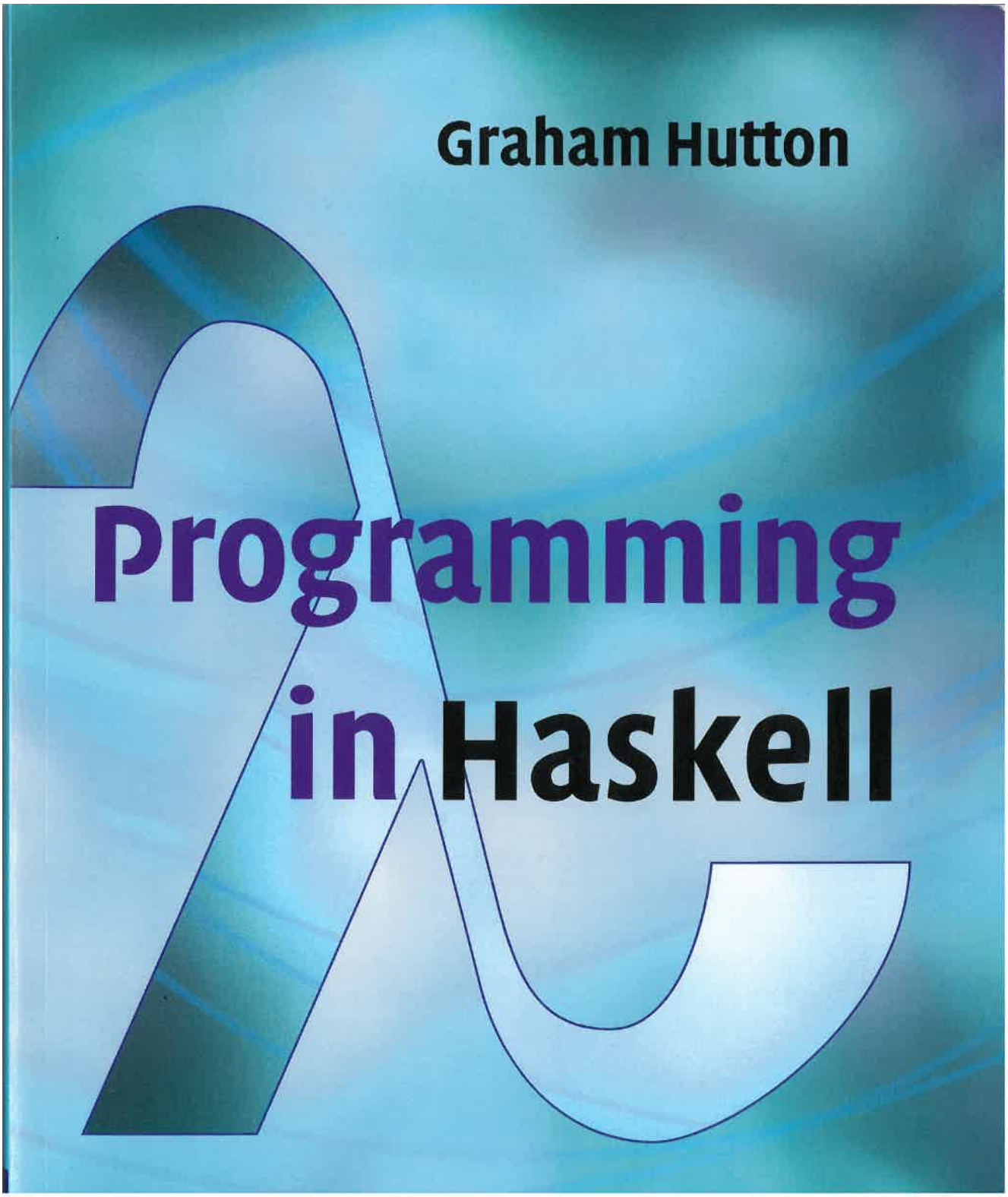
Graham Hutton
Programming
in Haskell

Contents

Foreword

It is nearly a century ago that Alonzo Church introduced the lambda calculus, and over half a century ago that John McCarthy introduced Lisp, the world's second oldest programming language and the first functional language based on the lambda calculus. By now, every major programming language including JavaScript, C++, Swift, Python, PHP, Visual Basic, Java, ... has support for lambda expressions or anonymous higher-order functions.

As with any idea that becomes mainstream, inevitably the underlying foundations and principles get watered down or forgotten. Lisp allowed mutation, yet today many confuse functions as first-class citizens with immutability. At the same time, other effects such as exceptions, reflection, communication with the outside world, and concurrency go unmentioned. Adding recursion in the form of feedback-loops to pure combinational circuits lets us implement mutable state via flip-flops. Similarly, using one effect such as concurrency or input/output we can simulate other effects such as mutability. John Hughes famously stated in his classic paper *Why Functional Programming Matters* that we cannot make a language more powerful by eliminating features. To that, we add that often we cannot even make a language less powerful by removing features. In this book, Graham demonstrates convincingly that the true value of functional programming lies in leveraging first-class functions to achieve compositionality and equational reasoning. Or in Graham's own words, "functional programming can be viewed as a style of programming in which the basic method of computation is the application of functions to arguments". These functions do not necessarily have to be pure or statically typed in order to realise the simplicity, elegance, and conciseness of expression that we get from the functional style.

While you can code like a functional hacker in a plethora of languages, a semantically pure and lazy, and syntactically lean and terse language such as Haskell is still the best way to learn how to think like a fundamentalist. Based upon decades of teaching experience, and backed by an impressive stream of research papers, in this book Graham gently guides us through the whole gambit of key functional programming concepts such as higher-order functions, recursion, list comprehensions, algebraic datatypes and pattern matching. The book does not shy away from more advanced concepts. If you are still confused by the n-th blog post that attempts to explain monads, you are in the right place. Gently starting with the IO monad, Graham progresses from functors to applicatives using many concrete examples. By the time he arrives at monads, every reader will feel that they themselves could have come up with the concept of a monad as a generic pattern for composing functions with effects. The chapter on monadic

parsers brings everything together in a compelling use-case of parsing arithmetic expressions in the implementation of a simple calculator.

This new edition not only adds many more concrete examples of concepts introduced throughout the book, it also introduces the novel Haskell concepts of foldable and traversable types. Readers familiar with object-oriented languages routinely use iterables and visitors to enumerate over all values in a container, or respectively to traverse complex data structures. Haskell's higher-kinded type classes allow for a very concise and abstract treatment of these concepts by means of the Foldable and Traversable classes. Last but not least, the final chapters of the book give an in-depth overview of lazy evaluation and equational reasoning to prove and derive programs. The capstone chapter on calculating compilers especially appeals to me because it touches a topic that has had my keen interest for many decades, ever since my own PhD thesis on the same topic.

While there are plenty of alternative textbooks on Haskell in particular and functional programming in general, Graham's book is unique amongst all of these in that it uses Haskell simply as a tool for thought, and never attempts to sell Haskell or functional programming as a silver bullet that magically solves all programming problems. It focuses on elegant and concise expression of intent and thus makes a strong case of how pure and lazy functional programming is an intelligible medium for efficiently reasoning about algorithms at a high level of abstraction. The skills you acquire by studying this book will make you a much better programmer no matter what language you use to actually program in. In the past decade, using the first edition of this book I have taught many tens of thousands of students how to juggle with code. With this new edition, I am looking forward to extending this streak for at least another 10 years.

Erik Meijer

Preface

What is this book?

Haskell is a purely functional language that allows programmers to rapidly develop software that is clear, concise and correct. The book is aimed at a broad spectrum of readers who are interested in learning the language, including professional programmers, university students and high-school students. However, no programming experience is required or assumed, and all concepts are explained from first principles with the aid of carefully chosen examples and exercises. Most of the material in the book should be accessible to anyone over the age of around sixteen with a reasonable aptitude for scientific ideas.

How is it structured?

The book is divided into two parts. Part I introduces the basic concepts of pure programming in Haskell and is structured around the core features of the language, such as types, functions, list comprehensions, recursion and higher-order functions. Part II covers impure programming and a range of more advanced topics, such as monads, parsing, foldable types, lazy evaluation and reasoning about programs. The book contains many extended programming examples, and each chapter includes suggestions for further reading and a series of exercises. The appendices provide solutions to selected exercises, and a summary of some of the most commonly used definitions from the Haskell standard prelude.

What is its approach?

The book aims to teach the key concepts of Haskell in a clean and simple manner. As this is a textbook rather than a reference manual we do not attempt to cover all aspects of the language and its libraries, and we sometimes choose to define functions from first principles rather than using library functions. As the book progresses the level of generality that is used is gradually increased. For example, in the beginning most of the functions that are used are specialised to simple types, and later on we see how many functions can be generalised to larger classes of types by exploiting particular features of Haskell.

How should it be read?

The basic material in part I can potentially be worked through fairly quickly, particularly for those with some prior programming experience, but additional time and effort may be required to absorb some of material in part II. Readers are recommended to work through all the material in part I, and then select

appropriate material from part II depending on their own interests. It is vital to write Haskell code for yourself as you go along, as you can't learn to program just by reading. Try out the examples from each chapter as you proceed, and solve the exercises for each chapter before checking the solutions.

What's new in this edition?

The book is an extensively revised and expanded version of the first edition. It has been extended with new chapters that cover more advanced aspects of Haskell, new examples and exercises to further reinforce the concepts being introduced, and solutions to selected exercises. The remaining material has been completely reworked in response to changes in the language and feedback from readers. The new edition uses the Glasgow Haskell Compiler (GHC), and is fully compatible with the latest version of the language, including recent changes concerning applicative, monadic, foldable and traversable types.

How can it be used for teaching?

An introductory course might cover all of part I and a few selected topics from part II; my first-year course covers chapters 1–9, 10 and 15. An advanced course might start with a refresher of part I, and cover a selection of more advanced topics from part II; my second-year course focuses on chapters 12 and 16, and is taught interactively on the board. The website for the book provides a range of supporting materials, including PowerPoint slides and Haskell code for the extended examples. Instructors can obtain a large collection of exams and solutions based on material in the book from `solutions@cambridge.org`.

Acknowledgements

I am grateful to the University of Nottingham for providing a sabbatical to produce this new edition; Thorsten Altenkirch, Venanzio Capretta, Henrik Nilsson and other members of the FP lab for our many enjoyable discussions; Iván Pérez Domínguez for useful comments on a number of chapters; the students and tutors on all of my Haskell courses for their feedback; Clare Dennison, David Tranah and Abigail Walkington at CUP for their editorial work; the GHC team for producing such a great compiler; and finally, Catherine and Ian Hutton for getting me started in computing all those years ago.

Many thanks also to Ki Yung Ahn, Bob Davison, Philip Hölzenspies and Neil Mitchell for providing detailed comments on the first edition, and to the following for pointing our errors and typos: Paul Brown, Sergio Queiroz de Medeiros, David Duke, Robert Fabian, Ben Fleis, Robert Furber, Andrew Kish, Tomoyas Kobayashi, Florian Larysch, Carlos Oroz, Douglas Philips, Bruce Turner, Gregor Ulm, Marco Valtorta and Kazu Yamamoto. All of these comments have been taken into account when preparing the new edition.

Graham Hutton

Part I

Basic Concepts

1 Introduction

In this chapter we set the stage for the rest of the book. We start by reviewing the notion of a function, then introduce the concept of functional programming, summarise the main features of Haskell and its historical background, and conclude with three small examples that give a taste of Haskell.

1.1 Functions

In Haskell, a *function* is a mapping that takes one or more arguments and produces a single result, and is defined using an equation that gives a name for the function, a name for each of its arguments, and a body that specifies how the result can be calculated in terms of the arguments.

For example, a function `double` that takes a number `x` as its argument, and produces the result `x + x`, can be defined by the following equation:

```
double x = x + x
```

When a function is applied to actual arguments, the result is obtained by substituting these arguments into the body of the function in place of the argument names. This process may immediately produce a result that cannot be further simplified, such as a number. More commonly, however, the result will be an expression containing other function applications, which must then be processed in the same way to produce the final result.

For example, the result of the application `double 3` of the function `double` to the number `3` can be determined by the following calculation, in which each step is explained by a short comment in curly parentheses:

```
    double 3
=       { applying double }
    3 + 3
=       { applying + }
    6
```

Similarly, the result of the nested application `double (double 2)` in which the function `double` is applied twice can be calculated as follows:

```
    double (double 2)
```

```
    =    { applying the inner double }
  double (2 + 2)
    =    { applying + }
  double 4
    =    { applying double }
  4 + 4
    =    { applying + }
  8
```

Alternatively, the same result can also be calculated by starting with the outer application of the function `double` rather than the inner:

```
  double (double 2)
    =    { applying the outer double }
  double 2 + double 2
    =    { applying the first double }
  (2 + 2) + double 2
    =    { applying the first + }
  4 + double 2
    =    { applying double }
  4 + (2 + 2)
    =    { applying the second + }
  4 + 4
    =    { applying + }
  8
```

However, this approach requires two more steps than our original version, because the expression `double 2` is duplicated in the first step and hence simplified twice. In general, the order in which functions are applied in a calculation does not affect the value of the final result, but it may affect the number of steps required, and whether the calculation process terminates. These issues are explored in more detail when we consider how expressions are evaluated in chapter 15.

1.2 Functional programming

What is functional programming? Opinions differ, and it is difficult to give a precise definition. Generally speaking, however, functional programming can be viewed as a *style* of programming in which the basic method of computation is the application of functions to arguments. In turn, a functional programming language is one that *supports* and *encourages* the functional style.

To illustrate these ideas, let us consider the task of computing the sum of the integers (whole numbers) between one and some larger number `n`. In many current programming languages, this would normally be achieved using two integer variables whose values can be changed over time by means of the assignment

operator =, with one such variable used to accumulate the total, and the other used to count from 1 to n. For example, in Java the following program computes the required sum using this approach:

```
int total = 0;
for (int count = 1; count <= n; count++)
   total = total + count;
```

That is, we first initialise an integer variable total to zero, and then enter a loop that ranges an integer variable count from 1 to n, adding the current value of the counter to the total each time round the loop.

In the above program, the basic method of computation is *changing stored values*, in the sense that executing the program results in a sequence of assignments. For example, the case of n = 5 gives the following sequence, in which the final value assigned to the variable total is the required sum:

```
total = 0;
count = 1;
total = 1;
count = 2;
total = 3;
count = 3;
total = 6;
count = 4;
total = 10;
count = 5;
total = 15;
```

In general, programming languages such as Java in which the basic method of computation is changing stored values are called *imperative* languages, because programs in such languages are constructed from imperative instructions that specify precisely how the computation should proceed.

Now let us consider computing the sum of the numbers between one and n using Haskell. This would normally be achieved using two library functions, one called [..] that is used to produce the list of numbers between 1 and n, and the other called sum that is used to produce the sum of this list:

```
sum [1..n]
```

In this program, the basic method of computation is *applying functions to arguments*, in the sense that executing the program results in a sequence of applications. For example, the case of n = 5 gives the following sequence, in which the final value in the sequence is the required sum:

```
   sum [1..5]
=      { applying [..] }
   sum [1,2,3,4,5]
```

```
=      { applying sum }
  1 + 2 + 3 + 4 + 5
=      { applying + }
  15
```

Most imperative languages provide some form of support for programming with functions, so the Haskell program `sum [1..n]` could be translated into such languages. However, many imperative languages do not *encourage* programming in the functional style. For example, many such languages discourage or prohibit functions from being stored in data structures such as lists, from constructing intermediate structures such as the list of numbers in the above example, from taking functions as arguments or producing functions as results, or from being defined in terms of themselves. In contrast, Haskell imposes no such restrictions on how functions can be used, and provides a range of features to make programming with functions both simple and powerful.

1.3 Features of Haskell

For reference, the main features of Haskell are listed below, along with particular chapters of this book that give further details.

- **Concise programs** (chapters 2 and 4)

  Due to the high-level nature of the functional style, programs written in Haskell are often much more *concise* than programs written in other languages, as illustrated by the example in the previous section. Moreover, the syntax of Haskell has been designed with concise programs in mind, in particular by having few keywords, and by allowing indentation to be used to indicate the structure of programs. Although it is difficult to make an objective comparison, Haskell programs are often between two and ten times shorter than programs written in other languages.

- **Powerful type system** (chapters 3 and 8)

  Most modern programming languages include some form of *type system* to detect incompatibility errors, such as erroneously attempting to add a number and a character. Haskell has a type system that usually requires little type information from the programmer, but allows a large class of incompatibility errors in programs to be automatically detected prior to their execution, using a sophisticated process called type inference. The Haskell type system is also more powerful than most languages, supporting very general forms of *polymorphism* and *overloading*, and providing a wide range of special purpose features concerning types.

- **List comprehensions** (chapter 5)

  One of the most common ways to structure and manipulate data in computing is using lists of values. To this end, Haskell provides lists as a basic

concept in the language, together with a simple but powerful *comprehension* notation that constructs new lists by selecting and filtering elements from one or more existing lists. Using the comprehension notation allows many common functions on lists to be defined in a clear and concise manner, without the need for explicit recursion.

- **Recursive functions** (chapter 6)

  Most programs involve some form of looping. In Haskell, the basic mechanism by which looping is achieved is through *recursive* functions that are defined in terms of themselves. It can take some time to get used to recursion, particularly for those with experience of programming in other styles. But as we shall see, many computations have a simple and natural definition in terms of recursive functions, especially when *pattern matching* and *guards* are used to separate different cases into different equations.

- **Higher-order functions** (chapter 7)

  Haskell is a *higher-order* functional language, which means that functions can freely take functions as arguments and produce functions as results. Using higher-order functions allows common programming patterns, such as composing two functions, to be defined as functions within the language itself. More generally, higher-order functions can be used to define *domain-specific languages* within Haskell itself, such as for list processing, interactive programming, and parsing.

- **Effectful functions** (chapters 10 and 12)

  Functions in Haskell are pure functions that take all their inputs as arguments and produce all their outputs as results. However, many programs require some form of *side effect* that would appear to be at odds with purity, such as reading input from the keyboard, or writing output to the screen, while the program is running. Haskell provides a uniform framework for programming with effects, without compromising the purity of functions, based upon the use of *monads* and *applicatives*.

- **Generic functions** (chapters 12 and 14)

  Most languages allow functions to be defined that are *generic* over a range of simple types, such as different forms of numbers. However, the Haskell type system also supports functions that are generic over much richer kinds of structures. For example, the language provides a range of library functions that can be used with any type that is *functorial*, *applicative*, *monadic*, *foldable*, or *traversable*, and moreover, allows new structures and generic functions over them to be defined.

- **Lazy evaluation** (chapter 15)

  Haskell programs are executed using a technique called *lazy evaluation*, which is based upon the idea that no computation should be performed

until its result is actually required. As well as avoiding unnecessary computation, lazy evaluation ensures that programs terminate whenever possible, encourages programming in a modular style using intermediate data structures, and even allows programming with infinite structures.

- **Equational reasoning** (chapters 16 and 17)

  Because programs in Haskell are pure functions, simple *equational reasoning* techniques can be used to execute programs, to transform programs, to prove properties of programs, and even to calculate programs directly from specifications of their intended behaviour. Equational reasoning is particularly powerful when combined with the use of *induction* to reason about functions that are defined using recursion.

1.4 Historical background

Many of the features of Haskell are not new, but were first introduced by other languages. To help place Haskell in context, some of the key historical developments related to the language are briefly summarised below:

- In the 1930s, Alonzo Church developed the lambda calculus, a simple but powerful mathematical theory of functions.
- In the 1950s, John McCarthy developed Lisp ("LISt Processor"), generally regarded as being the first functional programming language. Lisp had some influences from the lambda calculus, but still retained the concept of variable assignment as a central feature of the language.
- In the 1960s, Peter Landin developed ISWIM ("If you See What I Mean"), the first pure functional programming language, based strongly on the lambda calculus and having no variable assignments.
- In the 1970s, John Backus developed FP ("Functional Programming"), a functional programming language that particularly emphasised the idea of higher-order functions and reasoning about programs.
- Also in the 1970s, Robin Milner and others developed ML ("Meta-Language"), the first of the modern functional programming languages, which introduced the idea of polymorphic types and type inference.
- In the 1970s and 1980s, David Turner developed a number of lazy functional programming languages, culminating in the commercially produced language Miranda (meaning "admirable").
- In 1987, an international committee of programming language researchers initiated the development of Haskell (named after the logician Haskell Curry), a standard lazy functional programming language.

- In the 1990s, Philip Wadler and others developed the concept of type classes to support overloading, and the use of monads to handle effects, two of the main innovative features of Haskell.
- In 2003, the Haskell committee published the Haskell Report, which defined a long-awaited stable version of the language.
- In 2010, a revised and updated of the Haskell Report was published. Since then the language has continued to evolve, in response to both new foundational developments and new practical experience.

It is worthy of note that three of the above individuals — McCarthy, Backus, and Milner — have each received the ACM Turing Award, which is generally regarded as being the computing equivalent of a Nobel prize.

1.5 A taste of Haskell

We conclude this chapter with three small examples that give a taste of programming in Haskell. The examples involve processing lists of values of different types, and illustrate different features of the language.

Summing numbers

Recall the function `sum` used earlier in this chapter, which produces the sum of a list of numbers. In Haskell, `sum` can be defined using two equations:

```
sum []     = 0
sum (n:ns) = n + sum ns
```

The first equation states that the sum of the empty list is zero, while the second states that the sum of any non-empty list comprising a first number `n` and a remaining list of numbers `ns` is given by adding `n` and the sum of `ns`. For example, the result of `sum [1,2,3]` can be calculated as follows:

```
   sum [1,2,3]
=     { applying sum }
   1 + sum [2,3]
=     { applying sum }
   1 + (2 + sum [3])
=     { applying sum }
   1 + (2 + (3 + sum []))
=     { applying sum }
   1 + (2 + (3 + 0))
=     { applying + }
   6
```

Note that even though the function `sum` is defined in terms of itself and is hence *recursive*, it does not loop forever. In particular, each application of `sum` reduces the length of the argument list by one, until the list eventually becomes empty, at which point the recursion stops and the additions are performed. Returning zero as the sum of the empty list is appropriate because zero is the *identity* for addition. That is, `0 + x = x` and `x + 0 = x` for any number `x`.

In Haskell, every function has a *type* that specifies the nature of its arguments and results, which is automatically inferred from the definition of the function. For example, the function `sum` defined above has the following type:

```
Num a => [a] -> a
```

This type states that for any type `a` of numbers, `sum` is a function that maps a list of such numbers to a single such number. Haskell supports many different types of numbers, including integers such as `123`, and floating-point numbers such as `3.14159`. Hence, for example, `sum` could be applied to a list of integers, as in the calculation above, or to a list of floating-point numbers.

Types provide useful information about the nature of functions, but, more importantly, their use allows many errors in programs to be automatically detected prior to executing the programs themselves. In particular, for every occurrence of function application in a program, a check is made that the type of the actual arguments is compatible with the type of the function itself. For example, attempting to apply the function `sum` to a list of characters would be reported as an error, because characters are not a type of numbers.

Sorting values

Now let us consider a more sophisticated function concerning lists, which illustrates a number of other aspects of Haskell. Suppose that we define a function called `qsort` by the following two equations:

```
qsort []     = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where
                  smaller = [a | a <- xs, a <= x]
                  larger  = [b | b <- xs, b > x]
```

In this definition, `++` is an operator that appends two lists together; for example, `[1,2,3] ++ [4,5] = [1,2,3,4,5]`. In turn, `where` is a keyword that introduces local definitions, in this case a list `smaller` comprising all elements `a` from the list `xs` that are less than or equal to `x`, together with a list `larger` comprising all elements `b` from `xs` that are greater than `x`. For example, if `x = 3` and `xs = [5,1,4,2]`, then `smaller = [1,2]` and `larger = [5,4]`.

What does `qsort` actually do? First of all, we note that it has no effect on lists with a single element, in the sense that `qsort [x] = [x]` for any `x`. It is easy to verify this property using a simple calculation:

```
    qsort [x]
=       { applying qsort }
    qsort [] ++ [x] ++ qsort []
=       { applying qsort }
    [] ++ [x] ++ []
=       { applying ++ }
    [x]
```

In turn, we now work through the application of `qsort` to an example list, using the above property to simplify the calculation:

```
    qsort [3,5,1,4,2]
=       { applying qsort }
    qsort [1,2] ++ [3] ++ qsort [5,4]
=       { applying qsort }
    (qsort [] ++ [1] ++ qsort [2]) ++ [3]
      ++ (qsort [4] ++ [5] ++ qsort [])
=       { applying qsort, above property }
    ([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])
=       { applying ++ }
    [1,2] ++ [3] ++ [4,5]
=       { applying ++ }
    [1,2,3,4,5]
```

In summary, `qsort` has sorted the example list into numerical order. More generally, this function produces a sorted version of any list of numbers. The first equation for `qsort` states that the empty list is already sorted, while the second states that any non-empty list can be sorted by inserting the first number between the two lists that result from sorting the remaining numbers that are `smaller` and `larger` than this number. This method of sorting is called *quicksort*, and is one of the best such methods known.

The above implementation of quicksort is an excellent example of the power of Haskell, being both clear and concise. Moreover, the function `qsort` is also more general than might be expected, being applicable not just with numbers, but with any type of ordered values. More precisely, the type

```
qsort :: Ord a => [a] -> [a]
```

states that, for any type `a` of ordered values, `qsort` is a function that maps between lists of such values. Haskell supports many different types of ordered values, including numbers, single characters such as `'a'`, and strings of characters such as `"abcde"`. Hence, for example, the function `qsort` could also be used to sort a list of characters, or a list of strings.

Sequencing actions

Our third and final example further emphasises the level of precision and generality that can be achieved in Haskell. Consider a function called `seqn` that takes a list of input/output actions, such as reading or writing a single character, performs each of these actions in sequence, and returns a list of resulting values. In Haskell, this function can be defined as follows:

```
seqn []         = return []
seqn (act:acts) = do x <- act
                     xs <- seqn acts
                     return (x:xs)
```

These two equations state that if the list of actions is empty we return the empty list of results, otherwise we perform the first action in the list, then perform the remaining actions, and finally return the list of results that were produced. For example, the expression `seqn [getChar,getChar,getChar]` reads three characters from the keyboard using the action `getChar` that reads a single character, and returns a list containing the three characters.

The interesting aspect of the function `seqn` is its type. One possible type that can inferred from the above definition is the following:

```
seqn :: [IO a] -> IO [a]
```

This type states that `seqn` maps a list of `IO` (input/output) actions that produce results of some type `a` to a single `IO` action that produces a list of such results, which captures the high-level behaviour of `seqn` in a clear and concise manner. More importantly, however, the type also makes explicit that the function `seqn` involves the *side effect* of performing input/output actions. Using types in this manner to keep a clear distinction between functions that are pure and those that involve side effects is a central aspect of Haskell, and brings important benefits in terms of both programming and reasoning.

In fact, the function `seqn` is more general than it may initially appear. In particular, the manner in which the function is defined is not specific to the case of input/output actions, but is equally valid for other forms of effects too. For example, it can also be used to sequence actions that may change stored values, fail to succeed, write to a log file, and so on. This flexibility is captured in Haskell by means of the following more general type:

```
seqn :: Monad m => [m a] -> m [a]
```

That is, for any *monadic* type `m`, of which `IO` is just one example, `seqn` maps a list of actions of type `m a` into a single action that returns a list of values of type `a`. Being able to define generic functions such as `seqn` that can be used with different kinds of effects is a key feature of Haskell.

1.6 Chapter remarks

The Haskell Report is freely available from `http://www.haskell.org`. More detailed historical accounts of the development of functional languages in general, and Haskell in particular, are given in [1] and [2].

1.7 Exercises

1. Give another possible calculation for the result of `double (double 2)`.

2. Show that `sum [x]` = `x` for any number `x`.

3. Define a function `product` that produces the product of a list of numbers, and show using your definition that `product [2,3,4]` = 24.

4. How should the definition of the function `qsort` be modified so that it produces a *reverse* sorted version of a list?

5. What would be the effect of replacing `<=` by `<` in the original definition of `qsort`? Hint: consider the example `qsort [2,2,3,1,1]`.

Solutions to exercises 1–3 are given in appendix A.

2 First steps

In this chapter we take our first proper steps with Haskell. We start by introducing the GHC system and the standard prelude, then explain the notation for function application, develop our first Haskell script, and conclude by discussing a number of syntactic conventions concerning scripts.

2.1 Glasgow Haskell Compiler

As we saw in the previous chapter, small Haskell programs can be executed by hand. In practice, however, we usually require a system that can execute programs automatically. In this book we use the *Glasgow Haskell Compiler*, a state-of-the-art, open source implementation of Haskell.

The system has two main components: a batch compiler called GHC, and an interactive interpreter called GHCi. We will primarily use the interpreter in this book, as its interactive nature makes it well suited for teaching and prototyping purposes, and its performance is sufficient for most of our applications. However, if greater performance or a stand-alone executable version of a Haskell program is required, the compiler itself can be used. For example, we will use the compiler in extended programming examples in chapters 9 and 11.

2.2 Installing and starting

The Glasgow Haskell Compiler is freely available for a range of operating systems from the Haskell home page, `http://www.haskell.org`. For first time users we recommend downloading the *Haskell Platform*, which provides a convenient means to install the system and a collection of commonly used libraries. More advanced users may prefer to install the system and libraries manually.

Once installed, the interactive GHCi system can be started from the terminal command prompt, such as `$`, by simply typing `ghci`:

```
$ ghci
```

All being well, a welcome message will then be displayed:

```
GHCi, version A.B.C: http://www.haskell.org/ghc/  :? for help
Prelude>
```

The GHCi prompt > indicates that the system is now waiting for the user to enter an expression to be evaluated. For example, it can be used as a calculator to evaluate simple numeric expressions:

```
> 2+3*4
14

> (2+3)*4
20

> sqrt (3^2 + 4^2)
5.0
```

Following normal mathematical convention, in Haskell exponentiation is assumed to have higher priority than multiplication and division, which in turn have higher priority than addition and subtraction. For example, `2*3^4` means `2*(3^4)`, while `2+3*4` means `2+(3*4)`. Moreover, exponentiation associates (or brackets) to the right, while the other four main arithmetic operators associate to the left. For example, `2^3^4` means `2^(3^4)`, while `2-3+4` means `(2-3)+4`. In practice, however, it is often clearer to use explicit parentheses in such expressions, rather than relying on the above rules.

2.3 Standard prelude

Haskell comes with a large number of built-in functions, which are defined in a library file called the *standard prelude*. In addition to familiar numeric functions such as + and *, the prelude also provides a range of useful functions that operate on lists. In Haskell, the elements of a list are enclosed in square parentheses and are separated by commas, as in `[1,2,3,4,5]`. Some of the most commonly used library functions on lists are illustrated below.

- Select the first element of a non-empty list:

  ```
  > head [1,2,3,4,5]
  1
  ```

- Remove the first element from a non-empty list:

  ```
  > tail [1,2,3,4,5]
  [2,3,4,5]
  ```

- Select the nth element of list (counting from zero):

```
> [1,2,3,4,5] !! 2
3
```

- Select the first `n` elements of a list:

```
> take 3 [1,2,3,4,5]
[1,2,3]
```

- Remove the first `n` elements from a list:

```
> drop 3 [1,2,3,4,5]
[4,5]
```

- Calculate the length of a list:

```
> length [1,2,3,4,5]
5
```

- Calculate the sum of a list of numbers:

```
> sum [1,2,3,4,5]
15
```

- Calculate the product of a list of numbers:

```
> product [1,2,3,4,5]
120
```

- Append two lists:

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
```

- Reverse a list:

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

As a useful reference guide, appendix B presents some of the most commonly used definitions from the standard prelude.

2.4 Function application

In mathematics, the application of a function to its arguments is usually denoted by enclosing the arguments in parentheses, while the multiplication of two values is often denoted silently, by writing the two values next to one another. For example, in mathematics the expression

$$f(a, b) + c\,d$$

means apply the function f to two arguments a and b, and add the result to the product of c and d. Reflecting its central status in the language, function application in Haskell is denoted silently using spacing, while the multiplication of two values is denoted explicitly using the operator `*`. For example, the expression above would be written in Haskell as follows:

```
f a b + c*d
```

Moreover, function application has higher priority than all other operators in the language. For example, `f a + b` means `(f a) + b` rather than `f (a + b)`. The following table gives a few further examples to illustrate the differences between function application in mathematics and in Haskell:

Mathematics	Haskell
$f(x)$	`f x`
$f(x, y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x, g(y))$	`f x (g y)`
$f(x)\,g(y)$	`f x * g y`

Note that parentheses are still required in the Haskell expression `f (g x)` above because `f g x` on its own would be interpreted as the application of the function `f` to two arguments `g` and `x`, whereas the intention is that `f` is applied to one argument, namely the result of applying the function `g` to an argument `x`. A similar remark holds for the expression `f x (g y)`.

2.5 Haskell scripts

As well as the functions provided in the standard prelude, it is also possible to define new functions. New functions are defined in a *script*, a text file comprising a sequence of definitions. By convention, Haskell scripts usually have a `.hs` suffix on their filename to differentiate them from other kinds of files. This is not mandatory, but is useful for identification purposes.

My first script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running GHCi. As an example, suppose that we start a text editor and type in the following two function definitions, and save the script to a file called `test.hs`:

```
double x = x + x

quadruple x = double (double x)
```

In turn, suppose that we leave the editor open, and in another window start up the GHCi system and instruct it to load the new script:

```
$ ghci test.hs
```

Now both the standard prelude and the script `test.hs` are loaded, and functions from both can be freely used. For example:

```
> quadruple 10
40

> take (double 2) [1,2,3,4,5]
[1,2,3,4]
```

Now suppose that we leave GHCi open, return to the editor, add the following two function definitions to those already typed in, and resave the file:

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

average ns = sum ns `div` length ns
```

We could also have defined `average ns = div (sum ns) (length ns)`, but writing `div` between its two arguments is more natural. In general, any function with two arguments can be written between its arguments by enclosing the name of the function in single back quotes ` `.

GHCi does not automatically reload scripts when they are modified, so a reload command must be executed before the new definitions can be used:

```
> :reload

> factorial 10
3628800

> average [1,2,3,4,5]
3
```

For reference, the table in figure 2.1 summarises the meaning of some of the most commonly used GHCi commands. Note that any command can be abbreviated by its first character. For example, `:load` can be abbreviated by `:l`. The command `:set editor` is used to set the text editor that is used by the system. For example, if you wish to use `vim` you would enter `:set editor vim`. The command `:type` is explained in more detail in the next chapter.

Naming requirements

When defining a new function, the names of the function and its arguments must begin with a lower-case letter, but can then be followed by zero or more letters

Command	Meaning
`:load name`	load script `name`
`:reload`	reload current script
`:set editor name`	set editor to `name`
`:edit name`	edit script `name`
`:edit`	edit current script
`:type expr`	show type of `expr`
`:?`	show all commands
`:quit`	quit GHCi

Figure 2.1 Useful GHCi commands

(both lower- and upper-case), digits, underscores, and forward single quotes. For example, the following are all valid names:

```
myFun   fun1   arg_2   x'
```

The following list of *keywords* have a special meaning in the language, and cannot be used as the names of functions or their arguments:

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

By convention, list arguments in Haskell usually have the suffix `s` on their name to indicate that they may contain multiple values. For example, a list of numbers might be named `ns`, a list of arbitrary values might be named `xs`, and a list of lists of characters might be named `css`.

The layout rule

Within a script, each definition at the same level must begin in precisely the same column. This *layout rule* makes it possible to determine the grouping of definitions from their indentation. For example, in the script

```
a = b + c
    where
      b = 1
      c = 2
d = a * 2
```

it is clear from the indentation that `b` and `c` are local definitions for use within the body of `a`. If desired, such grouping can be made explicit by enclosing a sequence of definitions in curly parentheses and separating each definition by a semi-colon. For example, the above script could also be written as

```
a = b + c
    where
       {b = 1;
        c = 2};
d = a * 2
```

or even be combined into a single line:

```
a = b + c where {b = 1; c = 2}; d = a * 2
```

In general, however, it is usually preferable to rely on the layout rule to determine the grouping of definitions, rather than using explicit syntax.

Tabs

Tab characters can cause problems in scripts, because layout is significant but different text editors interpret tabs in different ways. For this reason, it is recommended to avoid using tabs when indenting definitions, and the GHC system issues a warning message if they are used. If you do wish to use tabs in your scripts, it is best to configure your editor to automatically convert them to spaces. Haskell assumes that tab stops are 8 characters wide.

Comments

In addition to new definitions, scripts can also contain comments that will be ignored by the compiler. Haskell supports two kinds of comments, called *ordinary* and *nested*. Ordinary comments begin with the symbol -- and extend to the end of the current line, as in the following examples:

```
-- Factorial of a positive integer:
factorial n = product [1..n]

-- Average of a list of integers:
average ns = sum ns `div` length ns
```

Nested comments begin and end with the symbols {- and -}, may span multiple lines, and may be nested in the sense that comments can contain other comments. Nested comments are particularly useful for temporarily removing sections of definitions from a script, as in the following example:

```
{-
double x = x + x

quadruple x = double (double x)
-}
```

2.6 Chapter remarks

In addition to the GHC system, `http://www.haskell.org` contains a wide range of other useful resources concerning Haskell, including community activities, language documentation, and news items.

2.7 Exercises

1. Work through the examples from this chapter using GHCi.

2. Parenthesise the following numeric expressions:

```
2^3*4

2*3+4*5

2+3*4^5
```

3. The script below contains three syntactic errors. Correct these errors and then check that your script works properly using GHCi.

```
N =  a 'div' length xs
     where
         a = 10
        xs = [1,2,3,4,5]
```

4. The library function `last` selects the last element of a non-empty list; for example, `last [1,2,3,4,5]` = `5`. Show how the function `last` could be defined in terms of the other library functions introduced in this chapter. Can you think of another possible definition?

5. The library function `init` removes the last element from a non-empty list; for example, `init [1,2,3,4,5]` = `[1,2,3,4]`. Show how `init` could similarly be defined in two different ways.

Solutions to exercises 2–4 are given in appendix A.

3 Types and classes

In this chapter we introduce types and classes, two of the most fundamental concepts in Haskell. We start by explaining what types are and how they are used in Haskell, then present a number of basic types and ways to build larger types by combining smaller types, discuss function types in more detail, and conclude with the concepts of polymorphic types and type classes.

3.1 Basic concepts

A *type* is a collection of related values. For example, the type `Bool` contains the two logical values `False` and `True`, while the type `Bool -> Bool` contains all functions that map arguments from `Bool` to results from `Bool`, such as the logical negation function `not`. We use the notation `v :: T` to mean that `v` is a value in the type `T`, and say that `v` *has type* `T`. For example:

```
False :: Bool

True :: Bool

not :: Bool -> Bool
```

More generally, the symbol `::` can also be used with expressions that have not yet been evaluated, in which case the notation `e :: T` means that evaluation of the expression `e` will produce a value of type `T`. For example:

```
not False :: Bool

not True :: Bool

not (not False) :: Bool
```

In Haskell, every expression must have a type, which is calculated prior to evaluating the expression by a process called *type inference*. The key to this process is the following simple typing rule for function application, which states that if `f` is a function that maps arguments of type `A` to results of type `B`, and `e`

is an expression of type `A`, then the application `f e` has type `B`:

$$\frac{\texttt{f :: A -> B} \qquad \texttt{e :: A}}{\texttt{f e :: B}}$$

For example, the typing `not False :: Bool` can be inferred from this rule using the fact that `not :: Bool -> Bool` and `False :: Bool`. On the other hand, the expression `not 3` does not have a type under the above rule, because this would require that `3 :: Bool`, which is not valid because `3` is not a logical value. Expressions such as `not 3` that do not have a type are said to contain a *type error*, and are deemed to be invalid expressions.

Because type inference precedes evaluation, Haskell programs are *type safe*, in the sense that type errors can never occur during evaluation. In practice, type inference detects a very large class of program errors, and is one of the most useful features of Haskell. Note, however, that the use of type inference does not eliminate the possibility that other kinds of error may occur during evaluation. For example, the expression `1 'div' 0` is well-typed, but produces an error when evaluated because the result of division by zero is undefined.

The downside of type safety is that some expressions that evaluate successfully will be rejected on type grounds. For example, the conditional expression `if True then 1 else False` evaluates to the number `1`, but contains a type error and is hence deemed invalid. In particular, the typing rule for a conditional expression requires that both possible results have the same type, whereas in this case the first such result, `1`, is a number and the second, `False`, is a logical value. In practice, however, programmers quickly learn how to work within the limits of the type system and avoid such problems.

In GHCi, the type of any expression can be displayed by preceding the expression by the command `:type`. For example:

```
> :type not
not :: Bool -> Bool

> :type False
False :: Bool

> :type not False
not False :: Bool
```

3.2 Basic types

Haskell provides a number of basic types that are built-in to the language, of which the most commonly used are described below.

`Bool` – logical values

This type contains the two logical values `False` and `True`.

`Char` – single characters

This type contains all single characters in the Unicode system, the international standard for representing text-based information. For example, it contains all characters on a normal English keyboard, such as `'a'`, `'A'`, `'3'` and `'_'`, as well as a number of control characters that have a special effect, such as `'\n'` (move to a new line) and `'\t'` (move to the next tab stop). As in most programming languages, single characters must be enclosed in single forward quotes ' '.

`String` – strings of characters

This type contains all sequences of characters, such as `"abc"`, `"1+2=3"`, and the empty string `""`. Again, as is standard in most programming languages, strings of characters must be enclosed in double quotes " ".

`Int` – fixed-precision integers

This type contains integers such as `-100`, `0`, and `999`, with a fixed amount of memory being used for their storage. For example, the GHC system has values of type `Int` in the range -2^{63} to $2^{63} - 1$. Going outside this range can give unexpected results. For example, evaluating `2^63 :: Int` gives a negative number as the result, which is incorrect. (The use of `::` in this example forces the result to be an `Int` rather than some other numeric type.)

`Integer` – arbitrary-precision integers

This type contains all integers, with as much memory as necessary being used for their storage, thus avoiding the imposition of lower and upper limits on the range of numbers. For example, evaluating `2^63 :: Integer` using any Haskell system will produce the correct result.

Apart from the different memory requirements and precision for numbers of type `Int` and `Integer`, the choice between these two types is also one of performance. In particular, most computers have built-in hardware for fixed-precision integers, whereas arbitrary-precision integers are usually processed using the slower medium of software, as sequences of digits.

`Float` – single-precision floating-point numbers

This type contains numbers with a decimal point, such as `-12.34`, `1.0`, and `3.1415927`, with a fixed amount of memory being used for their storage. The

term *floating-point* comes from the fact that the number of digits permitted after the decimal point depends upon the size of the number. For example, evaluating `sqrt 2 :: Float` using GHCi gives the result `1.4142135` (the library function `sqrt` calculates the square root of a floating-point number), which has seven digits after the decimal point, whereas `sqrt 99999 :: Float` gives `316.2262`, which only has four digits after the point.

`Double` – double-precision floating-point numbers

This type is similar to `Float`, except that twice as much memory is used for storage of these numbers to increase their precision. For example, evaluating `sqrt 2 :: Double` gives `1.4142135623730951`. Using floating-point numbers is a specialist topic that requires a careful treatment of rounding errors, and we don't often use such numbers in this book.

We conclude this section by noting that a single number may have more than one numeric type. For example, the number `3` could have type `Int`, `Integer`, `Float` or `Double`. This raises the interesting question of what type such numbers should be assigned during the process of type inference, which will be answered later in this chapter when we consider type classes.

3.3 List types

A *list* is a sequence of *elements* of the same type, with the elements being enclosed in square parentheses and separated by commas. We write `[T]` for the type of all lists whose elements have type `T`. For example:

```
[False,True,False] :: [Bool]

['a','b','c','d'] :: [Char]

["One","Two","Three"] :: [String]
```

The number of elements in a list is called its *length*. The list `[]` of length zero is called the empty list, while lists of length one, such as `[False]`, `['a']`, and `[[]]` are called singleton lists. Note that `[[]]` and `[]` are different lists, the former being a singleton list comprising the empty list as its only element, and the latter being simply the empty list that has no elements.

There are three further points to note about list types. First of all, the type of a list conveys no information about its length. For example, the lists `[False,True]` and `[False,True,False]` both have type `[Bool]`, even though they have different lengths. Secondly, there are no restrictions on the type of the elements of a list. At present we are limited in the range of examples that we can give because

the only non-basic type that we have introduced at this point is list types, but we can have lists of lists, such as:

```
[['a','b'],['c','d','e']] :: [[Char]]
```

Finally, there is no restriction that a list must have a finite length. In particular, due to the use of lazy evaluation in Haskell, lists with an infinite length are both natural and practical, as we shall see in chapter 15.

3.4 Tuple types

A *tuple* is a finite sequence of *components* of possibly different types, with the components being enclosed in round parentheses and separated by commas. We write `(T1,T2,...,Tn)` for the type of all tuples whose `i`th components have type `Ti` for any `i` in the range `1` to `n`. For example:

```
(False,True) :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)

("Yes",True,'a') :: (String,Bool,Char)
```

The number of components in a tuple is called its *arity*. The tuple `()` of arity zero is called the empty tuple, tuples of arity two are called pairs, tuples of arity three are called triples, and so on. Tuples of arity one, such as `(False)`, are not permitted because they would conflict with the use of parentheses to make the evaluation order explicit, such as in `(1+2)*3`.

In a similar manner to list types, there are three further points to note about tuple types. First of all, the type of a tuple conveys its arity. For example, the type `(Bool,Char)` contains all pairs comprising a first component of type `Bool` and a second component of type `Char`. Secondly, there are no restrictions on the types of the components of a tuple. For example, we can now have tuples of tuples, tuples of lists, and lists of tuples:

```
('a',(False,'b')) :: (Char,(Bool,Char))

(['a','b'],[False,True]) :: ([Char],[Bool])

[('a',False),('b',True)] :: [(Char,Bool)]
```

Finally, note that tuples must have a finite arity, in order to ensure that tuple types can always be inferred prior to evaluation.

3.5 Function types

A *function* is a mapping from arguments of one type to results of another type. We write `T1 -> T2` for the type of all functions that map arguments of type `T1` to results of type `T2`. For example, we have:

```
not :: Bool -> Bool

even :: Int -> Bool
```

(The library function `even` decides if an integer is even.) Because there are no restrictions on the types of the arguments and results of a function, the simple notion of a function with a single argument and a single result is already sufficient to handle the case of multiple arguments and results, by packaging multiple values using lists or tuples. For example, we can define a function `add` that calculates the sum of a pair of integers, and a function `zeroto` that returns the list of integers from zero to a given limit, as follows:

```
add :: (Int,Int) -> Int
add (x,y) = x+y

zeroto :: Int -> [Int]
zeroto n = [0..n]
```

In these examples we have followed the Haskell convention of preceding function definitions by their types, which serves as useful documentation. Any such types provided manually by the user are checked for consistency with the types calculated automatically using type inference.

Note that there is no restriction that functions must be *total* on their argument type, in the sense that there may be some arguments for which the result is not defined. For example, the result of the library function `head` that selects the first element of a list is undefined if the list is empty:

```
> head []
*** Exception: Prelude.head: empty list
```

3.6 Curried functions

Functions with multiple arguments can also be handled in another, perhaps less obvious way, by exploiting the fact that functions are free to return functions as results. For example, consider the following definition:

```
add' :: Int -> (Int -> Int)
add' x y = x+y
```

The type states that `add'` is a function that takes an argument of type `Int`, and returns a result that is a function of type `Int -> Int`. The definition itself states that `add'` takes an integer `x` followed by an integer `y`, and returns the result `x+y`. More precisely, `add'` takes an integer `x` and returns a function, which in turn takes an integer `y` and returns the result `x+y`.

Note that the function `add'` produces the same final result as the function `add` from the previous section, but whereas `add` takes its two arguments at the same time packaged as a pair, `add'` takes its two arguments one at a time, as reflected in the different types of the two functions:

```
add :: (Int,Int) -> Int

add' :: Int -> (Int -> Int)
```

Functions with more than two arguments can also be handled using the same technique, by returning functions that return functions, and so on. For example, a function `mult` that takes three integers `x`, `y` and `z`, one at a time, and returns their product, can be defined as follows:

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x*y*z
```

This definition states that `mult` takes an integer `x` and returns a function, which in turn takes an integer `y` and returns another function, which finally takes an integer `z` and returns the result `x*y*z`.

Functions such as `add'` and `mult` that take their arguments one at a time are called *curried functions*. As well as being interesting in their own right, curried functions are also more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function with less than its full complement of arguments. For example, a function that increments an integer can be given by the partial application `add' 1 :: Int -> Int` of the curried function `add'` with only one of its two arguments.

To avoid excess parentheses when working with curried functions, two simple conventions are adopted. First of all, the function arrow `->` in types is assumed to associate to the right. For example, the type

```
Int -> Int -> Int -> Int
```

means

```
Int -> (Int -> (Int -> Int))
```

Consequently, function application, which is denoted silently using spacing, is assumed to associate to the left. For example, the application

```
mult x y z
```

means

```
((mult x) y) z
```

Unless tupling is explicitly required, all functions in Haskell with multiple arguments are normally defined as curried functions, and the two conventions above are used to reduce the number of parentheses that are required. In chapter 4 we will see how the meaning of curried function definitions can be formalised in a simple manner using the notion of lambda expressions.

3.7 Polymorphic types

The library function `length` calculates the length of any list, irrespective of the type of the elements of the list. For example, it can be used to calculate the length of a list of integers, a list of strings, or even a list of functions:

```
> length [1,3,5,7]
4

> length ["Yes","No"]
2

> length [sin,cos,tan]
3
```

The idea that `length` can be applied to lists whose elements have any type is made precise in its type by the inclusion of a *type variable*. Type variables must begin with a lower-case letter, and are usually simply named `a`, `b`, `c`, and so on. For example, the type of `length` is as follows:

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

That is, for any type `a`, the function `length` has type `[a] -> Int`. A type that contains one or more type variables is called *polymorphic* ("of many forms"), as is an expression with such a type. Hence, `[a] -> Int` is a polymorphic type and `length` is a polymorphic function. More generally, many of the functions provided in the standard prelude are polymorphic. For example:

```
fst :: (a,b) -> a

head :: [a] -> a

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

zip :: [a] -> [b] -> [(a,b)]

id :: a -> a
```

The type of a polymorphic function often gives a strong indication about the function's behaviour. For example, from the type `[a] -> [b] -> [(a,b)]` we can conclude that `zip` pairs up elements from two lists, although the type on its own doesn't capture the precise manner in which this is done.

3.8 Overloaded types

The arithmetic operator `+` calculates the sum of any two numbers of the same numeric type. For example, it can be used to calculate the sum of two integers, or the sum of two floating-point numbers:

```
> 1 + 2
3

> 1.0 + 2.0
3.0
```

The idea that `+` can be applied to numbers of any numeric type is made precise in its type by the inclusion of a *class constraint*. Class constraints are written in the form `C a`, where `C` is the name of a class and `a` is a type variable. For example, the type of the addition operator `+` is as follows:

```
(+) :: Num a => a -> a -> a
```

That is, for any type `a` that is an *instance* of the class `Num` of numeric types, the function `(+)` has type `a -> a -> a`. (Parenthesising an operator converts it into a curried function, as we shall see in chapter 4.)

A type that contains one or more class constraints is called *overloaded*, as is an expression with such a type. Hence, `Num a => a -> a -> a` is an overloaded type and `(+)` is an overloaded function. More generally, most of the numeric functions provided in the prelude are overloaded. For example:

```
(*) :: Num a => a -> a -> a

negate :: Num a => a -> a

abs :: Num a => a -> a
```

Numbers themselves are also overloaded. For example, `3 :: Num a => a` means that for any numeric type `a`, the value 3 has type `a`. In this manner, the value 3 could be an integer, a floating-point number, or more generally a value of any numeric type, depending on the context in which it is used.

3.9 Basic classes

Recall that a type is a collection of related values. Building upon this notion, a *class* is a collection of types that support certain overloaded operations called *methods*. Haskell provides a number of basic classes that are built-in to the language, of which the most commonly used are described below. (More advanced built-in classes are considered in part II of the book.)

Eq – equality types

This class contains types whose values can be compared for equality and inequality using the following two methods:

```
(==) :: a -> a -> Bool

(/=) :: a -> a -> Bool
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, and `Double` are instances of the `Eq` class, as are list and tuple types, provided that their element and component types are instances. For example:

```
> False == False
True

> 'a' == 'b'
False

> "abc" == "abc"
True

> [1,2] == [1,2,3]
False

> ('a',False) == ('a',False)
True
```

Note that function types are not in general instances of the `Eq` class, because it is not feasible in general to compare two functions for equality.

Ord – ordered types

This class contains types that are instances of the equality class `Eq`, but in addition whose values are totally (linearly) ordered, and as such can be compared and processed using the following six methods:

```
(<) :: a -> a -> Bool
```

```
(<=) :: a -> a -> Bool

(>) :: a -> a -> Bool

(>=) :: a -> a -> Bool

min :: a -> a -> a

max :: a -> a -> a
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, and `Double` are instances of the `Ord` class, as are list types and tuple types, provided that their element and component types are instances. For example:

```
> False < True
True

> min 'a' 'b'
'a'

> "elegant" < "elephant"
True

> [1,2,3] < [1,2]
False

> ('a',2) < ('b',1)
True

> ('a',2) < ('a',1)
False
```

Note that strings, lists and tuples are ordered *lexicographically*; that is, in the same way as words in a dictionary. For example, two pairs of the same type are in order if their first components are in order, in which case their second components are not considered, or if their first components are equal, in which case their second components must be in order.

`Show` – showable types

This class contains types whose values can be converted into strings of characters using the following method:

```
show :: a -> String
```

All the basic types Bool, Char, String, Int, Integer, Float, and Double a instances of the Show class, as are list types and tuple types, provided that th element and component types are instances. For example:

```
> show False
"False"

> show 'a'
"'a'"

> show 123
"123"

> show [1,2,3]
"[1,2,3]"

> show ('a',False)
"('a',False)"
```

Read – readable types

This class is dual to Show, and contains types whose values can be converte from strings of characters using the following method:

```
read :: String -> a
```

All the basic types Bool, Char, String, Int, Integer, Float, and Double are instances of the Read class, as are list types and tuple types, provided that their element and component types are instances. For example:

```
> read "False" :: Bool
False

> read "'a'" :: Char
'a'

> read "123" :: Int
123

> read "[1,2,3]" :: [Int]
[1,2,3]

> read "('a',False)" :: (Char,Bool)
('a',False)
```

The use of :: in these examples resolves the type of the result, which would otherwise not be able to be inferred by GHCi. In practice, however, the necessary type information can usually be inferred automatically from the context. For example, the expression `not (read "False")` requires no explicit type information, because the application of the logical negation function `not` implies that `read "False"` must have type `Bool`.

Note that the result of `read` is undefined if its argument is not syntactically valid. For example, the expression `not (read "abc")` produces an error when evaluated, because `"abc"` cannot be read as a logical value:

```
> not (read "abc")
*** Exception: Prelude.read: no parse
```

Num – numeric types

This class contains types whose values are numeric, and as such can be processed using the following six methods:

```
(+) :: a -> a -> a

(-) :: a -> a -> a

(*) :: a -> a -> a

negate :: a -> a

abs :: a -> a

signum :: a -> a
```

(The method `negate` returns the negation of a number, `abs` returns the absolute value, while `signum` returns the sign.) The basic types `Int`, `Integer`, `Float`, and `Double` are instances of the `Num` class. For example:

```
> 1 + 2
3

> 1.0 + 2.0
3.0

> negate 3.0
-3.0

> abs (-3)
3
```

```
> signum (-3)
-1
```

As illustrated above, negative numbers must be parenthesised when used as arguments to functions, to ensure the correct interpretation of the minus sign. For example, `abs -3` without parentheses means `abs - 3`, which is both the incorrect meaning here and an ill-typed expression.

Note that the `Num` class does not provide a division method, but as we shall now see, division is handled separately using two special classes, one for integral numbers and one for fractional numbers.

`Integral` – integral types

This class contains types that are instances of the numeric class `Num`, but in addition whose values are integers, and as such support the methods of integer division and integer remainder:

```
div :: a -> a -> a

mod :: a -> a -> a
```

In practice, these two methods are often written between their two arguments by enclosing their names in single back quotes. The basic types `Int` and `Integer` are instances of the `Integral` class. For example:

```
> 7 `div` 2
3

> 7 `mod` 2
1
```

For efficiency reasons, a number of prelude functions that involve both lists and integers (such as `take` and `drop`) are restricted to the type `Int` of finite-precision integers, rather than being applicable to any instance of the `Integral` class. If required, however, such generic versions of these functions are provided as part of an additional library file called `Data.List`.

`Fractional` – fractional types

This class contains types that are instances of the numeric class `Num`, but in addition whose values are non-integral, and as such support the methods of fractional division and fractional reciprocation:

```
(/) :: a -> a -> a

recip :: a -> a
```

The basic types `Float` and `Double` are instances. For example:

```
> 7.0 / 2.0
3.5

> recip 2.0
0.5
```

3.10 Chapter remarks

The term `Bool` for the type of logical values celebrates the pioneering work of George Boole on symbolic logic, while the term *curried* for functions that take their arguments one at a time celebrates the work of Haskell Curry (after whom the language Haskell itself is named) on such functions. The relationship between the type of a polymorphic function and its behaviour is formalised in [3]. A more detailed account of the type system is given in the Haskell Report [4], and a formal description of the type system can be found in [5].

3.11 Exercises

1. What are the types of the following values?

```
['a','b','c']

('a','b','c')

[(False,'0'),(True,'1')]

([False,True],['0','1'])

[tail, init, reverse]
```

2. Write down definitions that have the following types; it does not matter what the definitions actually do as long as they are type correct.

```
bools :: [Bool]

nums :: [[Int]]

add :: Int -> Int -> Int -> Int

copy :: a -> (a,a)
```

```
apply :: (a -> b) -> a -> b
```

3. What are the types of the following functions?

```
second xs = head (tail xs)

swap (x,y) = (y,x)

pair x y = (x,y)

double x = x*2

palindrome xs = reverse xs == xs

twice f x = f (f x)
```

   Hint: take care to include the necessary class constraints in the types if the functions are defined using overloaded operators.

4. Check your answers to the preceding three questions using GHCi.

5. Why is it not feasible in general for function types to be instances of the `Eq` class? When is it feasible? Hint: two functions of the same type are equal if they always return equal results for equal arguments.

Solutions to exercises 1 and 2 are given in appendix A.

4 Defining functions

In this chapter we introduce a range of mechanisms for defining functions in Haskell. We start with conditional expressions and guarded equations, then introduce the simple but powerful idea of pattern matching, and conclude with the concepts of lambda expressions and operator sections.

4.1 New from old

Perhaps the most straightforward way to define new functions is simply by combining one or more existing functions. For example, a few library functions that can be defined in this way are shown below.

- Decide if an integer is even:

```
even :: Integral a => a -> Bool
even n = n `mod` 2 == 0
```

- Split a list at the nth element:

```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)
```

- Reciprocation:

```
recip :: Fractional a => a -> a
recip n = 1/n
```

Note the use of the class constraints in the types for `even` and `recip` above, which make precise the idea that these functions can be applied to numbers of any integral and fractional types, respectively.

4.2 Conditional expressions

Haskell provides a range of different ways to define functions that choose between a number of possible results. The simplest are *conditional expressions*, which use a logical expression called a *condition* to choose between two results of the same type. If the condition is `True`, then the first result is chosen, and if it is `False`,

then the second result is chosen. For example, the library function `abs` that returns the absolute value of an integer can be defined as follows:

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

Conditional expressions may be nested, in the sense that they can contain other conditional expressions. For example, the library function `signum` that returns the sign of an integer can be defined as follows:

```
signum :: Int -> Int
signum n = if n < 0 then -1 else
               if n == 0 then 0 else 1
```

Note that unlike in some programming languages, conditional expressions in Haskell must always have an `else` branch, which avoids the well-known *dangling else* problem. For example, if `else` branches were optional, then the expression `if True then if False then 1 else 2` could either return the result 2 or produce an error, depending upon whether the single `else` branch was assumed to be part of the inner or outer conditional expression.

4.3 Guarded equations

As an alternative to using conditional expressions, functions can also be defined using *guarded equations*, in which a sequence of logical expressions called *guards* is used to choose between a sequence of results of the same type. If the first guard is `True`, then the first result is chosen; otherwise, if the second is `True`, then the second result is chosen, and so on. For example, the library function `abs` can also be defined using guarded equations as follows:

```
abs n | n >= 0    = n
      | otherwise = -n
```

The symbol `|` is read as *such that*, and the guard `otherwise` is defined in the standard prelude simply by `otherwise = True`. Ending a sequence of guards with `otherwise` is not necessary, but provides a convenient way of handling all other cases, as well as avoiding the possibility that none of the guards in the sequence is `True`, which would otherwise result in an error.

The main benefit of guarded equations over conditional expressions is that definitions with multiple guards are easier to read. For example, the library function `signum` is easier to understand when defined as follows:

```
signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

4.4 Pattern matching

Many functions have a simple and intuitive definition using *pattern matching*, in which a sequence of syntactic expressions called *patterns* is used to choose between a sequence of results of the same type. If the first pattern is *matched*, then the first result is chosen; otherwise, if the second is matched, then the second result is chosen, and so on. For example, the library function `not` that returns the negation of a logical value can be defined as follows:

```
not :: Bool -> Bool
not False = True
not True  = False
```

Functions with more than one argument can also be defined using pattern matching, in which case the patterns for each argument are matched in order within each equation. For example, the library operator `&&` that returns the conjunction of two logical values can be defined as follows:

```
(&&) :: Bool -> Bool -> Bool
True && True   = True
True && False  = False
False && True  = False
False && False = False
```

However, this definition can be simplified by combining the last three equations into a single equation that returns `False` independent of the values of the two arguments, using the *wildcard pattern* `_` that matches any value:

```
True && True = True
_    && _    = False
```

This version also has the benefit that, under lazy evaluation as discussed in chapter 15, if the first argument is `False`, then the result `False` is returned without the need to evaluate the second argument. In practice, the prelude defines `&&` using equations that have this same property, but make the choice about which equation applies using the value of the first argument only:

```
True  && b = b
False && _ = False
```

That is, if the first argument is `True`, then the result is the value of the second argument, and, if the first argument is `False`, then the result is `False`.

Note that Haskell does not permit the same name to be used for more than one argument in a single equation. For example, the following definition for the operator `&&` is based upon the observation that, if the two logical arguments are equal, then the result is the same value, otherwise the result is `False`, but is invalid because of the above naming requirement:

```
b && b = b
_ && _ = False
```

If desired, however, a valid version of this definition can be obtained by using a guard to decide if the two arguments are equal:

```
b && c | b == c    = b
       | otherwise = False
```

So far, we have only considered basic patterns that are either values, variables, or the wildcard pattern. In the remainder of this section we introduce two useful ways to build larger patterns by combining smaller patterns.

Tuple patterns

A tuple of patterns is itself a pattern, which matches any tuple of the same arity whose components all match the corresponding patterns in order. For example, the library functions `fst` and `snd` that respectively select the first and second components of a pair are defined as follows:

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y
```

List patterns

Similarly, a list of patterns is itself a pattern, which matches any list of the same length whose elements all match the corresponding patterns in order. For example, a function `test` that decides if a list contains precisely three characters beginning with the letter `'a'` can be defined as follows:

```
test :: [Char] -> Bool
test ['a',_,_] = True
test _         = False
```

Up to this point, we have viewed lists as a primitive notion in Haskell. In fact they are not primitive as such, but are constructed one element at a time starting from the empty list `[]` using an operator `:` called *cons* that *cons*tructs a new list by prepending a new element to the start of an existing list. For example, the list `[1,2,3]` can be decomposed as follows:

```
   [1,2,3]
=      { list notation }
   1 : [2,3]
=      { list notation }
```

```
    1 : (2 : [3])
=       { list notation }
    1 : (2 : (3 : []))
```

That is, `[1,2,3]` is just an abbreviation for `1:(2:(3:[]))`. To avoid excess parentheses when working with such lists, the cons operator is assumed to associate to the right. For example, `1:2:3:[]` means `1:(2:(3:[]))`.

As well as being used to construct lists, the cons operator can also be used to construct patterns, which match any non-empty list whose first and remaining elements match the corresponding patterns in order. For example, we can now define a more general version of the function `test` that decides if a list containing any number of characters begins with the letter `'a'`:

```
test :: [Char] -> Bool
test ('a':_) = True
test _       = False
```

Similarly, the library functions `head` and `tail` that respectively select and remove the first element of a non-empty list are defined as follows:

```
head :: [a] -> a
head (x:_) = x

tail :: [a] -> [a]
tail (_:xs) = xs
```

Note that cons patterns must be parenthesised, because function application has higher priority than all other operators in the language. For example, the definition `head x:_ = x` without parentheses means `(head x):_ = x`, which is both the incorrect meaning and an invalid definition.

4.5 Lambda expressions

As an alternative to defining functions using equations, functions can also be constructed using *lambda expressions*, which comprise a pattern for each of the arguments, a body that specifies how the result can be calculated in terms of the arguments, but do not give a name for the function itself. In other words, lambda expressions are nameless functions.

For example, the nameless function that takes a single number `x` as its argument, and produces the result `x + x`, can be constructed as follows:

```
\x -> x + x
```

The symbol \ represents the Greek letter *lambda*, written as λ. Despite the fact that they have no names, functions constructed using lambda expressions can be used in the same way as any other functions. For example:

```
> (\x -> x + x) 2
4
```

As well as being interesting in their own right, lambda expressions have a number of practical applications. First of all, they can be used to formalise the meaning of curried function definitions. For example, the definition

```
add :: Int -> Int -> Int
add x y = x + y
```

can be understood as meaning

```
add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)
```

which makes precise that `add` is a function that takes an integer `x` and returns a function, which in turn takes another integer `y` and returns the result `x + y`. Moreover, rewriting the original definition in this manner also has the benefit that the type for the function and the manner in which it is defined now have the same syntactic form, namely `? -> (? -> ?)`.

Secondly, lambda expressions are also useful when defining functions that return functions as results by their very nature, rather than as a consequence of currying. For example, the library function `const` that returns a constant function that always produces a given value can be defined as follows:

```
const :: a -> b -> a
const x _ = x
```

However, it is more appealing to define `const` in a way that makes explicit that it returns a function as its result, by including parentheses in the type and using a lambda expression in the definition itself:

```
const :: a -> (b -> a)
const x = \_ -> x
```

Finally, lambda expressions can be used to avoid having to name a function that is only referenced once in a program. For example, a function `odds` that returns the first `n` odd integers can be defined as follows:

```
odds :: Int -> [Int]
odds n = map f [0..n-1]
         where f x = x*2 + 1
```

(The library function `map` applies a function to all elements of a list.) However, because the locally defined function `f` is only referenced once, the definition for `odds` can be simplified by using a lambda expression:

```
odds :: Int -> [Int]
odds n = map (\x -> x*2 + 1) [0..n-1]
```

4.6 Operator sections

Functions such as + that are written between their two arguments are called *operators*. As we have already seen, any function with two arguments can be converted into an operator by enclosing the name of the function in single back quotes, as in `7 ‘div‘ 2`. However, the converse is also possible. In particular, any operator can be converted into a curried function that is written before its arguments by enclosing the name of the operator in parentheses, as in `(+) 1 2`. Moreover, this convention also allows one of the arguments to be included in the parentheses if desired, as in `(1+) 2` and `(+2) 1`.

In general, if `#` is an operator, then expressions of the form `(#)`, `(x #)`, and `(# y)` for arguments `x` and `y` are called *sections*, whose meaning as functions can be formalised using lambda expressions as follows:

```
(#)   = \x -> (\y -> x # y)

(x #) = \y -> x # y

(# y) = \x -> x # y
```

Sections have three primary applications. First of all, they can be used to construct a number of simple but useful functions in a particularly compact way, as shown in the following examples:

`(+)` is the addition function `\x -> (\y -> x+y)`

`(1+)` is the successor function `\y -> 1+y`

`(1/)` is the reciprocation function `\y -> 1/y`

`(*2)` is the doubling function `\x -> x*2`

`(/2)` is the halving function `\x -> x/2`

Secondly, sections are necessary when stating the type of operators, because an operator itself is not a valid expression in Haskell. For example, the type of the addition operator + for integers is stated as follows:

```
(+) :: Int -> Int -> Int
```

Finally, sections are also necessary when using operators as arguments to other functions. For example, the library function `sum` that calculates the sum of a list of integers can be defined by using the operator + as an argument to the library function `foldl`, which is itself discussed in chapter 7:

```
sum :: [Int] -> Int
sum = foldl (+) 0
```

4.7 Chapter remarks

A formal meaning for pattern matching by translation using more primitive features of the language is given in the Haskell Report [4]. The Greek letter λ used when defining nameless functions comes from the *lambda calculus* [6], the mathematical theory of functions upon which Haskell is founded.

4.8 Exercises

1. Using library functions, define a function `halve :: [a] -> ([a],[a])` that splits an even-lengthed list into two halves. For example:

   ```
   > halve [1,2,3,4,5,6]
   ([1,2,3],[4,5,6])
   ```

2. Define a function `third :: [a] -> a` that returns the third element in a list that contains at least this many elements using:

   a. `head` and `tail`;

   b. list indexing `!!`;

   c. pattern matching.

3. Consider a function `safetail :: [a] -> [a]` that behaves in the same way as `tail` except that it maps the empty list to itself rather than producing an error. Using `tail` and the function `null :: [a] -> Bool` that decides if a list is empty or not, define `safetail` using:

   a. a conditional expression;

   b. guarded equations;

   c. pattern matching.

4. In a similar way to `&&` in section 4.4, show how the disjunction operator `||` can be defined in four different ways using pattern matching.

5. Without using any other library functions or operators, show how the meaning of the following pattern matching definition for logical conjunction `&&` can be formalised using conditional expressions:

   ```
   True  && True  = True
   _     && _     = False
   ```

   Hint: use two nested conditional expressions.

6. Do the same for the following alternative definition, and note the difference in the number of conditional expressions that are required:

```
True  && b = b
False && _ = False
```

7. Show how the meaning of the following curried function definition can be formalised in terms of lambda expressions:

```
mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z
```

8. The *Luhn algorithm* is used to check bank card numbers for simple errors such as mistyping a digit, and proceeds as follows:

- consider each digit as a separate number;
- moving left, double every other number from the second last;
- subtract 9 from each number that is now greater than 9;
- add all the resulting numbers together;
- if the total is divisible by 10, the card number is valid.

Define a function `luhnDouble :: Int -> Int` that doubles a digit and subtracts 9 if the result is greater than 9. For example:

```
> luhnDouble 3
6

> luhnDouble 6
3
```

Using `luhnDouble` and the integer remainder function `mod`, define a function `luhn :: Int -> Int -> Int -> Int -> Bool` that decides if a four-digit bank card number is valid. For example:

```
> luhn 1 7 8 4
True

> luhn 4 7 8 3
False
```

In the exercises for chapter 7 we will consider a more general version of this function that accepts card numbers of any length.

Solutions to exercises 1–4 are given in appendix A.

5 List comprehensions

In this chapter we introduce list comprehensions, which allow many functions on lists to be defined in simple manner. We start by explaining generators and guards, then introduce the function `zip` and the idea of string comprehensions, and conclude by developing a program to crack the Caesar cipher.

5.1 Basic concepts

In mathematics, the *comprehension* notation can be used to construct new sets from existing sets. For example, the comprehension $\{x^2 \mid x \in \{1..5\}\}$ produces the set $\{1, 4, 9, 16, 25\}$ of all numbers x^2 such that x is an element of the set $\{1..5\}$. In Haskell, a similar comprehension notation can be used to construct new lists from existing lists. For example:

```
> [x^2 | x <- [1..5]]
[1,4,9,16,25]
```

The symbol `|` is read as *such that*, `<-` is read as *is drawn from*, and the expression `x <- [1..5]` is called a *generator*. A list comprehension can have more than one generator, with successive generators being separated by commas. For example, the list of all possible pairings of an element from the list `[1,2,3]` with an element from the list `[4,5]` can be produced as follows:

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Changing the order of the two generators in this example produces the same set of pairs, but arranged in a different order:

```
> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

In particular, whereas in this case the `x` components of the pairs change more frequently than the `y` components (`1,2,3,1,2,3` versus `4,4,4,5,5,5`), in the previous case the `y` components change more frequently than the `x` components (`4,5,4,5,4,5` versus `1,1,2,2,3,3`). These behaviours can be understood by thinking

of later generators as being more deeply nested, and hence changing the values of their variables more frequently than earlier generators.

Later generators can also depend upon the values of variables from earlier generators. For example, the list of all possible ordered pairings of elements from the list `[1..3]` can be produced as follows:

```
> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

As a more practical example of this idea, the library function `concat` that concatenates a list of lists can be defined by using one generator to select each list in turn, and another to select each element from each list:

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

The wildcard pattern `_` is sometimes useful in generators to discard certain elements from a list. For example, a function that selects all the first components from a list of pairs can be defined as follows:

```
firsts :: [(a,b)] -> [a]
firsts ps = [x | (x,_) <- ps]
```

Similarly, the library function that calculates the length of a list can be defined by replacing each element by one and summing the resulting list:

```
length :: [a] -> Int
length xs = sum [1 | _ <- xs]
```

In this case, the generator `_ <- xs` simply serves as a counter to govern the production of the appropriate number of ones.

5.2 Guards

List comprehensions can also use logical expressions called *guards* to filter the values produced by earlier generators. If a guard is `True`, then the current values are retained; if it is `False`, then they are discarded. For example, the comprehension `[x | x <- [1..10], even x]` produces the list `[2,4,6,8,10]` of all even numbers from the list `[1..10]`. Similarly, a function that maps a positive integer to its list of positive factors can be defined by:

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
```

For example:

```
> factors 15
[1,3,5,15]

> factors 7
[1,7]
```

Recall that an integer greater than one is *prime* if its only positive factors are one and the number itself. Hence, by using `factors`, a simple function that decides if an integer is prime can be defined as follows:

```
prime :: Int -> Bool
prime n = factors n == [1,n]
```

For example:

```
> prime 15
False

> prime 7
True
```

Note that deciding that a number such as `15` is not prime does not require the function `prime` to produce all of its factors, because under lazy evaluation the result `False` is returned as soon as any factor other than one or the number itself is produced, which for this example is given by the factor `3`.

Returning to list comprehensions, using `prime` we can now define a function that produces the list of all prime numbers up to a given limit:

```
primes :: Int -> [Int]
primes n = [x | x <- [2..n], prime x]
```

For example:

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

In chapter 15 we will present a more efficient program to generate prime numbers using the famous *sieve of Eratosthenes*, which has a particularly clear and concise implementation in Haskell using the idea of lazy evaluation.

As a final example concerning guards, suppose that we represent a lookup table by a list of pairs of keys and values. Then for any type of keys that supports equality, a function `find` that returns the list of all values that are associated with a given key in a table can be defined as follows:

```
find :: Eq a => a -> [(a,b)] -> [b]
find k t = [v | (k',v) <- t, k == k']
```

For example:

```
> find 'b' [('a',1),('b',2),('c',3),('b',4)]
[2,4]
```

5.3 The `zip` function

The library function `zip` produces a new list by pairing successive elements from two existing lists until either or both lists are exhausted. For example:

```
> zip ['a','b','c'] [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

The function `zip` is often useful when programming with list comprehensions. For example, suppose that we define a function that returns the list of all pairs of adjacent elements from a list as follows:

```
pairs  :: [a] -> [(a,a)]
pairs xs = zip xs (tail xs)
```

For example:

```
> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

Then using `pairs` we can now define a function that decides if a list of elements of any ordered type is sorted by simply checking that all pairs of adjacent elements from the list are in the correct order:

```
sorted :: Ord a => [a] -> Bool
sorted xs = and [x <= y | (x,y) <- pairs xs]
```

For example:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Similarly to the function `prime`, deciding that a list such as `[1,3,2,4]` is not sorted may not require the function `sorted` to produce all pairs of adjacent elements, because the result `False` is returned as soon as any non-ordered pair is produced, which in this example is given by the pair `(3,2)`.

Using `zip` we can also define a function that returns the list of all positions at which a value occurs in a list, by pairing each element with its position, and selecting those positions at which the desired value occurs:

```
positions :: Eq a => a -> [a] -> [Int]
positions x xs = [i | (x',i) <- zip xs [0..], x == x']
```

For example:

```
> positions False [True, False, True, False]
[1,3]
```

Within the definition for `positions`, the expression `[0..]` produces the list of indices `[0,1,2,3,...]`. This list is notionally *infinite*, but under lazy evaluation only as many elements of the list as required by the context in which it is used, in this case zipping with the input list `xs`, will actually be produced. Exploiting lazy evaluation in this manner avoids the need to explicitly produce a list of indices of the same length as the input list.

5.4 String comprehensions

Up to this point we have viewed strings as a primitive notion in Haskell. In fact they are not primitive, but are constructed as lists of characters. For example, the string `"abc" :: String` is just an abbreviation for the list of characters `['a','b','c'] :: [Char]`. Because strings are lists, any polymorphic function on lists can also be used with strings. For example:

```
> "abcde" !! 2
'c'

> take 3 "abcde"
"abc"

> length "abcde"
5

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

For the same reason, list comprehensions can also be used to define functions on strings, such as functions that return the number of lower-case letters and particular characters that occur in a string, respectively:

```
lowers :: String -> Int
lowers xs = length [x | x <- xs, x >= 'a' && x <= 'z']

count :: Char -> String -> Int
count x xs = length [x' | x' <- xs, x == x']
```

For example:

```
> lowers "Haskell"
6

> count 's' "Mississippi"
4
```

5.5 The Caesar cipher

We conclude this chapter with an extended programming example. Consider the problem of encoding a string in order to disguise its contents. A well-known encoding method is the *Caesar cipher*, named after its use by Julius Caesar more than 2,000 years ago. To encode a string, Caesar simply replaced each letter in the string by the letter three places further down in the alphabet, wrapping around at the end of the alphabet. For example, the string

```
"haskell is fun"
```

would be encoded as

```
"kdvnhoo lv ixq"
```

More generally, the specific shift factor of three used by Caesar can be replaced by any integer between one and twenty-five, thereby giving twenty-five different ways of encoding a string. For example, with a shift factor of ten, the original string above would be encoded as follows:

```
"rkcuovv sc pex"
```

In the remainder of this section we show how Haskell can be used to implement the Caesar cipher, and how the cipher itself can easily be cracked by exploiting information about letter frequencies in English text.

Encoding and decoding

We will use a number of standard functions on characters that are provided in a library called `Data.Char`, which can be loaded into a Haskell script by including the following declaration at the start of the script:

```
import Data.Char
```

For simplicity, we will only encode the lower-case letters within a string, leaving other characters such as upper-case letters and punctuation unchanged. We begin by defining a function `let2int` that converts a lower-case letter between `'a'` and `'z'` into the corresponding integer between `0` and `25`, together with a function `int2let` that performs the opposite conversion:

```
let2int :: Char -> Int
let2int c = ord c - ord 'a'

int2let :: Int -> Char
int2let n = chr (ord 'a' + n)
```

(The library functions `ord :: Char -> Int` and `chr :: Int -> Char` convert between characters and their Unicode numbers.) For example:

```
> let2int 'a'
0

> int2let 0
'a'
```

Using these two functions, we can define a function `shift` that applies a shift factor to a lower-case letter by converting the letter into the corresponding integer, adding on the shift factor and taking the remainder when divided by twenty-six (thereby wrapping around at the end of the alphabet), and converting the resulting integer back into a lower-case letter:

```
shift :: Int -> Char -> Char
shift n c | isLower c = int2let ((let2int c + n) `mod` 26)
          | otherwise = c
```

(The library function `isLower :: Char -> Bool` decides if a character is a lower-case letter.) Note that this function accepts both positive and negative shift factors, and that only lower-case letters are changed. For example:

```
> shift 3 'a'
'd'

> shift 3 'z'
'c'

> shift (-3) 'c'
'z'

> shift 3 ' '
' '
```

Using `shift` within a list comprehension, it is now easy to define a function that encodes a string using a given shift factor:

```
encode :: Int -> String -> String
encode n xs = [shift n x | x <- xs]
```

A separate function to decode a string is not required, because this can be achieved by simply using a negative shift factor. For example:

```
> encode 3 "haskell is fun"
"kdvnhoo lv ixq"

> encode (-3) "kdvnhoo lv ixq"
"haskell is fun"
```

Frequency tables

The key to cracking the Caesar cipher is the observation that some letters are used more frequently than others in English text. By analysing a large volume of such text, one can derive the following table of approximate percentage frequencies of the twenty-six letters of alphabet:

```
table :: [Float]
table = [8.1, 1.5, 2.8, 4.2, 12.7, 2.2, 2.0, 6.1, 7.0,
         0.2, 0.8, 4.0, 2.4, 6.7, 7.5, 1.9, 0.1, 6.0,
         6.3, 9.0, 2.8, 1.0, 2.4, 0.2, 2.0, 0.1]
```

For example, the letter `'e'` occurs most often, with a frequency of 12.7%, while `'q'` and `'z'` occur least often, with a frequency of just 0.1%. It is also useful to produce frequency tables for individual strings. To this end, we first define a function that calculates the percentage of one integer with respect to another, returning the result as a floating-point number:

```
percent :: Int -> Int -> Float
percent n m = (fromIntegral n / fromIntegral m) * 100
```

(The library function `fromIntegral :: Int -> Float` converts an integer into a floating-point number.) For example:

```
> percent 5 15
33.333336
```

Using `percent` within a list comprehension, together with the functions `lowers` and `count` from the previous section, we can now define a function that returns a frequency table for any given string:

```
freqs :: String -> [Float]
freqs xs = [percent (count x xs) n | x <- ['a'..'z']]
           where n = lowers xs
```

For example:

```
> freqs "abbcccddddeeeee"
[6.666667, 13.333334, 20.0, 26.666668, ..., 0.0]
```

That is, the letter `'a'` occurs with a frequency of approximately 6.6%, the letter `'b'` with a frequency of 13.3%, and so on. The use of the local definition `n = lowers xs` within `freqs` ensures that the number of lower-case letters in the argument string is calculated once, rather than each of the twenty-six times that this number is used within the list comprehension.

Cracking the cipher

A standard method for comparing a list of observed frequencies os with a list of expected frequencies es is the *chi-square statistic*, defined by the following summation in which n denotes the length of the two lists, and xs_i denotes the ith element of a list xs counting from zero:

$$\sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

The details of the chi-square statistic need not concern us here, only the fact that the smaller the value it produces the better the match between the two frequency lists. Using the library function `zip` and a list comprehension, it is easy to translate the above formula into a function definition:

```
chisqr :: [Float] -> [Float] -> Float
chisqr os es = sum [((o-e)^2)/e | (o,e) <- zip os es]
```

In turn, we define a function that rotates the elements of a list `n` places to the left, wrapping around at the start of the list, and assuming that the integer argument `n` is between zero and the length of the list:

```
rotate :: Int -> [a] -> [a]
rotate n xs = drop n xs ++ take n xs
```

For example:

```
> rotate 3 [1,2,3,4,5]
[4,5,1,2,3]
```

Now suppose that we are given an encoded string, but not the shift factor that was used to encode it, and wish to determine this number in order that we can decode the string. This can usually be achieved by producing the frequency table of the encoded string, calculating the chi-square statistic for each possible rotation of this table with respect to the table of expected frequencies, and using the position of the minimum chi-square value as the shift factor. For example, if we let `table' = freqs "kdvnhoo lv ixq"`, then

```
[chisqr (rotate n table') table | n <- [0..25]]
```

gives the result

```
[1408.8524, 640.0218, 612.3969, 202.42024, ..., 626.4024]
```

in which the minimum value, 202.42024, appears at position three in this list. Hence, we conclude that three is the most likely shift factor that was used to encode the string. Using the function `positions` from earlier in this chapter, this procedure can be implemented as follows:

```
crack :: String -> String
crack xs = encode (-factor) xs
  where
    factor = head (positions (minimum chitab) chitab)
    chitab = [chisqr (rotate n table') table | n <- [0..25]]
    table' = freqs xs
```

For example:

```
> crack "kdvnhoo lv ixq"
"haskell is fun"

> crack "vscd mywzboroxcsyxc kbo ecopev"
"list comprehensions are useful"
```

More generally, the `crack` function can decode most strings produced using the Caesar cipher. Note, however, that it may not be successful if the string is short or has an unusual distribution of letters. For example:

```
> crack (encode 3 "haskell")
"piasmtt"

> crack (encode 3 "boxing wizards jump quickly")
"wjsdib rduvmyn ephk lpdxfgt"
```

5.6 Chapter remarks

The term *comprehension* comes from the *axiom of comprehension* in set theory, which makes precise the idea of constructing a set by selecting all values that satisfy a particular property. A formal meaning for list comprehensions by translation using more primitive features of the language is given in the Haskell Report [4]. A popular account of the Caesar cipher, and many other famous cryptographic methods, is given in *The Code Book* [7].

5.7 Exercises

1. Using a list comprehension, give an expression that calculates the sum $1^2 + 2^2 + \ldots 100^2$ of the first one hundred integer squares.

2. Suppose that a *coordinate grid* of size $m \times n$ is given by the list of all pairs (x, y) of integers such that $0 \leqslant x \leqslant m$ and $0 \leqslant y \leqslant n$. Using a list comprehension, define a function `grid :: Int -> Int -> [(Int,Int)]` that returns a coordinate grid of a given size. For example:

```
> grid 1 2
[(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]
```

3. Using a list comprehension and the function `grid` above, define a function `square :: Int -> [(Int,Int)]` that returns a coordinate square of size n, excluding the diagonal from $(0, 0)$ to (n, n). For example:

```
> square 2
[(0,1),(0,2),(1,0),(1,2),(2,0),(2,1)]
```

4. In a similar way to the function `length`, show how the library function `replicate :: Int -> a -> [a]` that produces a list of identical elements can be defined using a list comprehension. For example:

```
> replicate 3 True
[True,True,True]
```

5. A triple (x, y, z) of positive integers is *Pythagorean* if it satisfies the equation $x^2 + y^2 = z^2$. Using a list comprehension with three generators, define a function `pyths :: Int -> [(Int,Int,Int)]` that returns the list of all such triples whose components are at most a given limit. For example:

```
> pyths 10
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
```

6. A positive integer is *perfect* if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension and the function `factors`, define a function `perfects :: Int -> [Int]` that returns the list of all perfect numbers up to a given limit. For example:

```
> perfects 500
[6,28,496]
```

7. Show how the list comprehension `[(x,y) | x <- [1,2], y <- [3,4]]` with

two generators can be re-expressed using two comprehensions with single generators. Hint: nest one comprehension within the other and make use of the library function `concat :: [[a]] -> [a]`.

8. Redefine the function `positions` using the function `find`.

9. The *scalar product* of two lists of integers xs and ys of length n is given by the sum of the products of corresponding integers:

$$\sum_{i=0}^{n-1} (xs_i * ys_i)$$

In a similar manner to `chisqr`, show how a list comprehension can be used to define a function `scalarproduct :: [Int] -> [Int] -> Int` that returns the scalar product of two lists. For example:

```
> scalarproduct [1,2,3] [4,5,6]
32
```

10. Modify the Caesar cipher program to also handle upper-case letters.

Solutions to exercises 1–5 are given in appendix A.