after nil}
  concat . map (if (p . f) (map f . one) nil)
= {map after one}
  concat . map (if (p . f) (one . f) nil)
```

Again, some of the laws could have been applied in a different order. No more laws apply to the final expression so that is the result of the calculation.

The point about these two calculations is that the two final expressions are the same, so we have proved
12.1 Basic considerations

\[ \text{filter } p \cdot \text{map } f = \text{map } f \cdot \text{filter } (p \cdot f) \]

This is the way we will conduct equational proofs, simplifying both sides to the same conclusion. Rather than show two calculations, one after the other, the two results can be pasted together by recording the first calculation and then appending the steps of the second calculation in reverse. The main advantage of this scheme is simplicity; we do not have to invent a new format for proofs, and we do not have to apply laws from right to left in order to reach the desired goal. Accordingly, we will also define a function

\[
\text{prove} :: \text{[Law]} \rightarrow \text{Equation} \rightarrow \text{Calculation}
\]

for proving equations.

Further considerations

It is a basic constraint of our calculator that laws are applied in one direction only, namely from left to right. This is primarily to prevent calculations from looping. If laws could be applied in both directions, then the calculator could oscillate by applying a law in one direction and then immediately applying it in the reverse direction.

Even with a left-to-right rule, some laws can lead to infinite calculations. Typically, these laws are the definitions of recursive functions. For example, consider the definition of \texttt{iterate}:

\[
\text{defn iterate: iterate } f = \text{cons} \cdot \text{fork } \text{id} (\text{iterate } f \cdot f)
\]

This is the definition of \texttt{iterate} expressed in point-free form. The functions \texttt{cons} and \texttt{fork} are defined by

\[
\text{cons } (x, xs) = x:xs \\
\text{fork } f \cdot g \cdot x = (f \cdot x, g \cdot x)
\]

We have met \texttt{fork} before in the exercises in Chapters 4 and 6, except that we wrote \texttt{fork } (f, g) instead of \texttt{fork } f \cdot g. In what follows, all our functions will be curried. The appearance of the term \texttt{iterate } f on both sides of the law means that any calculation that can apply the definition of \texttt{iterate} once can, potentially, apply it infinitely often. But not necessarily. Here is a calculation (produced by the calculator) that avoids infinite regress:

\[
\text{head } \cdot \text{iterate } f \\
= \quad \{\text{defn iterate}\} \\
\text{head } \cdot \text{cons} \cdot \text{fork } \text{id} (\text{iterate } f \cdot f)
\]
A simple equational calculator

\[
\begin{align*}
&= \{\text{head after cons}\} \\
&\quad \text{fst} \cdot \text{fork} \text{id} \ (\text{iterate f . f}) \\
&= \{\text{fst after fork}\} \\
&\quad \text{id}
\end{align*}
\]

The calculation makes use of the two laws:

\[
\begin{align*}
\text{head after cons:} & \quad \text{head} \cdot \text{cons} = \text{fst} \\
\text{fst after fork:} & \quad \text{first} \cdot \text{fork} f g = f
\end{align*}
\]

The reason non-termination is avoided is that these two laws are given preference over definitions in calculations, a wrinkle that we will elaborate on below.

In order to appreciate just what the calculator can and cannot do, here is another example of rendering a recursive definition into point-free form. Consider the definition of concatenation:

\[
\begin{align*}
[] ++ ys &= ys \\
(x:xs) ++ ys &= x:(xs ++ ys)
\end{align*}
\]

We will use \text{cat} to stand for \((++). \) We will also need \text{nil}, \text{cons} and the function \text{cross} \((f,g), \) which we will now write as \(f * g. \) Thus,

\[
(f * g) (x,y) = (f x, g y)
\]

Finally we will need a combinator \text{assocr} (short for ‘associate-right’), defined by

\[
\text{assocr} ((x,y),z) = (x,(y,z))
\]

Here are the translations of the two defining equations of \text{cat} in point-free form:

\[
\begin{align*}
\text{cat} \cdot (\text{nil} * \text{id}) &= \text{snd} \\
\text{cat} \cdot (\text{cons} * \text{id}) &= \text{cons} \cdot (\text{id} * \text{cat}) \cdot \text{assocr}
\end{align*}
\]

We cannot prove that \text{cat} is associative with our calculator, for that would involve a proof by induction, but we can state it as a law:

\[
\text{cat associative: } \text{cat} \cdot (\text{cat} * \text{id}) = \text{cat} \cdot (\text{id} * \text{cat}) \cdot \text{assocr}
\]

Continuing with this example for a bit longer, here are the two bifunctor laws of \((*)\):

\[
\begin{align*}
\text{bifunctor *:} & \quad \text{id} * \text{id} = \text{id} \\
\text{bifunctor *:} & \quad (f * g) \cdot (h * k) = (f . h) * (g . k)
\end{align*}
\]

And here is a law about \text{assocr}:

\[
\text{assocr law: } \text{assocr} . ((f * g) * h) = (f * (g * h)) \cdot \text{assocr}
\]
Now for the point of the example: our calculator cannot perform the following valid calculation:

\[
\text{cat} \cdot (\text{cat} \cdot (f \ast g)) \ast h
\]

\[
= \{\text{identity law, in backwards direction}\}
\]

\[
\text{cat} \cdot ((\text{cat} \cdot (f \ast g)) \ast (\text{id} \cdot h))
\]

\[
= \{\text{bifunctor }, \text{in backwards direction}\}
\]

\[
\text{cat} \cdot (\text{cat} \cdot \text{id}) \cdot ((f \ast g) \ast h)
\]

\[
= \{\text{cat associative}\}
\]

\[
\text{cat} \cdot (\text{id} \ast \text{cat}) \cdot \text{assocr} \cdot ((f \ast g) \ast h)
\]

\[
= \{\text{assoc law}\}
\]

\[
\text{cat} \cdot (\text{id} \ast \text{cat}) \cdot (f \ast (g \ast h)) \cdot \text{assocr}
\]

\[
= \{\text{bifunctor }\}
\]

\[
\text{cat} \cdot ((\text{id} \cdot f) \ast (\text{cat} \cdot (g \ast h))) \cdot \text{assocr}
\]

\[
= \{\text{identity law}\}
\]

\[
\text{cat} \cdot (f \ast (\text{cat} \cdot (g \ast h))) \cdot \text{assocr}
\]

The problem here is that we have to apply the identity and bifunctor laws in both directions, and the calculator is simply not up to the task. Observe that the essence of the proof is the simplification of the expression

\[
\text{cat} \cdot (\text{id} \ast \text{cat}) \cdot \text{assocr} \cdot ((f \ast g) \ast h)
\]

in two different ways, one by using the associativity of \text{cat}, written in the form

\[
\text{cat associative: cat} \cdot (\text{id} \ast \text{cat}) \cdot \text{assocr} = \text{cat} \cdot (\text{cat} \ast \text{id})
\]

and one by using the \text{assocr} law. Even if we generalised calculate to return a tree of possible calculations, it would not be obvious what expression we would have to start out with in order to achieve the calculation above, so we abandon any attempt to get the calculator to produce it.

It is not just the functor laws that sometimes have to be applied in both directions. For an example, see Section 12.8. Sometimes we can get around the problem by stating a law in a more general form than necessary, sometimes by using a hack, and sometimes not at all. As we said at the outset, our calculator is a limited one.

In the scheme of automatic calculation that we are envisaging there are only two degrees of freedom: the choice of which law to apply, and the choice of which subexpression to be changed. The first degree of freedom can be embodied in the order in which laws are presented to the calculator: if two different laws are applicable, then the one earlier in the list is chosen.

Certainly some laws should be tried before others; these are laws that reduce the complexity of intermediate expressions. Good examples are the laws \(f \cdot \text{id} = f\) and \(\text{id} \cdot f = f\). The naive definition of complexity is that there are fewer compositions on the right than on the left. It is unlikely to be a mistake to apply these
laws as soon as the opportunity arises. Indeed the fact that \( \text{id} \) is the identity element of composition can and will be built into the calculator, so the two identity laws will be taken care of automatically. Similarly, early application of laws like \( \text{nil} \cdot f = \text{nil} \) and \( \text{map} \ f \cdot \text{nil} = \text{nil} \) (and indeed the two laws used in the calculation about \text{iterate}), all of which reduce the number of compositions, help to reduce the sizes of intermediate expressions. For the sake of a word, let us call these the \textit{simple} laws.

On the other hand, some laws should be applied only as a last resort. Typically, these laws are definitions, such as the definition of \text{filter} or \text{iterate}. For example, in the expression

\[
\text{map} \ f \cdot \text{concat} \cdot \text{map} (\text{filter} \ p)
\]

we really don’t want to apply the definition of \text{filter} too early; rather we would prefer to apply the \text{map} after \text{concat} law first, and only apply the definition of \text{filter} later on if and when it becomes necessary. Apart from anything else, intermediate expressions will be shorter.

In summary it looks sensible to sort our laws into the simple laws, followed the non-simple laws that are not definitions, followed by the definitions.

The second degree of freedom is represented by the order in which the subexpressions of a given expression are presented as candidates for instances of laws: if laws are applicable to two different subexpressions, then the subexpression coming earlier in the enumeration is chosen.

That still leaves open the decision whether to give preference to laws or to subexpressions in calculations. Do we start with a subexpression and try every law in turn, or start with a law and see if it applies anywhere? Does it really matter which of these alternatives is chosen? While it is true that, having applied some law at some subexpression, the next law to be applied is likely to be at a ‘nearby’ expression, it is not clear how to formalise this notion of nearness, nor is it clear whether it would contribute significantly to the efficiency of calculations, either in the computation time or in the length of the result.

### 12.2 Expressions

At the heart of the calculator is the data type \( \text{Expr} \) of expressions. Most of the components of the calculator are concerned with analysing and manipulating expressions in one way or the other. Expressions are built from (function) variables
12.2 Expressions

and constants, using functional composition as the basic combining form. Variables take no arguments, but constants can take any number of arguments, which are themselves expressions. We will suppose all functions are curried and there are no tuples; for example we write \( \text{pair } f \ g \) instead of \( \text{pair } (f, g) \). There is no particular reason for avoiding tuples, it is just that most functions we have discussed in the book are curried and we don’t really need both.

To compensate, we will also allow ourselves binary infix operators, writing, for example, \( f * g \) instead of \( \text{cross } f \ g \). Except for functional composition we will not assume any order of precedence or association between binary operators, insisting that expressions involving such operators be fully parenthesised. That still leaves open the question of the precedence of composition. Does \( f * g . h \) mean \( (f * g) . h \) or \( f *(g . h) \)? Haskell puts composition at a high level of precedence and we will adopt the same convention. Thus \( f * g . h \) will be parsed as \( f *(g . h) \). But we will always write such expressions using parentheses to avoid ambiguity.

Here is the proposed BNF grammar for expressions:

\[
\text{expr ::= simple \{op simple\}} \\
\text{simple ::= term \{'\.' term\}^} \\
\text{term ::= var | con \{arg\}^ | '(' expr ')'} \\
\text{arg ::= var | con | '(' expr ')'} \\
\text{var ::= letter \{digit\}} \\
\text{con ::= letter letter \{letter | digit\}^} \\
\text{op ::= \{symbol\}^}
\]

Variable names consist of single letters only, possibly followed by a single digit. Thus \( f \) and \( f1 \) are legitimate variable names. Constant names are sequences of at least two alphanumeric characters beginning with two letters, such as \text{map} or \text{lhs2tex}, while operator names are nonempty sequences of non-alphanumeric symbols, such as \( * \) and \( <+> \). The first line says that an expression is a simple expression, possibly followed by an operator and another simple expression. Simple expressions are compositions of terms. The remaining lines are, we trust, self-explanatory.

Here is the definition of \text{Expr} we will use:

\[
\text{newtype Expr = Compose [Atom] deriving Eq} \\
\text{data Atom = Var VarName | Con ConName [Expr] deriving Eq} \\
\text{type VarName = String} \\
\text{type ConName = String}
\]
Expressions and atoms are declared to be members of the class Eq because we will need to test expressions for equality. Later on we will install expressions as an instance of Show for printing them at the terminal.

Here are some examples of expressions and their representations:

- f . g . h => Compose [Var "f", Var "g", Var "h"]
- id => Compose []
- fst => Compose [Con "fst" []]
- fst . f => Compose [Con "fst" [], Var "f"]
- (f * g) . h => Compose [Con "*" [Var "f", Var "g"], Var "h"]
- f * g . h => Compose [Con "*" [Compose [Var "f"],
  Compose [Var "g", Var "h"]]]

The fact that composition is an associative operation is built into the design of Expr. The particular constant id is reserved and will always be interpreted as the identity element of composition.

The parsing combinators described in the previous chapter enable us to parse expressions. Following the BNF, we start with

```haskell
expr :: Parser Expr
expr = simple >>= rest
where
  rest s1 = do {op <- operator;
    s2 <- simple;
    return (Compose [Con op [s1,s2]])
  } <|> return s1
```

An operator is a sequence of one or more operator symbols, as long as it is neither the composition operator nor an equals sign:

```haskell
operator :: Parser String
operator = do {op <- token (some (sat symbolic));
    Parsing.guard (op /= "." && op /= "+=");
    return op}
symbolic = (\'elem\' opsymbols)
opsymbols = "!@#$%^&*+./<>?\^|:-~""
```

The function Parsing.guard is an example of a qualified name. The Haskell Prelude also provides a function guard, but we want the function of the same name from a module Parsing that includes all our parsing functions. A qualified name consists of a module name followed by a period followed by the name of the qualified value.
A simple expression is a sequence of one or more terms separated by composition:

```haskell
simple :: Parser Expr
simple = do {es <- somewith (symbol ".") term;
             return (Compose (concatMap deCompose es))}
```

The function `concatMap f` as an alternative to `concat . map f` is provided in the standard prelude, and `deCompose` is defined by

```haskell
deCompose :: Expr -> [Atom]
deCompose (Compose as) = as
```

Next, a term is an identifier, either a variable or a constant, possibly with arguments, or a parenthesised expression:

```haskell
term :: Parser Expr
term = ident args <|> paren expr
args = many (ident none <|> paren expr)
```

The parser `ident` takes a parser for a list of expressions and returns a parser for expressions:

```haskell
ident :: Parser [Expr] -> Parser Exp
ident args
    = do {x <- token (some (sat isAlphaNum));
           Parsing.guard (isAlpha (head x));
           if isVar x
              then return (Compose [Var x])
              else if (x == "id")
                        then return (Compose [])
                        else
                            do {as <- args;
                               return (Compose [Con x as])}
```

The test for being a variable is implemented by

```haskell
isVar [x]   = True
isVar [x,d] = isDigit d
isVar _     = False
```

Note that any identifier consisting entirely of alphabetic characters and beginning with a letter and which is not a variable is a constant.

Next, we make `Expr` and `Atom` instances of `Show`. As in the previous chapter we
A simple equational calculator

will do this by defining `showsPrec p` for each type. A little thought reveals that we need three values for `p`:

- At top level, there is no need for parentheses. For example, we write all of `map f . map g . foo * baz`, `and bar bie doll` without parentheses. We assign `p=0` to this case.

- When an expression is a composition of terms, or an operator expression, occurring as an argument to a constant, we need to parenthesise it. For example, parentheses are necessary in the expression
  
  \[ \text{map} (f \cdot g) \cdot \text{foo} f g \cdot (\text{bar} \ast \text{bar}) \]

  But we don’t have to parenthesise the middle term. We assign `p=1` to this case.

- Finally, `p=2` means we should parenthesise compositions of terms, operator expressions and curried functions of at least one argument, as in
  
  \[ \text{map} (f \cdot g) \cdot \text{foo} (\text{foldr} f e) g \cdot (\text{bar} \ast \text{bar}) \]

Here goes. We start with

```haskell
instance Show Expr where
  showsPrec p (Compose []) = showString "id"
  showsPrec p (Compose [a]) = showsPrec p a
  showsPrec p (Compose as) = showParen (p>0) (showSep "$ " (showsPrec 1) as)
```

The last line makes use of the function `showSep`, defined by

```haskell
showSep :: String -> (a -> ShowS) -> [a] -> ShowS
showSep sep f = compose . intersperse (showString sep) . map f
```

The utility function `compose` is defined by `compose = foldr (.) id`. The function `intersperse :: a -> [a] -> [a]` can be found in `Data.List` and intersperses its first argument between elements of its second. For example,

```haskell
intersperse ', ' "abcde" == "a,b,c,d,e"
```

The two occurrences of `showsPrec` on the right-hand sides of the second two clauses of `showsPrec` refer to the corresponding function for atoms:

```haskell
instance Show Atom where
  showsPrec p (Var v) = showString v
  showsPrec p (Con f []) = showString f
  showsPrec p (Con f [e1,e2])
```
| isOp f = showParen (p>0) (showsPrec 1 e1 . showSpace .
  showString f . showSpace . showsPrec 1 e2)
  showsPrec p (Con f es)
  = showParen (p>1) (showString f . showSpace .
    showSep " " (showsPrec 2) es)

isOp f = all symbolic f

The value p=2 is needed in the final clause because we want parentheses in, for ex-
ample, foo (bar bie) doll. Variables and nullary constants never need paren-
theses.

\section*{A module structure}

The final step is to install these definitions, and possibly others, in a module for
expressions. Such a module will include all the functions specifically related to
expressions.

Creating such a module is not immediate because we do not yet know what other
functions on expressions we may need in other modules, modules that deal with
laws, calculations and so on. But for the moment we declare

\begin{verbatim}
module Expressions
    (Expr (Compose), Atom (Var,Con),
    VarName, ConName, deCompose, expr)
where
import Parsing
import Data.List (intersperse)
import Utilities (compose)
import Data.Char (isAlphaNum,isAlpha,isDigit)
\end{verbatim}

The module Expressions has to be stored in a file Expressions.lhs to enable
Haskell to find out where it resides. It exports the types Expr and Atom along with
their constructors. It also exports the type synonyms VarName and ConName, as
well as the functions deCompose and expr, all of which are likely to be needed in
the module that deals with laws. Later on we might add more functions on expres-
sions to this export list.

Next comes the imports. We import the module Parsing that contains the parsing
functions, and also some functions from Data.List and Data.Char. We will also
set up a module Utilities containing general utility functions. A good example
A simple equational calculator

of a utility function is \texttt{compose}, defined above. It is not specific to expressions and may be needed in other places, so we put it into the utilities module.

12.3 Laws

We define laws in the following way:

\begin{verbatim}
data Law     = Law LawName Equation
type LawName = String
type Equation = (Expr,Expr)
\end{verbatim}

A law consists of a descriptive name and an equation. To parse a law we define:

\begin{verbatim}
law :: Parser Law
law = do {name <- upto ':';
eqn <- equation;
return (Law name eqn)}
\end{verbatim}

The parsing function \texttt{upto \ c} returns the string up to but not including the character \texttt{c}, and then discards \texttt{c} if found. It wasn’t included among the parsing functions of the previous chapter, but we will put it into the module \texttt{Parsing} to avoid breaking the parser abstraction. One definition is:

\begin{verbatim}
upto :: Char -> Parser String
upto c
= Parser (\s ->
  let (xs,ys) = break (\=c) s in
    if null ys then []
    else [(xs,tail ys)]
\end{verbatim}

The parser equation is defined by

\begin{verbatim}
equation :: Parser Equation
equation = do {e1 <- expr;
  symbol "=";
  e2 <- expr;
  return (e1,e2)}
\end{verbatim}

We probably don’t need to show laws, but here is the definition anyway:

\begin{verbatim}
instance Show Law where
  showsPrec _ (Law name (e1,e2))
    = showString name .
\end{verbatim}
showString "": " .
sshows e1 .
showString " = " .
sshows e2

The precedence number is not needed to define showPrec so it is made a don’t care pattern. Recall that shows takes a printable value, here an expression, and returns a function of type ShowS, a synonym for String \( \rightarrow \) String.

Finally we sort the laws:

\[
\text{sortLaws} :: [\text{Law}] \rightarrow [\text{Law}]
\]
\[
\text{sortLaws} \text{ laws} = \text{simple ++ others ++ defns}
\]
where
\[
(\text{simple}, \text{nonsimple}) = \text{partition} \text{ isSimple} \text{ laws}
\]
\[
(\text{defns}, \text{others}) = \text{partition} \text{ isDefn} \text{ nonsimple}
\]

This definition makes use of a Data.List function partition that partitions a list:

\[
\text{partition} \ p \ \text{xs} = (\text{filter} \ p \ \text{xs}, \text{filter} \ (\text{not} \ . \ p) \ \text{xs})
\]

The various tests are defined by

\[
\text{isSimple} \ (\text{Law } \_ \ (\text{Compose as1,Compose as2}))
\]
\[
= \text{length as1} > \text{length as2}
\]
\[
\text{isDefn} \ (\text{Law } \_ \ (\text{Compose [Con f es], _}))
\]
\[
= \text{all isVar es}
\]
\[
\text{isDefn } \_ = \text{False}
\]
\[
\text{isVar} \ (\text{Compose [Var _]}) = \text{True}
\]
\[
\text{isVar } \_ = \text{False}
\]

The test isVar also appears in the module Expressions though with a different definition. There is no problem though since that function is not exported from the expressions module.

Here is the module declaration for laws:

\[
\text{module} \text{ Laws}
\]
\[
(\text{Law} \ (\text{Law}), \text{LawName}, \text{law}, \text{sortLaws},
\text{Equation}, \text{equation})
\]
where
\[
\text{import} \text{ Expressions}
\]
\[
\text{import} \text{ Parsing}
\]
\[
\text{import} \text{ Data.List} \ (\text{partition})
\]
Having shown how to parse and print expressions and laws, we can now define two functions, one a version of calculate that consumes strings rather than laws and expressions:

```haskell
simplify :: [String] -> String -> Calculation
simplify strings string
    = let laws = map (parse law) strings
        e = parse expr string
    in calculate laws e
```

In a similar vein we can define

```haskell
prove :: [String] -> String -> Calculation
prove strings string
    = let laws = map (parse law) strings
        (e1,e2) = parse equation string
    in paste (calculate laws e1) (calculate laws e2)
```

These two functions can be put in a module Main. We put `paste` and `calculate` into a module concerned solely with calculations, and we turn to this module next.

### 12.4 Calculations

Calculations are defined by

```haskell
data Calculation = Calc Expr [Step]
type Step = (LawName,Expr)
```

Let’s begin with the key definition of the calculator, that of `calculate`:

```haskell
calculate :: [Law] -> Expr -> Calculation
calculate laws e = Calc e (manyStep rws e)
where rws e = [(name,e')
            | Law name eqn <- sortedlaws,
              e' <- rewrites eqn e,
              e' /= e]

sortedlaws = sortLaws laws
```

The function `rewrites :: Equation -> Expr -> [Expr]` returns a list of all the possible ways of rewriting an expression using a given equation, a function that will be defined in a separate module. It may be the case that an expression can be rewritten to itself (see Exercise H), but such rewrites are disallowed because they would lead to infinite calculations. The function `rws :: Expr -> [Step]`
returns a list of all the single steps, leading to new expressions, that can arise by using the laws in all possible ways. This list is defined by taking each law in turn and generating all the rewrites associated with the law. That means we give preference to laws over subexpressions in calculations, resolving one of the issues we worried about in the first section. Only experimentation will show if we have made the right decision.

The function `manyStep` uses `rws` to construct as many steps as possible:

```haskell
manyStep :: (Expr -> [Step]) -> Expr -> [Step]
manyStep rws e
  = if null steps then []
    else step : manyStep rws (snd step)
      where steps = rws e
                step = head steps
```

The calculation ends if `rws e` is the empty list; otherwise the head of the list is used to continue the calculation.

The remaining functions of the calculations module deal with showing and pasting calculations. We show a calculation as follows:

```haskell
instance Show Calculation where
  showsPrec _ (Calc e steps)
    = showString "\n   " .
      shows e .
      showChar '\n' .
      compose (map showStep steps)
```

Each individual step is shown as follows:

```haskell
showStep :: Step -> ShowS
showStep (why,e)
  = showString "= {" .
      showString why .
      showString "}\n   " .
      shows e .
      showChar '\n'
```

In order to paste two calculations together we have to reverse the steps of a calculation. For example, the calculation

```
Calc e0 [(why1,e1),(why2,e2),(why3,e3)]
```

has to be turned into
A simple equational calculator

Calc e3 [(why3,e2),(why2,e1),(why1,e0)]

In particular, the conclusion of a calculation is the first expression in the reversed
calculation. Here is how to reverse a calculation:

reverseCalc :: Calculation -> Calculation
reverseCalc (Calc e steps)
= foldl shunt (Calc e []) steps
  where shunt (Calc e1 steps) (why,e2)
    = Calc e2 ((why,e1):steps)

In order to paste two calculations together we first have to check that their con-
cclusions are the same. If they are not, then we go ahead and paste the calculations
anyway with an indication of failure:

conc1
= {... ??? ...}
conc2

If the two conclusions are the same, we can be a little smarter than just stitch-
ing the calculations together. If the penultimate conclusion of one calculation also
matches the penultimate conclusion of the other, then we can cut out the final steps
altogether. And so on. Here, then, is how we paste two calculations:

paste :: Calculation -> Calculation -> Calculation
paste calc1@(Calc e1 steps1) calc2
  = if conc1 == conc2
     then Calc e1 (prune conc1 rsteps1 rsteps2)
     else Calc e1 (steps1 ++ (gap,conc2):rsteps2)
       where Calc conc1 rsteps1 = reverseCalc calc1
                         Calc conc2 rsteps2 = reverseCalc calc2
                         gap = "... ??? ..."

The function prune is defined by:

prune :: Expr -> [Step] -> [Step] -> [Step]
prune e ((_,e1):steps1) ((_,e2):steps2)
  | e1==e2 = prune e1 steps1 steps2
prune e steps1 steps2 = rsteps ++ steps2
  where Calc _ rsteps = reverseCalc (Calc e steps1)

Finally, here is the module declaration of Calculations:

module Calculations
  (Calculation (Calc), Step, calculate, paste)
where
import Expressions
import Laws
import Rewrites
import Utilities (compose)

The exports are those types and functions needed to define simplify and prove in the main module.

### 12.5 Rewrites

The sole purpose of the module `Rewrites` is to provide a definition of the function `rewrites` that appears in the definition of `calculate`. Recall that the expression `rewrites eqn e` returns a list of all expressions that can arise by matching some subexpression of `e` against the left-hand expression of `eqn` and replacing the subexpression with the appropriate instance of the right-hand expression of `eqn`.

The fun is in figuring out how to define `rewrites`. Suppose we construct a list of all possible subexpressions of an expression. We can match the given equation against each subexpression, get the substitutions that do the matching (of which there may be none, one or more than one; see the section on matching below) and compute the new subexpressions. But how do we replace an old subexpression with a new one in the original expression? The simple answer is that we can’t, at least not without determining alongside each subexpression its context or location in the original expression. The new subexpression can then be inserted at this location.

Rather than introducing contexts explicitly, we take another approach. The idea is to burrow into an expression, applying a rewrite to some subexpression at some point, and then to build the rewritten expression as we climb back out of the burrow. We will need a utility function `anyOne` that takes a function yielding a choice of alternatives, and a list, and installs a single choice for one of the elements. The definition is

\[
\text{anyOne} :: (a -> [a]) -> [a] -> [[a]]
\]

\[
\text{anyOne} f [] = []
\]

\[
\text{anyOne} f (x:xs) = [x':xs | x' <- f x] ++ [x:xs' | xs' <- \text{anyOne} f xs]
\]

For example, if
\[
f 1 = [-1,-2] \text{ and } f 2 = [-3,-4],
\]
then
\[
\text{anyOne} f [1,2] = [\n\quad [-1,2],[2,2],[-2,2],[1,-3],[1,-4]]
\]
Either one of the choices for the first element is installed, or one of the choices for the second, but not both at the same time.

Here is our definition of rewrites:

\[
\text{rewrites} \, : \, \text{Equation} \to \text{Expr} \to \text{[Expr]}
\]
\[
\text{rewrites} \, \text{eqn} \, (\text{Compose as}) = \text{map Compose (}
\text{rewritesSeg eqn as} \, \text{++} \, \text{anyOne} \, \text{(rewritesA eqn) as})
\]
\[
\text{rewritesA eqn} \, (\text{Var} \, v) = \text{[]} \\
\text{rewritesA eqn} \, (\text{Con} \, k \, \text{es}) = \text{map} \, (\text{Con} \, k) \, \text{(anyOne (rewrites eqn) es)}
\]

In the first line we concatenate the rewrites for a segment of the current expression with the rewrites for any one of its proper subexpressions. Only constants with arguments have subexpressions. Note that the two uses of anyOne have different types, one taking a list of atoms, and one taking a list of expressions.

It remains to define rewritesSeg:

\[
\text{rewritesSeg} \, : \, \text{Equation} \to \text{[Atom]} \to \text{[[Atom]]}
\]
\[
\text{rewritesSeg} \, (\text{e1, e2}) \, \text{as} = [\text{as1} \, \text{++} \, \text{deCompose} \, \text{(apply sub e2)} \, \text{++} \, \text{as3} \\
| \, (\text{as1, as2, as3}) \leftarrow \text{segments as}, \text{sub} \leftarrow \text{match (e1, Compose as2)}]
\]

The function segments splits a list into segments:

\[
\text{segments as} = [(\text{as1, as2, as3}) \\
| \, (\text{as1, bs}) \leftarrow \text{splits as}, \text{as2, as3} \leftarrow \text{splits bs}]
\]

The utility function splits splits a list in all possible ways:

\[
\text{splits} \, : \, [\text{a}] \to \text{[[[a],[a]]]} \\
\text{splits} \, [] = [[[[],[[]]]]} \\
\text{splits} \, (\text{a:as}) = [[[[],a:as]] \, \text{++} \\
\text{[(a:as1,as2) | (as1,as2) \leftarrow \text{splits as}]}}
\]

For example,

\[
\text{ghci}> \text{splits "abc"}
\text{[(["","abc"),("a","bc"),("ab","c"),("abc","")]}] \\
\]

The remaining functions apply and match have types

\[
\text{apply} \, : \, \text{Subst} \to \text{Expr} \to \text{Expr}
\]
match :: (Expr,Expr) -> [Subst]

Each will be defined in their own modules, Substitutions and Matchings. Finally, here is the module declaration for Rewrites:

module Rewrites (rewrites)
where
import Expressions
import Laws (Equation)
import Matchings (match)
import Substitutions (apply)
import Utilities (anyOne, segments)

12.6 Matchings

The sole purpose of the module Matchings is to define the function match. This function takes two expressions and returns a list of substitutions under which the first expression can be transformed into the second. Matching two expressions produces no substitutions if they don’t match, but possibly many if they do. Consider matching the expression foo (f . g) against foo (a . b . c). There are four substitutions that do the trick: f may be bound to any of the expressions

id, a, a . b, a . b . c

with four corresponding bindings for g. Although the calculator will select a single substitution at each step, it is important to take account of multiple substitutions in the process of obtaining the valid matchings. For example, in matching foo (f . g) . bar g against foo (a . b . c) . bar c, the subexpression f . g is matched against a . b . c, resulting in four possible substitutions. Only when bar g is matched against bar c are three of the substitutions rejected. A premature commitment to a single substitution for the first match may result in a successful match being missed.

The most straightforward way of defining match (e1,e2) is to first line up the atoms of e1 with a partition of the atoms of e2; the first atom is associated with the first segment of the partition, the second with the second segment, and so on. The function alignments has type

alignments :: (Expr,Expr) -> [[[Atom,Expr]]]

and does the alignments. To define it we need a function parts that partitions a list into a given number of segments:
A simple equational calculator

parts :: Int -> [a] -> [[[a]]]
parts 0 [] = [[]]
parts 0 as = []
parts n as = [bs:bss
| (bs,cs) <- splits as,
bss <- parts (n-1) cs]

The interesting clauses are the first two: there is one partition of the empty list into
0 segments, namely the empty partition, but there are no partitions of a nonempty
list into 0 segments. For example,

ghci> parts 3 "ab"
["","","ab"],["","a","b"],["","ab",""],
["a","","b"],["a","b",""],["ab","",""]]

Now we can define

alignments (Compose as,Compose bs)
= [zip as (map Compose bss) | bss <- parts n bs]
  where n = length as

Having aligned each atom with a subexpression, we define matchA that matches
atoms with expressions:

matchA :: (Atom,Expr) -> [Subst]
matchA (Var v,e) = [unitSub v e]
matchA (Con k1 es1,Compose [Con k2 es2])
  | k1==k2 = combine (map match (zip es1 es2))
matchA _ = []

Matching a variable always succeeds and results in a single substitution. Matching
two constants succeeds only if the two constants are the same. In all other cases
matchA returns an empty list of substitutions. The function matchA depends on
match, which we can now define by

match :: (Expr,Expr) -> [Subst]
match = concatMap (combine . map matchA) . alignments

The final ingredient is the function combine :: [[[Subst]]] -> [Subst]. Each
component list of substitutions in the argument of combine represents alternatives,
so combine has to combine alternatives by selecting, in all possible ways, one sub-
stitution from each list and then unifying the result. We will return to this function
in the module for substitutions. This completes the definition of matches. The
module declaration is
12.7 Substitutions

A substitution is a finite mapping associating variables with expressions. A simple representation as an association list suffices:

```haskell
type Subst = [(VarName,Expr)]
```

The empty and unit substitutions are then defined by

```haskell
emptySub = []
unitSub v e = [(v,e)]
```

We can apply a substitution to an expression to get another expression by defining

```haskell
apply :: Subst -> Expr -> Expr
apply sub (Compose as) = Compose (concatMap (applyA sub) as)
applyA sub (Var v) = deCompose (binding sub v)
applyA sub (Con k es) = [Con k (map (apply sub) es)]
```

The function `binding` looks up a nonempty substitution for the binding for a variable:

```haskell
binding :: Subst -> VarName -> Expr
binding sub v = fromJust (lookup v sub)
```

The function `lookup` is supplied in the Haskell Prelude and returns `Nothing` if no binding is found, and `Just e` if `v` is bound to `e`. The function `fromJust` is in the library `Data.Maybe` and removes the wrapper `Just`.

Next we tackle `combine`. This function has to combine alternative substitutions by selecting, in all possible ways, one substitution from each component list and then unifying each resulting list of substitutions:

```haskell
combine = concatMap unifyAll . cp
```
A simple equational calculator

The utility function \( cp \), which we have seen many times before, computes the cartesian product of a list of lists.

The function \( unifyAll \) takes a list of substitutions and unifies them. To define it we first show how to unify two substitutions. The result of unification is either the union of the two substitutions if they are compatible, or no substitution if they are incompatible. To handle the possibility of failure, we can use the Maybe type, or simply return either an empty list or a singleton list. We choose the latter simply because in the following section we are going to calculate another version of the calculator, and it is simplest to stick with list-based functions:

\[
\begin{align*}
\text{unify} & : \text{Subst} \to \text{Subst} \to [\text{Subst}] \\
\text{unify sub1 sub2} & = \begin{cases} 
\text{union sub1 sub2} & \text{if compatible sub1 sub2} \\
[] & \text{else}
\end{cases}
\end{align*}
\]

In order to define \( \text{compatible} \) and \( \text{union} \) we will suppose that substitutions are maintained as lists in lexicographic order of variable name. Two substitutions are incompatible if they associate different expressions with one and the same variable:

\[
\begin{align*}
\text{compatible} [\ ] \text{sub2} & = \text{True} \\
\text{compatible} \text{sub1} [\ ] & = \text{True} \\
\text{compatible} \text{sub1@((v1,e1):sub1')} \text{sub2@((v2,e2):sub2')} & = \\
& \begin{cases} 
\text{compatible sub1' sub2} & \text{if } v1<v2 \\
\text{compatible sub1' sub2'} & \text{if } v1==v2 \text{ then } e1==e2 \text{ then compatible sub1' sub2'} \\
\text{else False} & \text{else}
\end{cases} \\
& \text{compatible sub1 sub2'} \text{if } v1>v2
\end{align*}
\]

The union operation is defined in a similar style:

\[
\begin{align*}
\text{union} [\ ] \text{sub2} & = \text{sub2} \\
\text{union} \text{sub1} [\ ] & = \text{sub1} \\
\text{union} \text{sub1@((v1,e1):sub1')} \text{sub2@((v2,e2):sub2')} & = \\
& \begin{cases} 
(v1,e1):\text{union sub1' sub2} & \text{if } v1<v2 \\
(v1,e1):\text{union sub1' sub2'} & \text{if } v1==v2 \\
(v2,e2):\text{union sub1 sub2'} & \text{if } v1>v2
\end{cases}
\end{align*}
\]

The function \( \text{unifyAll} \) returns either an empty list or a singleton list:

\[
\begin{align*}
\text{unifyAll} & : [\text{Subst}] \to [\text{Subst}] \\
\text{unifyAll} = \text{foldr f [emptySub]} \\
& \text{where f sub subs} = \text{concatMap (unify sub) subs}
\end{align*}
\]

That completes the definitions we need. Here is the module declaration:
module Substitutions
  (Subst, unitSub, combine, apply)
where
import Expressions
import Utilities (cp)
import Data.Maybe (fromJust)

That makes nine modules in total for our calculator.

12.8 Testing the calculator

How useful is the calculator in practice? The only way to answer this question is to try it out on some examples. We are going to record just two. The first is the calculation we performed in Chapter 5 about pruning the matrix of choices in Sudoku. In effect we want to prove

\[
\text{filter (all nodups . boxs) . expand . pruneBy boxs} = \text{filter (all nodups . boxs) . expand}
\]

from the laws

\[
\begin{align*}
\text{defn pruneBy:} & \quad \text{pruneBy } f = f . \text{map pruneRow . } f \\
\text{expand after boxs:} & \quad \text{expand . boxs} = \text{map boxs . expand} \\
\text{filter with boxs:} & \quad \text{filter (p . boxs)} \quad = \text{map boxs . filter p . map boxs} \\
\text{boxs involution:} & \quad \text{boxs . boxs} = \text{id} \\
\text{map functor:} & \quad \text{map } f . \text{map } g = \text{map (f.g)} \\
\text{map functor:} & \quad \text{map } \text{id} = \text{id} \\
\text{defn expand:} & \quad \text{expand} = \text{cp . map cp} \\
\text{filter after cp:} & \quad \text{filter (all p) . cp} = \text{cp . map (filter p)} \\
\text{law of pruneRow:} & \quad \text{filter nodups . cp . pruneRow} \quad = \text{filter nodups . cp}
\end{align*}
\]

Here is the calculation exactly as performed by the calculator, except that we have broken some expressions across two lines, a task that should be left to a pretty-printer. Don’t bother to study it in detail, just note the important bit towards the end:

\[
\begin{align*}
\text{filter (all nodups . boxs) . expand . pruneBy boxs} \\
= \quad \{\text{filter with boxs}\} \\
\quad \text{map boxs . filter (all nodups) . map boxs . expand .} \\
\quad \text{pruneBy boxs} \\
= \quad \{\text{defn pruneBy}\} \\
\quad \text{map boxs . filter (all nodups) . map boxs . expand .} \\
\quad \text{boxs . map pruneRow . boxs} \\
= \quad \{\text{expand after boxs}\}
\end{align*}
\]
map boxs . filter (all nodups) . map boxs . expand . map pruneRow . boxs
= \{ map functor \}
map boxs . filter (all nodups) . map (boxs . boxs) . expand . map pruneRow . boxs
= \{ boxs involution \}
map boxs . filter (all nodups) . map id . expand . map pruneRow . boxs
= \{ map functor \}
map boxs . filter (all nodups) . expand . map pruneRow . boxs
= \{ defn expand \}
map boxs . filter (all nodups) . cp . map cp . map pruneRow . boxs
= \{ map functor \}
map boxs . filter (all nodups) . cp . map (cp . pruneRow) . boxs
= \{ filter after cp \}
map boxs . cp . map (filter nodups) . map (cp . pruneRow) . boxs
= \{ map functor \}
map boxs . cp . map (filter nodups . cp . pruneRow) . boxs
= \{ law of pruneRow \}
map boxs . cp . map (filter nodups . cp) . boxs
= \{ \ldots \text{ ???} \ldots \}\}
map boxs . filter (all nodups) . map boxs . cp . map cp
= \{ defn expand \}
map boxs . filter (all nodups) . map boxs . expand
= \{ filter with boxs \}
filter (all nodups . boxs) . expand

Yes, the calculation fails. The reason is not hard to spot: we need to apply the law

```
expand after boxs: expand . boxs = map boxs . expand
```

in both directions, and the calculator simply cannot do that.

The solution is a hack. We add in the extra law

```
hack: map boxs . cp . map cp = cp . map cp . boxs
```

which is just the expand after boxs law written in the opposite direction and with expand replaced by its definition. Then the calculator is happy, producing the conclusion

```
\ldots
```

```
map boxs . cp . map (filter nodups . cp) . boxs
= \{ map functor \}
map boxs . cp . map (filter nodups) . map cp . boxs
= \{ filter after cp \}
map boxs . filter (all nodups) . cp . map cp . boxs
= \{ hack \}
map boxs . filter (all nodups) . map boxs . cp . map cp
= \{ defn expand \}
map boxs . filter (all nodups) . map boxs . expand
```
= \{\text{filter with boxes}\}
  \text{filter (all nodups . boxes) . expand}

In both cases the calculations were performed in a fraction of a second, so efficiency does not seem to be an issue. And, apart from the hack, the calculations pass muster, being almost exactly what a good human calculator would produce.

*Improving the calculator*

Our second example is more ambitious: we are going to use the calculator to derive another version of the calculator. Look again at the definition of \texttt{match}. This relies on \texttt{combine}, which in turn involves a messy appeal to the unification of two substitutions, with all the paraphernalia of having to test them for compatibility and computing the union. A better idea is to compute the union of two substitutions only when one of them is a unit substitution. Then everything becomes simpler and probably faster. And the technique which describes this optimisation? Yes, it’s another example of accumulating parameters. Just as an accumulating parameter can avoid expensive uses of \texttt{++} operations, our hope is to avoid expensive \texttt{unify} operations.

First of all, here is the definition of \texttt{match} again, written with a couple of new subsidiary functions:

\begin{verbatim}
match = concatMap matchesA . alignments
matchesA = combine . map matchA
matchA (Var v,e) = \[\text{unitSub v e}\]
matchA (Con k1 es1,Compose [Con k2 es2])
  \mid k1==k2 = matches (zip es1 es2)
matchA _ = []
matches = combine . map match
\end{verbatim}

Note the cycle of dependencies of these functions:

\[
\text{match} \rightarrow \text{matchesA} \rightarrow \text{matchA} \rightarrow \text{matches} \rightarrow \text{match}
\]

These four functions are generalised as follows:

\begin{verbatim}
xmatch sub = concatMap (unify sub) . match
xmatchA sub = concatMap (unify sub) . matchA
xmatches sub = concatMap (unify sub) . matches
xmatchesA sub = concatMap (unify sub) . matchesA
\end{verbatim}

The additional argument in each case is an accumulating parameter. Our aim will
be to obtain new versions of these definitions, whose cycle of dependencies is the same as the one above:

For the first calculation, we want to rewrite \( \text{match} \) in terms of \( \text{xmatch} \), thereby linking the two groups of definitions. To save a lot of ink, we henceforth abbreviate \( \text{concatMap} \) to \( \text{cmap} \). The three laws we need are

\[
\text{defn xmatch: } \quad \text{xmatch s} = \text{cmap} \left( \text{unify s} \right) \cdot \text{match}
\]

\[
\text{unify of empty: } \quad \text{unify emptySub} = \text{one}
\]

\[
\text{cmap of one: } \quad \text{cmap one} = \text{id}
\]

In the first law we have to write \( s \) rather than \( \text{sub} \) (why?); the second two laws are the pointless versions of the facts that

\[
\text{unify emptySub sub} = [\text{sub}]
\]

\[
\text{cmap one xs} = \text{concat} \left( [x] \mid x \leftarrow \text{xs} \right) = \text{xs}
\]

The calculator is hardly stretched to give:

\[
\text{xmatch emptySub} = \{\text{defn xmatch}\}
\]

\[
\text{cmap} \left( \text{unify emptySub} \right) \cdot \text{match} = \{\text{unify of empty}\}
\]

\[
\text{cmap one} \cdot \text{match} = \{\text{cmap of one}\}
\]

Let us next deal with \( \text{xmatchA} \). Because of the awkward pattern-matching style of definition of \( \text{matchA} \), we simply record the following result of an easy (human) calculation:

\[
\text{xmatchA sub (Var v,e)} = \text{concat} \left( \text{unify sub (unitSub v e)} \right)
\]

\[
\text{xmatchA sub (Con k1 es1,Compose [Con k2 es2])}
\]

\[
| k1==k2 = \text{xmatches sub (zip es1 es2)}
\]

\[
\text{xmatchA \_} = [\_]
\]

If we introduce

\[
\text{extend sub v e} = \text{concat} \left( \text{unify sub (unitSub v e)} \right)
\]

then it is easy to derive

\[
\text{extend sub v e} = \text{case lookup v sub of}
\]

\[
\text{Nothing} \rightarrow [(v,e):\text{sub}]
\]

\[
\text{Just e'} \rightarrow \text{if e==e' then [sub] else []}
\]
No elaborate compatibility test, and no general union of two substitutions. Instead, as we promised earlier, we unify substitutions only with unit substitutions.

Having disposed of $xmatchA$ we concentrate on the other three members of the quartet. Just as $xmatchA$ is defined in terms of $xmatches$, so $xmatch$ can be defined in terms of $xmatchesA$. Specifically, we want to prove that

$$xmatch \; s = cmap \; (xmatchesA \; s) \; . \; alignments$$

Here are the laws we need:

```plaintext
defn match: match = cmap \; matchesA \; . \; alignments
defn xmatch: xmatch \; s = cmap \; (unify \; s) \; . \; match
defn xmatchesA: xmatchesA \; s = cmap \; (unify \; s) \; . \; matchesA
cmap \; after \; cmap: cmap \; f \; . \; cmap \; g = cmap \; (cmap \; f \; . \; g)
```

The last, purely combinatorial law is new; we leave verification as an exercise. The calculator produces:

```plaintext
xmatch \; s
= \{defn xmatch\}
cmap \; (unify \; s) \; . \; match
= \{defn match\}
cmap \; (unify \; s) \; . \; cmap \; matchesA \; . \; alignments
= \{cmap \; after \; cmap\}
cmap \; (cmap \; (unify \; s) \; . \; matchesA) \; . \; alignments
= \{defn xmatchesA\}
cmap \; (xmatchesA \; s) \; . \; alignments
```

So far, so good. That leaves us with the two remaining members of the quartet, $xmatches$ and $xmatchesA$. In each case we want to obtain recursive definitions, ones that do not involve $unify$. The two functions are defined in a very similar way, and it is likely that any calculation about one can be adapted immediately to the other. This kind of meta-calculational thought is, of course, beyond the reaches of the calculator.

Let us concentrate on $xmatchesA$. We first make $xmatchesA$ entirely pointless, removing the parameter $s$ in the definition above. The revised definition is:

```plaintext
xmatchesA :: (Subst,[(Atom,Expr)]) -> Subst
xmatchesA = cup \; . \; (one \; * \; matchesA)
cup = cmap \; unify \; . \; cpp
```

where the combinator $cpp$ is defined by

```plaintext
cpp \; (xs,ys) = [(x,y)| x <- xs, y <- ys]
```

Thus
A simple equational calculator

\[
\text{xmatchesA (sub,aes)} = \text{cup ([sub],aes)} = \text{concat [unify (s,ae) | s <- [sub], ae <- matchesA aes]} = \text{concat [unify (sub,ae) | ae <- matchesA aes]}
\]

Apart from the fact that unify is now assumed to be a non-curried function, this is a faithful rendition of the definition of \text{xmatchesA} in pointless form.

The new function \text{cup} has type \([\text{Subst}] \rightarrow [\text{Subst}] \rightarrow [\text{Subst}]\). Later on we will exploit the fact that \text{cup} is an associative function, something that unify could never be (why not?). As we saw in Chapter 7 the accumulating parameter technique depends on the operation of interest being associative.

The first thing to check is that the previous calculation is still valid with the new definitions. Suppose we set up the laws

\[
\text{defn match: } \text{match} = \text{cmap matchesA} \cdot \text{alignments}
\]

\[
\text{defn xmatch: } \text{xmatch} = \text{cup} \cdot (\text{one} \star \text{match})
\]

\[
\text{defn xmatchesA: } \text{xmatchesA} = \text{cup} \cdot (\text{one} \star \text{matchesA})
\]

The calculator then produces

\[
\text{xmatch} = \{\text{defn xmatch}\} \\text{cup} \cdot (\text{one} \star \text{match}) = \{\text{defn match}\} \\text{cup} \cdot (\text{one} \star (\text{cmap matchesA} \cdot \text{alignments})) = \{... ??? ...\}
\]

\[
\text{cmap (cup} \cdot (\text{one} \star \text{matchesA})) \cdot \text{cpp} \cdot (\text{one} \star \text{alignments}) = \{\text{defn xmatchesA}\} \\text{cmap xmatchesA} \cdot \text{cpp} \cdot (\text{one} \star \text{alignments})
\]

Ah, it doesn’t go through. Inspecting the gap in the calculation, it seems we need both the bifunctor law of \(*\) and a claim relating \text{cmap} and \text{cup}:

\[
\text{cross bifunctor: } (f \star g) \cdot (h \star k) = (f \cdot h) \star (g \cdot k)
\]

\[
\text{cmap-cup: } \text{cmap (cup} \cdot (\text{one} \star g)) \cdot \text{cpp} = \text{cup} \cdot (\text{id} \star \text{cmap g})
\]

The calculator is then happy:

\[
\text{xmatch} = \{\text{defn xmatch}\} \\text{cup} \cdot (\text{one} \star \text{match}) = \{\text{defn match}\} \\text{cup} \cdot (\text{one} \star (\text{cmap matchesA} \cdot \text{alignments})) = \{\text{cross bifunctor}\} \\text{cup} \cdot (\text{id} \star \text{cmap matchesA}) \cdot (\text{one} \star \text{alignments}) = \{\text{cmap-cup}\} \\text{cmap (cup} \cdot (\text{one} \star \text{matchesA})) \cdot \text{cpp} \cdot (\text{one} \star \text{alignments}) = \{\text{defn xmatchesA}\}
\]
That still leaves us with the claim; apart from the fact that it works we have no reason to suppose it is true. However, we can get the calculator to prove it by using another law that is not specific to matching. We leave the proof as Exercise M. Define the additional laws

\[
\text{defn } cup: \quad cup = \text{cmap unify . cpp} \\
\text{cmap-cpp: } \text{cmap (cpp . (one * f)) . cpp = cpp . (id * cmap f)}
\]

The calculator then produces

\[
\text{cmap (cup . (one * g)) . cpp} \\
= \{\text{defn } cup\} \\
\text{cmap (cmap unify . cpp . (one * g)) . cpp} \\
= \{\text{cmap after cmap}\} \\
\text{cmap unify . cmap (cpp . (one * g)) . cpp} \\
= \{\text{cmap-cpp}\} \\
\text{cmap unify . cpp . (id * cmap g)} \\
= \{\text{defn } cup\} \\
\text{cup . (id * cmap g)}
\]

Good. It seems that the cmap-cup law is valid, and it even might be useful again later on. Now let us return to the main point, which is to express xmatchesA recursively by two equations of the form

\[
xmatchesA . (id * \text{nil}) = \ldots \\
xmatchesA . (id * \text{cons}) = \ldots
\]

The hope is that such a definition will not involve unify.

It is not at all clear what laws we need for this purpose. Instead, we will write down every law we can think of that might prove useful. The first group consists of our main definitions:

\[
\text{defn } match: \quad match = \text{cmap matchesA . alignments} \\
\text{defn } matchesA: \quad matchesA = \text{combine . map matchA} \\
\text{defn } xmatch: \quad xmatch = \text{cup . (one * match)} \\
\text{defn } xmatchesA: \quad xmatchesA = \text{cup . (one * matchesA)} \\
\text{defn } xmatchA: \quad xmatchA = \text{cup . (one * matchA)} \\
\text{defn } combine: \quad combine = \text{cmap unifyAll . cp}
\]

The second group are some new laws about cmap:

\[
\text{cmap after map: } \text{cmap } f . \text{ map } g = \text{cmap } (f . g) \\
\text{cmap after concat: } \text{cmap } f . \text{ concat } = \text{cmap } (\text{cmap } f) \\
\text{cmap after nil: } \text{cmap } f . \text{ nil } = \text{nil} \\
\text{cmap after one: } \text{cmap } f . \text{ one } = f
\]

The third group are some new laws about map:
A simple equational calculator

map after nil: map f . nil = nil
map after one: map f . one = one . f
map after cons: map f . cons = cons . (f * map f)
map after concat: map f . concat = concat . map (map f)

The fourth group concerns cup:

cup assoc: cup . (id * cup) = cup . (cup * id) . assocl
cup ident: cup . (f * (one . nil)) = f . fst
cup ident: cup . ((one . nil) * g) = g . snd
assocl: assocl. (f * (g * h)) = ((f * g) * h) . assocl

Finally we add in various other definitions and laws:

cross bifunctor: (f * g) . (h * k) = (f . h) * (g . k)
cross bifunctor: (id * id) = id
defn cp: cp . nil = one . nil
defn cp: cp . cons = map cons . cpp . (id * cp)
defn unifyAll: unifyAll . nil = one . nil
defn unifyAll: unifyAll . cons = cup . (one * unifyAll)
unify after nil: unify . (id * nil) = one . fst

That’s a total of 30 laws (including the two map functor laws and three laws about cmap that we haven’t repeated). We cross our fingers and hope:

xmatchesA . (id * nil)
= {defn xmatchesA}
cup . (one * matchesA) . (id * nil)
= {cross bifunctor}
cup . (one * (matchesA . nil))
= {defn matchesA}
cup . (one * (combine . map matchA . nil))
= {map after nil}
cup . (one * (combine . nil))
= {defn combine}
cup . (one * (cmap unifyAll . cp . nil))
= {defn cp}
cup . (one * (cmap unifyAll . one . nil))
= {cmap after one}
cup . (one * (unifyAll . nil))
= {defn unifyAll}
cup . (one * (one . nil))
= {cup ident}
one . fst

That’s gratifying. We have shown that xmatchesA sub [] = [sub]. However, the recursive case cannot be established so easily. Instead we have to guess the result and then try to prove it. Here is the desired result, first expressed in pointed form and then in pointless form:

xmatchesA sub (ae:aes)
= concat [xmatchesA sub' aes | sub' <- xmatchA sub ae]

xmatchesA . (id * cons)
= cmap xmatchesA . cpp . (xmatchA * one) . assocl

We can perform simplification with the right-hand side (we temporarily remove the definitions of xmatchA and matchesA from laws2):

  cmap xmatchesA . cpp . (xmatchA * one) . assocl
= {defn xmatchesA}
  cmap (cup . (one * matchesA)) . cpp . (xmatchA * one) . assocl
= {cmap-cup}
  cup . (id * cmap matchesA) . (xmatchA * one) . assocl
= {cross bifunctor}
  cup . (xmatchA * (cmap matchesA . one)) . assocl
= {cmap after one}
  cup . (xmatchA * matchesA) . assocl

Now we would like to show

  xmatchesA . (id * cons)
  = cup . (xmatchA * matchesA) . assocl

But unfortunately the calculator can't quite make it. The gap appears here:

  cup . ((cup . (one * matchA)) * matchesA)
= {... ???? ...}
  cup . (one * (cup . (matchA * matchesA))) . assocl

The gap is easily eliminable by hand:

  cup . ((cup . (one * matchA)) * matchesA)
= {cross bifunctor (backwards})
  cup . (cup * id) . ((one * matchA) * matchesA)
= {cup assoc}
  cup . (id * cup) . assocl . ((one * matchA) * matchesA)
= {assocl}
  cup . (id * cup) . (one * (matchA * matchesA)) . assocl
= {cross bifunctor}
  cup . (one * (cup . (matchA * matchesA))) . assocl

Once again, the inability to apply laws in both directions is the culprit. Instead of trying to force the laws into a form that would be acceptable to the calculator, we leave it here with the comment 'A hand-finished product!'.

To round off the example, here is the program we have calculated:

  match = xmatch emptySub
  xmatch sub (e1,e2)
  = concat [xmatchesA sub aes | aes <- alignments (e1,e2)]
xmatchesA sub [] = [sub]
xmatchesA sub (ae:aes)
   = concat [xmatchesA sub' aes | sub' <- xmatchA sub ae]

xmatchA sub (Var v,e) = extend sub v e
xmatchA sub (Con k1 es1,Compose [Con k2 es2])
   | k1==k2 = xmatches sub (zip es1 es2)
xmatchA _ = []

The missing definition is that of xmatches. But exactly the same treatment for
xmatchesA goes through for matches, and we end up with

xmatches sub [] = [sub]
xmatches sub ((e1,e2):es)
   = concat [xmatches sub' es | sub' <- xmatch sub (e1,e2)]

Conclusions

The positive conclusion of these two exercises is that one can indeed get the calcu-
lator to assist in the construction of formal proofs. But there remains the need for
substantial human input to the process, to set up appropriate laws, to identify sub-
sidiary claims and to control the order in which calculations are carried out. The
major negative conclusion is that it is a significant failing of the calculator to be
unable to apply laws in both directions. The functor laws are the major culprits, but
there are others as well (see the exercises for some examples). The calculator can
be improved in a number of ways, but we leave further discussion to the exercises.

There are three other aspects worth mentioning about the calculator. Firstly, the
complete calculator is only about 450 lines of Haskell, and the improved version
is even shorter. That alone is a testament to the expressive power of functional
programming. Secondly, it does seem a viable approach to express laws as purely
functional equations and to use a simple equational logic for conducting proofs. To
be sure, some work has to be done to express definitions in point-free form, but
once this is achieved, equational logic can be surprisingly effective.

The third aspect is that, apart from parsing, no monadic code appears in the calcu-
lator. In fact, earlier versions of the calculator did use monads, but gradually they
were weeded out. One reason was that we found the code became simpler with-
out monads, without significant loss of efficiency; another was that we wanted to
set things up for the extended exercise in improving the calculator. Monads are
absolutely necessary for many applications involving interacting with the world, but they can be overused in places where a purely functional approach would be smoother.

On that note, we end.

12.9 Exercises

**Exercise A**

Suppose we did want calculate to return a tree of possible calculations. What would be a suitable tree to use?

**Exercise B**

Why should the laws

\[
\text{map } (f \cdot g) = \text{map } f \cdot \text{map } g \\
\text{cmap } (f \cdot g) = \text{cmap } f \cdot \text{map } g
\]

*never* be used in calculations, at least if they are given in the form above?

**Exercise C**

Here is a calculation, as recorded by the calculator

\[
\text{map } f \cdot \text{map } g \ h \\
= \{\text{map } \text{functor}\} \\
\text{map } (f \cdot g)
\]

Explain this strange and clearly nonsensical result. What simple change to the calculator would prevent the calculation from being valid?

**Exercise D**

On the same general theme as the previous question, one serious criticism of the calculator is that error messages are totally opaque. For example, both

\[
\text{parse law } "\text{map } f \cdot \text{map } g = \text{map } (f \cdot g)" \\
\text{parse law } "\text{map } \text{functor: map } f \cdot \text{map } g \map (f \cdot g)"
\]

cause the same cryptic error message. What is it? What would be the effect of using the law

\[
\text{strange: map } f \cdot \text{map } g = \text{map } h
\]
A simple equational calculator

Exercise E

The definition of showsPrec for atoms makes use of a fact about Haskell that we haven’t needed before. And the same device is used in later calculator functions that mix a pattern-matching style with guarded equations. What is the fact?

Exercise F

Define

\[
\begin{align*}
e_1 &= \text{foo} (f \cdot g) \cdot g \\
e_2 &= \text{bar} f \cdot \text{baz} g
\end{align*}
\]

List the expressions that \text{rewrites} \ (e_1, e_2) produces when applied to the expression \text{foo} (a \cdot b \cdot c) \cdot c. Which one would the calculator pick?

Exercise G

Can the calculator successfully match \text{foo} f \cdot \text{foo} f with the expression

\[
\text{foo} (\text{bar} g h) \cdot \text{foo} (\text{bar} (\text{daz} a) b)
\]

Exercise H

It was claimed in the text that it is possible to apply a perfectly valid non-trivial law that will leave some expressions unchanged. Give an example of such a law and an expression that is rewritten to itself.

Exercise I

The function anyOne used in the definition of rewrites installs a single choice, but why not use everyOne that installs every choice at the same time? Thus if \(f \ 1 = [-1,-2]\) and \(f \ 2 = [-3,-4]\), then

\[
\text{everyOne} f \ [1,2] = [[-1,-3], [-1,-4], [-2,-3], [-3,-4]]
\]

Using everyOne instead of anyOne would mean that a rewrite would be applied to every possible subexpression that matches a law. Give a definition of everyOne.
Exercise J

How many segments of a list of length \(n\) are there? The definition of \(\text{rewritesSeg}\) is inefficient because the empty segment appears \(n+1\) times as the middle component of the segments of a list of length \(n\). That means matching with \(\text{id}\) is performed \(n+1\) times instead of just once. How would you rewrite \(\text{segments}\) to eliminate these duplicates?

Exercise K

Prove that \(\text{cmap } f . \text{ cmap } g = \text{ cmap } (\text{ cmap } f . \text{ g})\). The laws needed are:

- defn cmap: \(\text{cmap } f = \text{ concat . map } f\)
- map functor: \(\text{map } f . \text{ map } g = \text{ map } (f . g)\)
- map after concat: \(\text{map } f . \text{ concat } = \text{ concat . map } (\text{map } f)\)
- concat twice: \(\text{concat . concat } = \text{ concat . map } \text{ concat}\)

Exercise L

The \(\text{cmap-cpp}\) law is as follows:

\[\text{cmap } (\text{cpp . (one } \ast f)) . \text{cpp} = \text{cpp . (id } \ast \text{ cmap } f)\]

Prove it from the laws

- cmap after cmap: \(\text{cmap } f . \text{ map } g = \text{ cmap } (f . g)\)
- cmap after cpp: \(\text{cmap } \text{cpp} . \text{cpp} = \text{cpp . (concat } \ast \text{ concat)}\)
- cross bifunctor: \((f \ast g) . (h \ast k) = (f . h) \ast (g . k)\)
- map after cpp: \(\text{map } (f \ast g) . \text{cpp} = \text{cpp . (map } f \ast \text{ map } g)\)
- defn cmap: \(\text{cmap } f = \text{ concat . map } f\)
- concat after id: \(\text{concat . map } \text{ one } = \text{id}\)

Can a calculator conduct the proof?

Answer to Exercise A

We would want expressions as labels of nodes and law names as labels of edges. That gives

\[
\text{type Calculation } = \text{ Tree Expr LawName}\\
\text{data Tree a b } = \text{ Node a [(b,Tree a b)]}
\]
**Answer to Exercise B**

They would both cause the calculator to spin off into an infinite calculation. For example,

```haskell
m map foo = {map functor}
m map foo . map id = {map functor}
m map foo . map id . map id
```

and so on.

**Answer to Exercise C**

The expression `map f . map g h` is perfectly valid by the rules of syntax, but of course it shouldn’t be. The evaluator does not force the restriction that each appearance of one and the same constant should possess the same number of arguments. The reason the functor law can be matched successfully against the expression is that in the definition of `matchA` the function `zip` truncates the two arguments to the second `map` to one. A better calculator should check that each constant has a fixed arity.

**Answer to Exercise D**

The cryptic message is ‘head of empty list’. The first parse fails because the law is missing its name, and the second is missing an equals sign. Use of the strange law would cause the calculator to fall over because pattern-matching with the left-hand side would not bind `h` to any expression, causing an error when the binding for `h` is requested. The calculator should have checked that every variable on the right-hand side of a law appears somewhere on the left-hand side.

**Answer to Exercise E**

The code for `showsPrec` takes the form

```haskell```
showsPrec p (Con f [e1,e2])
| isOp f = expression1 e1 e2
showsPrec p (Con f es)
= expression2 es
```

A more ‘mathematical’ style would have been to write

```haskell```
showsPrec p (Con f [e1,e2])
| isOp f = expression1 e1 e2
| otherwise = expression2 [e1,e2]
showsPrec p (Con f es) = expression2 es

The point is this: in a given clause if a pattern does not match the argument, or if it does but the guard fails to be true, the clause is abandoned and the next clause is chosen.

**Answer to Exercise F**

There are two rewrites, not one:

\[
\begin{align*}
\text{bar} (a \ . \ b \ . \ c) & . \ \text{baz id} \ . \ c \\
\text{bar} (a \ . \ b) & . \ \text{baz} \ c
\end{align*}
\]

The calculator would pick the first subexpression that matches, and that means the first rewrite is chosen. Perhaps it would be better to arrange that \texttt{rewritesSeg} is applied to longer segments before shorter ones.

**Answer to Exercise G**

No, not with our definition of \texttt{match}. They can be matched by binding \(f\) to the expression \(\text{bar} (\text{daz} \ a) \ b\) provided \(g\) is bound to \(\text{daz} \ a\) and \(h\) to \(b\), but our definition of \texttt{match} does not perform full unification.

**Answer to Exercise H**

To take just one example out of many, consider the law

\[
\text{if } p \ f \ g . \ h = \text{if } (p . \ h) (f . \ h) (g . \ h)
\]

The left-hand side matches \text{if} \ a \ b \ c with \(h\) bound to \(\text{id}\), and the result is again the same expression.

**Answer to Exercise I**

The temptation is to define

\[
\text{everyOne } f = \text{cp} . \ \text{map} \ f
\]

but that doesn’t work if \(f\) returns no alternatives for some element. Instead we have to define

\[
\begin{align*}
\text{everyOne} :: \ (a \rightarrow \ [a]) & \rightarrow \ [a] \rightarrow \ [[a]] \\
\text{everyOne \ } f & = \text{cp} . \ \text{map} \ (\text{possibly \ } f) \\
\text{possibly \ } f \ x & = \text{if \ null \ } xs \ \text{then} \ [x] \ \text{else} \ xs \\
& \ \text{where} \ xs = f \ x
\end{align*}
\]

In this version, \(f\) returns a nonempty list of alternatives.
A simple equational calculator

Answer to Exercise J

There are \((n+1)(n+2)/2\) segments of a list of length \(n\). The improved definition is

\[
\text{segments } \text{xs} = [([],[],\text{xs}] ++
   [(\text{as},\text{bs},\text{cs})
   \mid (\text{as},\text{ys}) \leftarrow \text{splits } \text{xs},
   (\text{bs},\text{cs}) \leftarrow \text{tail } (\text{splits } \text{ys})]
\]

Answer to Exercise K

The calculator produced:

\[
\begin{align*}
\text{cmap } f . \text{ cmap } g &= \{\text{defn cmap}\} \\
\text{concat . map } f . \text{ cmap } g &= \{\text{defn cmap}\} \\
\text{concat . map } f . \text{ concat . map } g &= \{\text{map after concat}\} \\
\text{concat . concat . map } (\text{map } f) . \text{ map } g &= \{\text{map functor}\} \\
\text{concat . concat . map } (\text{map } f . g) &= \{\text{concat after concat}\} \\
\text{concat . map concat . map } (\text{map } f . g) &= \{\text{map functor}\} \\
\text{concat . map } (\text{concat . map } f . g) &= \{\text{defn cmap}\} \\
\text{concat . map } (\text{cmap } f . g) &= \{\text{defn cmap}\} \\
\text{cmap } (\text{cmap } f . g)
\end{align*}
\]

Answer to Exercise L

The human proof is:

\[
\begin{align*}
\text{cmap } (\text{cpp . (one } \ast g)) . \text{ cpp} &= \{\text{cmap after cmap (backwards)}\} \\
\text{cmap } \text{cpp . map } (\text{one } \ast g) . \text{ cpp} &= \{\text{map after cpp}\} \\
\text{cmap } \text{cpp . cpp . (map one } \ast \text{ map } g) &= \{\text{map after cpp}\} \\
\text{cpp . (concat } \ast \text{ concat) . (map one } \ast \text{ map } g) &= \{\text{cross bifunctor}\} \\
\text{cpp . ((concat . map one) } \ast \text{ concat (map } g)) &= \{\text{defn cmap (backwards)}\} \\
\text{cpp . ((concat . map one) } \ast \text{ cmap } g) &= \{\text{concat after id}\} \\
\text{cpp . (id } \ast \text{ cmap } g)
\end{align*}
\]

No, the calculation cannot be performed automatically. The cmap after cmap
law cannot be installed in the backwards direction without causing the calculator
to loop (see Exercise B).

12.11 Chapter notes

The calculator in this chapter is based on an undocumented theorem prover by
Mike Spivey, a colleague at Oxford. Ross Paterson of City University, London, has
produced a version with built-in functor laws that can be applied in both directions
when necessary.

One state-of-the-art proof assistant is Coq; see http://coq.inria.fr/.
Index

\[ \Theta \] asymptotic notation, 157
\[ [\cdot] \] floor function, 46, 52, 62
\[ \bot \] undefined value, 24, 28, 30, 104, 177, 215, 217, 226, 233
\[ \pi \] 50, 215
\[ \sqsubseteq \] approximation ordering, 217
\[ (\%) \] integral ratio, 50
\[ " " \] empty list of characters, 18
\[ " \] double quotes, 4
\[ ' \] single quote, 3
\[ (!) \] list-indexing operation, 9
\[ (!) \] array-indexing operation, 261
\[ ($!) \] strict application operator, 153, 252
\[ ($) \] application operator, 153
\[ (&&) \] boolean conjunction, 10
\[ () \] null tuple, 30, 33, 240
\[ (**) \] exponentiation, 59
\[ (,) \] pair constructor, 74, 146
\[ (-) \] subtraction operator, 27
\[ (->) \] function type, 1
\[ (.) \] function composition, 3, 15, 31
\[ (//) \] array update operation, 262
\[ (\neq) \] inequality test, 9
\[ (:\:) \] cons, list constructor, 64
\[ (\langle\langle) \] comparison test, 9, 76
\[ (\langle\langle\langle) \] right-to-left Kleisli composition, 247
\[ (==) \] equality test, 9
\[ (\gg\gg) \] left-to-right Kleisli composition, 247
\[ (\gg) \] sequence operation, 240
\[ (\gg\gg) \] monadic bind, 241, 279
\[ (^) \] exponentiation, 59
\[ (^^) \] exponentiation, 59
\[ (|\mid) \] boolean disjunction, 32
\[ .hs \] Haskell script, 35, 227
\[ .lhs \] literate Haskell script, 35, 227
\[ :load \] instruction, 13
\[ :set \] instruction, 13, 148
\[ :type \] instruction, 22
\[ @ \] as patterns, see patterns
\[ [\square] \] empty list, 17, 63, 64, 104, 121, 151
\[ \backslash \] escape character, 18
\[ \newline \] newline character, 3, 18
\[ \tab \] tab character, 18
\[ ` \] back-quote, 9, 25
\[ (++) \] list concatenation, 10, 15, 69, 113
\[ n+k \] patterns, 111
\[ abs \] 50, 54
\[ absolute value, see abs \] abstract data types, 194, 239, 259
abstract syntax trees, 201
accumArray, 260
accumulating functions, 260
accumulating parameters, 159, 171, 288, 289, 323
actions, 239
Agda, 48
Algorithm Design, 145, 154
all, 94
alphabetical order, 5, 19
anagrams, 16
and, 74
anti-symmetric relation, 217
any, 102
approx, 218
Array, 259
array, 259
arrays

immutable, 259
mutable, 254
associative operations, 15, 26, 70, 112, 113, 119, 124, 129, 184, 231, 246, 247, 281, 326
assocs, 262
asymptotic complexity, 155
Augustsson, L., 47
auxiliary results, 114, 117, 140
base cases, see induction
Bentley, J., 21, 127, 144
Bifunctor, 82
bifunctors, 82, 87, 302, 326
binary operators, 25
binary search, 54
binary trees, 165, 249
Index

binding power, 2, 14, 25
Bird, R., 87, 180
blank characters, 3
BNF (Backus-Naur form), 286, 305
Bool, 10, 30
boolean
  conjunction, see (&&)
  disjunction, see (||)
bottom, see ⊥ undefined value
braces, 15, 36, 242
brackets, 15
breadth-first search, 257
break, 102
C, 239
C#, x
case expression, see expressions
case analysis, 10, 53
Category Theory, 71, 87
chain completeness, 116, 212, 220
chain of approximations, 215, 218, 225
Char, 3, 30, 90, 254
Chitil, O., 209
comment convention, 8
common subexpression elimination, 147
commonWords, 3, 4, 34, 75
comparison operations, 32
compiled functions, 154
compilers, 36, 154
complete partial orderings, 218
Complex, 49
cat, 6, 67, 70, 157
catMap, 307
cartesian product, 92, 93, 97, 131, 155, 244, 320
cross, 81, 301
curry, 86, 135
Curry, H.B, 87
cycle, 232
cyclic lists, 212
data constructors, 25, 56
data type declarations, 30, 56, 189, 194, 202, 229
Data.Array, 259
Data.Char, 13, 42, 227, 283
Data.Complex, 49
Data.List, 75, 106, 125, 154, 311
Data.Maybe, 319
Data.STRef, 251
de Moor, O., 87
depending definitions, 31
dependently-typed languages, 48, 90
deriving clauses, 39, 57
directed graphs, 260
distributive operations, 130
div, 9, 24, 59
divMod, 51
do-notation, 34, 36, 239, 242, 245
done, 240
Double, 49
drop, 79
dropWhile, 106
e, 141
echoing, 241
efficiency, ix, 145
Either, 82, 132
either, 132
else, 12
embedded domain-specific language, 209
empty list, see []
Enum, 65, 90, 211
enumerations, 65, 210, 217
enumFrom, 211
Eq, 31, 56
equality operations, 31
equational reasoning, ix, 1, 73, 81, 89, 96, 99, 110, 135, 298
error, 36, 39, 179
error messages, 23, 24, 39, 43, 103, 151, 192, 250, 279
evaluation
  eager, 28, 154, 157
  innermost, 28
  lazy, 28, 75, 80, 89, 145, 154, 175, 198, 243
  outermost, 28
to normal form, 22, 27, 146, 156
exception handling, 239
exp, 40, 110, 141
explicit layout, 242
exponentiation, 40, 59, 110
export declarations, see modules
expressions
  case, 127, 245
  conditional, 12, 23, 181
  lambda, 26, 148, 242
  let, 24, 146, 147, 248
  well-formed, 22
factorial function, 28, 220
factoring parsers, 282
failure, 39
Feijen, W., xi
Fibonacci function, 164, 232, 251, 266
FilePath, 34
Index

filter, 38, 67, 70, 72, 97, 98, 119, 134, 299
flat ordering, 217
flip, 59, 123, 148
Float, 2, 30, 49
Floating, 52
floating-point literal, 51
floating-point numbers, 112, 293
floor, 46, 52, 60
fmap, 71
foldl, 122, 150, 152, 260
foldl', 150, 152
foldl1, 231
foldr, 117, 152, 164
foldr1, 121, 231
forall, 253
fork, 81, 134, 301
Fractional, 50
fromInteger, 50, 67, 151
fromIntegral, 51, 67, 151
fromJust, 319
fst, 28, 51
function, 1
application, 2, 4, 14
arguments, 1, 3
composition, see (.)
computable, 219
continuous, 219
conversion, 50
higher-order, 110
identity, see id
monotonic, 219
non-strict, 29, 58
overloaded, 31
partial, 9
polymorphic, 31, 72
primitive, 150
recursive, 17, 219
results, 1, 3
strict, 29, 59, 72, 120, 137, 150, 154
type, see ->
values, 145
Functor, 71, 264
functors, 87, 264
getChar, 240
getLine, 241
GHC, 35, 36, 154, 169, 275
GHCi, 12, 22, 36, 154
Gibbons, J., 209, 275
global definition, see top-level definition
Goerzen, J., 21
Gofer, x, 275
golden ratio, 164
Graham, R., 62
grammars, 286
greedy algorithms, 192
guard, 306
guarded equations, 9, 12, 332
guards, 10

Hamming, W.R., 232
Hangman, 266
Hardy, G.H., 78
Harper, B., 47
hash tables, 256
Haskell, x, 1
1998 online report, 21
2010 online report, 21, 111
commands, 1, 33, 34, 239
layout, 36
libraries, 7, 181
numbers, 2, 112
Platform, 12, 154
reserved words, 12
standard prelude, see standard prelude
syntax, 22
values, 22
well-formed literals, 122
head, 38, 68, 72
head normal form, 146, 157, 172
helper functions, 26
Hinze, R., 275
History of Haskell, 20
homomorphisms, see laws
Hughes, J., 209
Hutton, G., 21, 297
id, 18, 70, 71, 94, 96, 253
idempotence, 205, see laws
identities, see laws
identity elements, 15, 96, 184, 247, 281
identity function, see id
if, 12
imperative languages, x
import declarations, see modules
in-place algorithms, 170, 254
indexitis, 95
induction, 110, 111
base case, 77, 111
general, 179
inductive case, 77, 111
over lists, 113
over numbers, 110
pre-packaged, 120
inductive cases, see induction
infinite loops, 24, 28, 30, 77, 124, 211
infix data constructors, 194
infix operations, 9
infixr fixity declaration, 153
inits, 125
inlining, 54
insertion sort, see sorting
instance declarations, 32
Int, 2, 30, 49, 155, 254
Integer, 2, 49, 155
integer literals, 50
Integral, 51, 155
interact, 227
interaction, 221
Index

interactive programs, 227
interpreters, 36
involutions, see laws
IO, 33, 239
isAlpha, 42
isSpace, 283
it, 229
iterate, 64, 213, 301
Ix, 254, 255
Jeuring, J., 209
join, 264
Jones, M., 144, 275
Just, 244
Knuth, D.E., 21, 62, 167
KRC, x
Lapalme, G., 180
last, 68
Launchbury, J., 275
laws
  arithmetic, 17
  bifunctor, 302
  commutative, 186
  distributive, 186
  equational, ix
  functor, 71, 81, 303, 330
  fusion, 120, 131, 176
  homomorphisms, 185, 186
  idempotence, 204
  identities, ix, 110
  involutions, 96, 113
  leapfrog, 247, 265
  left-distributive, 293
  left-zero, 293
  monad laws, 246, 279
  naturality, 72, 87, 97
  point-free, 110
  right-zero, 293
  trigonometric, ix
  tupling, 152, 165
layout description language, 182
lazy evaluation, see evaluation
least fixed points, 220
least upper bounds, 218
left-recursion problem, 287
Leibniz, G., 243
length, 69, 79, 160
let, see expressions
lexicographic order, 76, 189, 217
liftM, 264
lines, 188, 200
Linux, 12
list
  adjacency, 261
  comprehensions, 66, 92, 156, 244, 245, 248
  concatenation, see (++)
  cyclic, 210
  doubly-linked, 228
  finite, 64
  identity function, 118
  indexing, see (!!)
  infinite, 53, 64, 75, 108, 210
  notation, 3
  partial, 64, 115, 211
listArray, 260
lists
  adjacency, 261
  literate programming, 8
  local definitions, 11, 149, 256
  log, 141
  logarithmic factors, 159
  logarithmic time, 56
  logBase, 2, 141
  lookup, 244, 319
  loop invariants, 255, 263
  lower bounds, 157
  lowercase, 5
  Loyd, S., 267
Mac, 12
main, 34, 227
map, 5, 23, 31, 38, 67, 70, 80, 91, 119, 299
mapM, 265
mapM_, 265
Marlow, S., 20
Maslanka, C., 48
mathematical operators, 217
matrices, 90, 94, 234
matrix
  addition, 105
  multiplication, 105
  transpose, 94, 106
maximum, 122
Maybe, 39, 244
McIlory, D., 21
mean, 151
Meijer, E., 209, 297
merge, 76, 211, 231
mergesort, see sorting
minimum, 104, 107, 122, 157, 211
Miranda, x
mkStdGen, 223
ML, 29, 48
mod, 9
modules, 13, 25, 35, 309
  export declarations, 35, 199
  hierarchical names, 21
  import declarations, 13, 35
Monad, 243, 264
monadic programming, 239
MonadPlus, 292
monads, 243
  commutative, 264
  monoids, 247
  mplus, 292
mutable structures, 248
mzero, 292
Index

Nat, 56, 110, 132
natural transformations, 87
\negate, 50
newline character, see \n
\texttt{newSTRef}, 251
Newton's method, 60
\texttt{newtype} declarations, 278
non-decreasing order, 74
\texttt{none}, 134
normal form, 146
\texttt{not}, 30
\texttt{notElem}, 98
\texttt{Nothing}, 244
\texttt{nub}, 106, 108
\texttt{null}, 39, 68
null tuple, see ()
\texttt{Num}, 23, 31, 49, 56
Number Theory, 219
numbers
  complex, see \texttt{Complex}
  floating point, see \texttt{Float}, \texttt{Double}
  floating-point, 60
  integer, see \texttt{Int}, \texttt{Integer}
  limited precision integers, see \texttt{Int}
  natural, 56, 110
  unlimited precision integers, see \texttt{Integer}
\texttt{O'Neill}, M., 237
\texttt{O'Sullivan}, B., 21
offside rule, 36, 242
\texttt{one}, 134
\texttt{Oppen}, D., 209
\texttt{or}, 102
\texttt{Ord}, 32
order of association, 2–4, 17, 25, 62, 196
\texttt{Orwell}, x
otherwise, 11
pairs, 2, 74, 76, 82, 177
palindromes, 41
paper–rock–scissors, 221
paragraphs, 191
parentheses, 15
parsers, 276
parsing, 239
partial application, 87
partial numbers, 58
\texttt{partition}, 311
\texttt{Patashnik}}, O., 62
\texttt{Paterson}}, R., 337
patterns
  n+k, 111
  as patterns, 77, 83
  disjoint, 68
don’t care, 67, 73, 103, 268
exhaustive, 68
irrefutable, 231
matching, 57, 67, 68, 74, 115, 194, 332
wildcard, 67, 268
perfect numbers, 65
\texttt{Perlis}}, A., 145
persistent data structures, 254
\texttt{Peyton Jones}}, S., 21, 209
\texttt{Pierce}}, B., 87
plumbing combinators, 86
point-free calculations, 86, 330
point-free reasoning, 298
pointers, 146
postconditions, 255, 263
precedence, 14, 25, 37
preconditions, 255, 263
prefix names, 25
prefix operators, 50
\texttt{Prelude}, 25
primes, 148, 213, 219, 220, 233
printing values, 33
profiling tools, 154
program variables, 251
programs, 7
prompt symbol, 13
prompts, 229, 240
proof format, xi, 112
\texttt{properFraction}, 56
\texttt{putChar}, 240
\texttt{putStrLn}, 16, 33, 239, 240
Pythagorean triads, 66
Python, x, 239, 252
qualified names, 306
quicksort, see sorting
\texttt{Rabbi}}, F., 180
\texttt{Ramanujan}}, S., 78
random numbers, 223, 250
\texttt{randomR}, 223
rank 2 polymorphic types, 253
\texttt{Rational}, 49, 50
\texttt{Read}, 41, 122, 277
\texttt{read}, 52, 122
\texttt{readFile}, 34
reading files, 34
\texttt{ReadS}, 277
\texttt{reads}, 277
\texttt{readSTRef}, 251
\texttt{Real}, 50
recursive definitions, 29, 77, 219
reduction, see evaluation
reduction steps, 155, 156
reference variables, 251
reflexive relation, 217
\texttt{repeat}, 108, 212
\texttt{return}, 241
\texttt{reverse}, 42, 72, 113, 117, 123, 159
running sums, 125
\texttt{runST}, 252
SASL, x
\texttt{scanl}, 125, 127
\texttt{scanr}, 130
scientific notation, 60, 112
scope, 11
scripts, 7, 25, 148
sections, 26, 52, 53
segments, 127, 316
select, 175
selection sort, see sorting
semicolon, 36
separator characters, 200
seq, 150, 153, 226
sequence, 264
set theory, 211
shallow embeddings, 187, 192
shared values, 146
Show, 32, 40, 56, 229, 276, 291
ShowS, 289, 311
showsPrec, 290, 291, 308
side-effects, 243
sieve of Sundaram, 233
signum, 50
Sijtsma, B., 237
sin, 2, 14
sine function, ix, xi, 2, 14
size measures, 156, 202
snd, 51
sort, 14, 75, 154
sorting, 5, 6, 94, 167
insertion sort, 172
mergesort, 76, 168
numbers, 262
quicksort, 169, 254, 263
selection sort, 172
space character, 3
space efficiency, 29, 84, 147, 149, 154, 171
space leaks, 147, 151, 170, 171
span, 75, 102
Spivey, M., 337
splitAt, 80, 168
sqrt, 60
stand-alone programs, 34, 227
standard prelude, 13, 30, 39, 42, 51, 56, 73, 80, 86, 87, 127, 150, 161, 168, 213, 232
STArray, 254
state monad, 247
state threads, 251
state-thread monad, 251
Stewart, D., 21
stream-based interaction, 226
STRef, 251
strictness flags, 58, 194
String, 6
strings, 4
subclasses, 50
subseqs, 133, 146, 158
subsequences, 146
subtract, 53, 59
Sudoku, 89, 258, 321
sum, 150
Sundaram, S.P., 233
superclasses, 32
Swierstra, D., 209
syntactic categories, 286
System.Random, 223
System.IO, 242
tail, 38, 68
tails, 128
take, 6, 30, 79, 218
takeWhile, 52, 106
taxicab numbers, 79, 88
terminator characters, 200
texts, 3, 5, 181
then, 12
time efficiency, 92, 123, 126, 147, 154, 156, 182
toInteger, 51, 151
toLower, 5, 7, 13, 38
top-level definitions, 147, 256
toRational, 50
toruses, 234
toUpper, 44, 227
transitive relation, 217
transpose, 106
trees, 71, 161, 165, 190, 194
trigonometry, ix, 2
tupling, 164, 170
tupling law of foldr, 165
tyling the recursive knot, 214, 238
type
annotations, 41
classes, 25, 30, 31, 155
declarations, 7, 20
inference, 20, 22
signatures, 6, 20, 30, 155
synonyms, 5, 90, 183, 278
variables, 7, 31
well-formed, 22
types
compound, 30
isomorphic, 278
polymorphic, 31, 72
primitive, 30

UHC (Utrecht Haskell Compiler), 47
uncurry, 86, 135
undefined, 24, 64
underscore convention, 264
unlines, 227
unsafePerformIO, 242
until, 52, 65
unwords, 44
unzip, 81
upper bounds, 218
visible characters, 3
Visual Basic, x
Wadler, P., 87, 209
where clauses, 11, 24, 36, 147, 242
white space, 283
wholemeal programming, 95
Wilde, O., 145
Windows, 12
WinGHCi, 12
words, 4, 5, 38, 107, 191
World, 239
writeFile, 34
writeSTRef, 251
writing files, 34
zip, 73
zipWith, 73, 105
zipWith3, 230