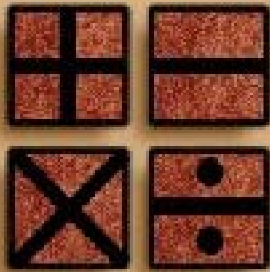
Programming in Haskell

Graham Hutton



© Graham Hutton

Version of August 10, 2005

NOT FOR DISTRIBUTION

For Annette, Callum and Tom

Contents

Preface **7**

1 Introduction **9**
1.1 Functions . . . 9
1.2 Functional programming . . . 10
1.3 Features of Haskell . . . 12
1.4 Historical background . . . 14
1.5 A taste of Haskell . . . 15
1.6 Chapter remarks . . . 17
1.7 Exercises . . . 17

2 First Steps **19**
2.1 The Hugs system . . . 19
2.2 The standard prelude . . . 19
2.3 Function application . . . 22
2.4 Haskell scripts . . . 22
2.5 Chapter remarks . . . 25
2.6 Exercises . . . 25

3 Types and Classes **27**
3.1 Basic concepts . . . 27
3.2 Basic types . . . 28
3.3 List types . . . 30
3.4 Tuple types . . . 30
3.5 Function types . . . 31
3.6 Curried functions . . . 32
3.7 Polymorphic types . . . 33
3.8 Overloaded types . . . 34
3.9 Basic classes . . . 34
3.10 Chapter remarks . . . 39
3.11 Exercises . . . 39

4 Defining Functions **41**
4.1 New from old . . . 41
4.2 Conditional expressions . . . 42

4.3 Guarded equations 42
4.4 Pattern matching 43
4.5 Lambda expressions 46
4.6 Sections 47
4.7 Chapter remarks 48
4.8 Exercises 48

5 List Comprehensions **51**
5.1 Generators 51
5.2 Guards 52
5.3 The *zip* function 54
5.4 String comprehensions 55
5.5 The Caesar cipher 55
5.6 Chapter remarks 60
5.7 Exercises 60

6 Recursive Functions **63**
6.1 Basic concepts 63
6.2 Recursion on lists 65
6.3 Multiple arguments 67
6.4 Multiple recursion 68
6.5 Mutual recursion 69
6.6 Advice on recursion 70
6.7 Chapter remarks 75
6.8 Exercises 75

7 Higher-Order Functions **77**
7.1 Basic concepts 77
7.2 Processing lists 78
7.3 The *foldr* function 80
7.4 The *foldl* function 83
7.5 The composition operator 85
7.6 String transmitter 86
7.7 Chapter remarks 89
7.8 Exercises 89

8 Functional Parsers **91**
8.1 Parsers 91
8.2 The parser type 92
8.3 Basic parsers 92
8.4 Sequencing 94
8.5 Choice 95
8.6 Derived primitives 96
8.7 Handling spacing 98
8.8 Arithmetic expressions 99
8.9 Chapter remarks 103

8.10 Exercises 103

9 Interactive Programs 105
9.1 Interaction 105
9.2 The input/output type 106
9.3 Basic actions 107
9.4 Sequencing 107
9.5 Derived primitives 108
9.6 Calculator 110
9.7 Game of life 113
9.8 Chapter remarks 116
9.9 Exercises 116

10 Declaring Types and Classes 119
10.1 Type declarations 119
10.2 Data declarations 120
10.3 Recursive types 122
10.4 Tautology checker 125
10.5 Abstract machine 130
10.6 Class and instance declarations 132
10.7 Chapter remarks 135
10.8 Exercises 135

11 The Countdown Problem 137
11.1 Introduction 137
11.2 Formalising the problem 138
11.3 Brute force solution 140
11.4 Combining generation and evaluation 142
11.5 Exploiting algebraic properties 143
11.6 Chapter remarks 144
11.7 Exercises 144

12 Lazy Evaluation 147
12.1 Introduction 147
12.2 Evaluation strategies 148
12.3 Termination 151
12.4 Number of reductions 152
12.5 Infinite structures 154
12.6 Modular programming 155
12.7 Strict application 158
12.8 Chapter remarks 161
12.9 Exercises 161

13 Reasoning About Programs **163**
13.1 Equational reasoning 163
13.2 Reasoning about Haskell 164
13.3 Simple examples 165
13.4 Induction on numbers 166
13.5 Induction on lists 169
13.6 Making append vanish 171
13.7 Compiler correctness 174
13.8 Chapter remarks 179
13.9 Exercises 179

A Standard Prelude **181**
A.1 Classes 181
A.2 Logical values 182
A.3 Characters and strings 183
A.4 Numbers 184
A.5 Tuples 185
A.6 Maybe 185
A.7 Lists 185
A.8 Functions 189
A.9 Input/Output 190

B Symbol Table **191**

Bibliography **192**

Index **196**

Preface

> ... there are two ways of constructing a software design: One way is to make it so simple that there are *obviously* no deficiencies and the other way is to make it so complicated that there are no *obvious* deficiencies. The first method is far more difficult.
>
> — Tony Hoare, 1980 ACM Turing Award Lecture

This book is about an approach to programming in which simplicity, clarity and elegance are the key goals. More specifically, it is an introduction to the functional style of programming, using the language Haskell.

The functional style is quite different to that promoted by most current languages, such as Java, C++, C, and Visual Basic. In particular, most current languages are closely linked to the underlying hardware, in the sense that programming is based upon the idea of changing stored values. In contrast, Haskell promotes a more abstract style of programming, based upon the idea of applying functions to arguments. As we shall see, moving to this higher-level leads to considerably simpler programs, and supports a number of powerful new ways to structure and reason about programs.

The book is primarily aimed at students studying computing science at university level, but is also appropriate for a broader spectrum of readers who would like to learn about programming in Haskell. No previous programming experience is required or assumed, and all the concepts are explained from first principles, with the aid of carefully chosen examples.

The version of Haskell used in the book is Haskell 98, the standard version of the language, for which the recently published definition is the culmination of fifteen years of work by its designers. As this is an introductory text, we do not attempt to cover all aspects of Haskell 98 and its associated libraries. Around half of the book is dedicated to introducing the main features of the language, while the other half comprises examples and case studies of programming of Haskell. Each chapter includes a series of exercises, and suggestions for further reading on more advanced and specialist topics.

The book is based upon course material that has been refined and class-tested over many years at the University of Nottingham. Most of the material from the book can be covered in twenty hours of lectures, supported by approximately forty hours of private study, practical sessions in a supervised

laboratory, and take-home programming courseworks. However, additional time would be required to cover some of the later chapters in more detail, along with some of the later programming examples.

The web site for the book, `www.cs.nott.ac.uk/~gmh/book.html`, provides a range of supporting material, including lecture slides for each chapter, and Haskell code for each of the extended examples. Instructors can also obtain model answers to the exercises for each chapter, together with a large collection of exam questions and their model answers.

Acknowledgements

The Foundations of Programming group at the University of Nottingham is an excellent environment in which to do research and teaching on functional programming. I am grateful to the University for providing a sabbatical to start work on this book; all the students and tutors on my Haskell courses for their feedback; and Thorsten Altenkirch, Mark Jones (now in Oregon), Conor McBride and Henrik Nilsson in the FOP group for our many discussions about functional ideas and how to present them.

I would also like to thank Rik van Geldrop and Jaap van der Woude for their feedback on using draft versions of the book; Ian Bayley for providing useful comments; Kees van den Broek, Frank Heitmann and Bill Tonkin for pointing out errors; Mark Jones for the Hugs interpreter for Haskell; Ralf Hinze and Andres Löh for the lhs2TeX system for typesetting Haskell; and Fritz Ruehr for producing the cover design.

Graham Hutton
Nottingham, Summer 2005

Chapter 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 history, and conclude with two 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:

$$\mathit{double}\ x \quad = \quad x + x$$

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:

$$\begin{array}{ll} & \mathit{double}\ 3 \\ = & \quad \{\ \text{applying}\ \mathit{double}\ \} \\ & 3 + 3 \\ = & \quad \{\ \text{applying}\ +\ \} \\ & 6 \end{array}$$

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

$$
\begin{array}{ll}
 & double\ (double\ 2) \\
= & \quad \{ \text{ applying the inner } double \ \} \\
 & double\ (2+2) \\
= & \quad \{ \text{ applying } + \ \} \\
 & double\ 4 \\
= & \quad \{ \text{ applying } double \ \} \\
 & 4+4 \\
= & \quad \{ \text{ applying } + \ \} \\
 & 8
\end{array}
$$

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

$$
\begin{array}{ll}
 & double\ (double\ 2) \\
= & \quad \{ \text{ applying the outer } double \ \} \\
 & double\ 2 + double\ 2 \\
= & \quad \{ \text{ applying the first } double \ \} \\
 & (2+2) + double\ 2 \\
= & \quad \{ \text{ applying the first } + \ \} \\
 & 4 + double\ 2 \\
= & \quad \{ \text{ applying } double \ \} \\
 & 4 + (2+2) \\
= & \quad \{ \text{ applying the second } + \ \} \\
 & 4+4 \\
= & \quad \{ \text{ applying } + \ \} \\
 & 8
\end{array}
$$

However, this calculation 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 may affect whether the calculation process terminates. These issues are explored in more detail in chapter 12.

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 most current programming languages, this would normally be achieved using two variables that store values that can be changed over time, one such variable used to count up to n, and the other used to accumulate the total.

For example, if we use the assignment symbol := to change the value of a variable, and the keywords **repeat** and **until** to repeatedly execute a sequence of instructions until a condition is satisfied, then the following sequence of instructions computes the required sum:

$count := 0$
$total := 0$
repeat
    $count := count + 1$
    $total := total + count$
until
    $count = n$

That is, we first initialise both the counter and the total to zero, and then repeatedly increment the counter and add this value to the total until the counter reaches n, at which point the computation stops.

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:

$$\begin{array}{lcl} count & := & 0 \\ total & := & 0 \\ count & := & 1 \\ total & := & 1 \\ count & := & 2 \\ total & := & 3 \\ count & := & 3 \\ total & := & 6 \\ count & := & 4 \\ total & := & 10 \\ count & := & 5 \\ total & := & 15 \end{array}$$

In general, programming languages 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 $[\,..\,]$ used to produce the list of numbers between one and n, and the other called *sum* 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 result is the required sum:

$$
\begin{aligned}
& \mathit{sum}\,[1\,..\,5] \\
= & \quad \{ \text{ applying } [\ ..\ ] \ \} \\
& \mathit{sum}\,[1,2,3,4,5] \\
= & \quad \{ \text{ applying } \mathit{sum} \ \} \\
& 1+2+3+4+5 \\
= & \quad \{ \text{ applying } + \ \} \\
& 15
\end{aligned}
$$

Most imperative languages support some form of programming with functions, so the Haskell program $\mathit{sum}\,[1\,..\,n]$ could be translated into such languages. However, most imperative languages do not *encourage* programming in the functional style. For example, many 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 the 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 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 current languages.

- **Powerful type system** (chapters 3 and 10)

  Most modern programming languages include some form of *type system* to detect incompatibility errors, such as attempting to add a number and a character. Haskell has a type system that 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 current languages, by allowing functions to be "polymorphic" and "overloaded".

- **List comprehensions** (chapter 5)

One of the most common ways to structure and manipulate data in computing is using lists. 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 non-trivial programs involve some form of repetition or looping. In Haskell, the basic mechanism by which looping is achieved is by using *recursive* functions that are defined in terms of themselves. Many computations have a simple and natural definition in terms of recursive functions, particularly 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, such as for list processing, parsing, and interactive programming.

- **Monadic effects** (chapters 8 and 9)

  Functions in Haskell are pure functions that take all their input as arguments and produce all their output 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 handling effects without compromising the purity of functions, based upon the mathematical notion of a *monad*.

- **Lazy evaluation** (chapter 12)

  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 data structures with an infinite number of elements, such as an infinite list of numbers.

- **Reasoning about programs** (chapter 13)

  Because programs in Haskell are pure functions, simple *equational reasoning* can be used to execute programs, to transform programs, to

prove properties of programs, and even to derive programs directly from specifications of their 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 main 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 adopted variable assignments 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 researchers initiated the development of Haskell (named after the logician Haskell Curry), a standard lazy functional programming language.

- In 2003, the committee published the Haskell Report, which defines a long-awaited stable version of Haskell, and is the culmination of fifteen years of work on the language by its designers.

It is worthy of note that three of the above researchers — 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 two small examples that give a taste of programming in Haskell. First of all, recall the function *sum* used earlier in this chapter, which produces the sum of a list of numbers. In Haskell, this function can be defined using the following two equations:

$$\begin{array}{lcl} \mathit{sum}\ [\,] & = & 0 \\ \mathit{sum}\ (x : \mathit{xs}) & = & x + \mathit{sum}\ \mathit{xs} \end{array}$$

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 x and a remaining list of numbers xs is given by adding x and the sum of xs. For example, the result of $\mathit{sum}\ [1, 2, 3]$ can be calculated as follows:

$$\begin{array}{ll} & \mathit{sum}\ [1,2,3] \\ = & \quad \{\ \text{applying}\ \mathit{sum}\ \} \\ & 1 + \mathit{sum}\ [2,3] \\ = & \quad \{\ \text{applying}\ \mathit{sum}\ \} \\ & 1 + (2 + \mathit{sum}\ [3]) \\ = & \quad \{\ \text{applying}\ \mathit{sum}\ \} \\ & 1 + (2 + (3 + \mathit{sum}\ [\,])) \\ = & \quad \{\ \text{applying}\ \mathit{sum}\ \} \\ & 1 + (2 + (3 + 0)) \\ = & \quad \{\ \text{applying}\ +\ \} \\ & 6 \end{array}$$

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. 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* has the following type:

$$\mathit{Num}\ a \Rightarrow [\,a\,] \rightarrow a$$

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

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.

Now let us consider a more interesting 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:

$$\begin{array}{lcl} \mathit{qsort}\;[\,] & = & [\,] \\ \mathit{qsort}\;(x : \mathit{xs}) & = & \mathit{qsort}\;\mathit{smaller} +\!\!+ [x] +\!\!+ \mathit{qsort}\;\mathit{larger} \\ & & \textbf{where} \\ & & \quad \mathit{smaller} = [a \mid a \leftarrow \mathit{xs}, a \leqslant x] \\ & & \quad \mathit{larger} = [b \mid b \leftarrow \mathit{xs}, b > x] \end{array}$$

In this definition, $+\!\!+$ is an operator that appends two lists; 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* that consists of all elements a from the list xs that are less than or equal to x, together with a list *larger* that consists of all elements b from xs that are greater than x. For example, if $x = 3$ and $\mathit{xs} = [5, 1, 4, 2]$, then $\mathit{smaller} = [1, 2]$ and $\mathit{larger} = [5, 4]$.

What does *qsort* actually do? First of all, we show that it has no effect on lists with a single element, in the sense that $\mathit{qsort}\;[x] = [x]$ for any x:

$$\begin{array}{ll} & \mathit{qsort}\;[x] \\ = & \quad \{\text{ applying } \mathit{qsort} \} \\ & \mathit{qsort}\;[\,] +\!\!+ [x] +\!\!+ \mathit{qsort}\;[\,] \\ = & \quad \{\text{ applying } \mathit{qsort} \} \\ & [\,] +\!\!+ [x] +\!\!+ [\,] \\ = & \quad \{\text{ applying } +\!\!+ \} \\ & [x] \end{array}$$

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

$$\begin{array}{ll} & \mathit{qsort}\;[3, 5, 1, 4, 2] \\ = & \quad \{\text{ applying } \mathit{qsort} \} \\ & \mathit{qsort}\;[1, 2] +\!\!+ [3] +\!\!+ \mathit{qsort}\;[5, 4] \\ = & \quad \{\text{ applying } \mathit{qsort} \} \\ & (\mathit{qsort}\;[\,] +\!\!+ [1] +\!\!+ \mathit{qsort}\;[2]) +\!\!+ [3] +\!\!+ (\mathit{qsort}\;[4] +\!\!+ [5] +\!\!+ \mathit{qsort}\;[\,]) \\ = & \quad \{\text{ applying } \mathit{qsort}\text{, above property} \} \\ & ([\,] +\!\!+ [1] +\!\!+ [2]) +\!\!+ [3] +\!\!+ ([4] +\!\!+ [5] +\!\!+ [\,]) \\ = & \quad \{\text{ applying } +\!\!+ \} \\ & [1, 2] +\!\!+ [3] +\!\!+ [4, 5] \\ = & \quad \{\text{ applying } +\!\!+ \} \\ & [1, 2, 3, 4, 5] \end{array}$$

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

$$qsort :: Ord\ a \Rightarrow [a] \rightarrow [a]$$

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.

1.6 Chapter remarks

The Haskell Report is freely available on the web from the Haskell home page, *www.haskell.org*, and has also been published as a book [26]. A more detailed historical account of the development of functional programming languages is given in Hudak's survey article [11].

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 $\leqslant$ by $<$ in the definition of *qsort*? Hint: consider the example $qsort\ [2, 2, 3, 1, 1]$.

Chapter 2

First Steps

In this chapter we take our first proper steps with Haskell. We start by introducing the Hugs 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 The Hugs system

As we saw in the previous chapter, small Haskell examples can be executed by hand. In practice, however, we usually require an implementation of Haskell that can execute programs automatically. In this book we use an interactive system called *Hugs*, which is the most widely used implementation of Haskell 98, the recently defined stable version of the language.

The interactive nature of Hugs makes it well suited for teaching and prototyping purposes, and its performance is sufficient for most applications. However, if greater performance or a stand-alone executable version of a Haskell program is required, a number of compilers for Haskell 98 are also available, of which the most widely used is the Glasgow Haskell Compiler.

2.2 The standard prelude

When the Hugs system is started it first loads a library file called *Prelude.hs*, and then displays a > prompt to indicate that the system is waiting for the user to enter an expression to be evaluated. For example, the library file defines many familiar functions that operate on integers, including the five main arithmetic operations of addition, subtraction, multiplication, division, and exponentiation, as illustrated below:

$>\ 2 + 3$
5

$> \ 2 - 3$
-1

$> \ 2 * 3$
6

$> \ 7 \text{ `}div\text{` } 2$
3

$> \ 2 \uparrow 3$
8

Note that the integer division operator is written as '*div*', and rounds down to the nearest integer if the result is a proper fraction.

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

In addition to functions on integers, the library file 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. 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:

  $> \; \mathit{drop}\ 3\ [1,2,3,4,5]$
  $[4,5]$

- Calculate the length of a list:

  $> \; \mathit{length}\ [1,2,3,4,5]$
  5

- Calculate the sum of a list of numbers:

  $> \; \mathit{sum}\ [1,2,3,4,5]$
  15

- Calculate the product of a list of numbers:

  $> \; \mathit{product}\ [1,2,3,4,5]$
  120

- Append two lists:

  $> \; [1,2,3] +\!\!+ [4,5]$
  $[1,2,3,4,5]$

- Reverse a list:

  $> \; \mathit{reverse}\ [1,2,3,4,5]$
  $[5,4,3,2,1]$

Some of the functions in the standard prelude may produce an error for certain values of their arguments. For example, attempting to divide by zero or select the first element of an empty list will produce an error:

$> \; 1\ \text{`}\mathit{div}\text{`}\ 0$
Error

$> \; \mathit{head}\ [\,]$
Error

In practice, when an error occurs the Hugs system also produces a message that provides some information about the likely cause, but these messages are often rather technical, and are not discussed further in this book.

For reference, appendix A presents some of the most commonly used definitions from the standard prelude, and appendix B shows how special Haskell symbols, such as $\uparrow$ and $+\!\!+$, are typed using a normal keyboard.

2.3 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 primary 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. For example, $f\ a + b$ means $(f\ a) + b$. The following table gives a few further examples to illustrate the differences between the notation for 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 $f\ x\ (g\ y)$.

2.4 Haskell scripts

As well as the functions provided in the standard prelude, it is also possible to define new functions. New functions cannot be defined at the $>$ prompt within Hugs, but must be defined within 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.

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 Hugs. 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*:

$$\begin{aligned} double\ x &= x + x \\ quadruple\ x &= double\ (double\ x) \end{aligned}$$

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

$> :load\ test.hs$

Now both *Prelude.hs* and *test.hs* are loaded, and functions from both scripts can be freely used. For example:

$>\ quadruple\ 10$
40

$>\ take\ (double\ 2)\ [1,2,3,4,5,6]$
$[1,2,3,4]$

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

$$\begin{aligned} factorial\ n &= product\ [1 \mathinner{..} n] \\ average\ ns &= sum\ ns\ \text{`}div\text{`}\ length\ ns \end{aligned}$$

We could equally well 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 ‘ ‘.

Hugs 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 below summarises the meaning of some of the most commonly used Hugs commands. Note that any command can be abbreviated by its first character. For example, :*load* can be abbreviated by :*l*. The command :*type* is explained in more detail in the next chapter.

Command	Meaning
:*load name*	load script *name*
:*reload*	reload current script
:*edit name*	edit script *name*
:*edit*	edit current script
:*type expr*	show type of *expr*
:?	show all commands
:*quit*	quit Hugs

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

$$myFun \qquad fun1 \qquad arg_2 \qquad x'$$

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**
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 list of characters might be named *css*.

The layout rule

When constructing a script, each definition 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

$$\begin{array}{lcl} a & = & b + c \\ & & \quad \textbf{where} \\ & & \qquad b = 1 \\ & & \qquad c = 2 \\ d & = & a * 2 \end{array}$$

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:

$$\begin{aligned}
&a = b + c \\
&\quad \textbf{where} \\
&\qquad \{\, b = 1; \\
&\qquad\ \ c = 2 \} \\
&d = a * 2
\end{aligned}$$

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

Comments

In addition to new definitions, scripts can also contain comments that will be ignored by Hugs. Haskell provides 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)
-}
```

2.5 Chapter remarks

The Hugs system is freely available on the web from the Haskell home page, *www.haskell.org*, which also contains a wealth of other useful resources.

2.6 Exercises

1. Parenthesise the following arithmetic expressions:

$$2 \uparrow 3 * 4$$
$$2 * 3 + 4 * 5$$
$$2 + 3 * 4 \uparrow 5$$

2. Work through the examples from this chapter using Hugs.

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

$$\begin{array}{lll} N & = & a \ \texttt{'div'} \ \mathit{length} \ \mathit{xs} \\ & & \textbf{where} \\ & & \quad a = 10 \\ & & \quad \mathit{xs} = [1,2,3,4,5] \end{array}$$

4. Show how the library function *last* that selects the last element of a non-empty list could be defined in terms of the library functions introduced in this chapter. Can you think of another possible definition?

5. Show how the library function *init* that removes the last element from a non-empty list could similarly be defined in two different ways.

Chapter 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 $Bool \rightarrow Bool$ contains all functions that map arguments from *Bool* to results from *Bool*, such as the logical negation function $\neg$. 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:

$$\begin{array}{lcl} False & :: & Bool \\ True & :: & Bool \\ \neg & :: & Bool \rightarrow Bool \end{array}$$

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

$$\begin{array}{lcl} \neg\ False & :: & Bool \\ \neg\ True & :: & Bool \\ \neg\ (\neg\ False) & :: & Bool \end{array}$$

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 a 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:

$$\frac{f :: A \rightarrow B \qquad e :: A}{f\ e :: B}$$

For example, the typing $\neg\ \mathit{False} :: \mathit{Bool}$ can be inferred from this rule using the fact that $\neg :: \mathit{Bool} \rightarrow \mathit{Bool}$ and $\mathit{False} :: \mathit{Bool}$. On the other hand, the expression $\neg\ 3$ does not have a type under the above rule for function application, because this would require that $3 :: \mathit{Bool}$, which is not valid because 3 is not a logical value. Expressions such as $\neg\ 3$ 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 free from type errors, but produces an error when evaluated because 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 Hugs, the type of any expression can be displayed by preceding the expression by the command :*type*. For example:

$> :\mathit{type}\ \neg$
$\neg :: \mathit{Bool} \rightarrow \mathit{Bool}$

$> :\mathit{type}\ \neg\ \mathit{False}$
$\neg\ \mathit{False} :: \mathit{Bool}$

$> :\mathit{type}\ \neg\ 3$
Error

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 that are available from a normal 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 is standard 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 computer memory being used for their storage. For example, the Hugs system has values of type *Int* in the range -2^{31} to $2^{31} - 1$. Going outside this range can give unexpected results. For example, evaluating $2 \uparrow 31 :: Int$ using Hugs (the use of :: forces the result to an *Int* rather than some other numeric type) gives a negative number as the result, which is incorrect.

Integer - arbitrary-precision integers

This type contains all integers, with as much memory are necessary being used for their storage, thus avoiding the imposition of lower and upper limits on the range of numbers. For example, evaluating $2 \uparrow 31 :: Integer$ 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 processing fixed-precision integers, whereas arbitrary-precision integers must usually be 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 3.14159, 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 magnitude of the number. For example, evaluating $sqrt\ 2 :: Float$ using Hugs gives the result 1.41421 (the library function *sqrt* calculates the square root of a number), which has five digits after the point, whereas $sqrt\ 99999 :: Float$ gives 316.226, which only has three digits after the point. Programming with floating-point numbers is a specialist topic that requires a careful treatment of rounding errors, and we say little more about such numbers in this introductory text.

We conclude this section by noting a single number may have more than one numeric type. For example, $3 :: Int$, $3 :: Integer$, and $3 :: Float$ are all valid typings for the number 3. This raises the question of what type such numbers

should be assigned during 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:

$$
\begin{array}{lcl}
[\mathit{False}, \mathit{True}, \mathit{False}] & :: & [\mathit{Bool}] \\
[\texttt{'a'}, \texttt{'b'}, \texttt{'c'}, \texttt{'d'}] & :: & [\mathit{Char}] \\
[\texttt{"One"}, \texttt{"Two"}, \texttt{"Three"}] & :: & [\mathit{String}]
\end{array}
$$

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 such as $[\mathit{False}]$ and $[\texttt{'a'}]$, 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.

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 $[\mathit{False}, \mathit{True}]$ and $[\mathit{False}, \mathit{True}, \mathit{False}]$ both have type $[\mathit{Bool}]$, 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:

$$[[\texttt{'a'}, \texttt{'b'}], [\texttt{'c'}, \texttt{'d'}, \texttt{'e'}]] \; :: \; [[\mathit{Char}]]$$

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 12.

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 $(T_1, T_2, \ldots, T_n)$ for the type of all tuples whose ith components have type T_i for any i in the range 1 to n. For example:

$$
\begin{array}{lcl}
(\mathit{False}, \mathit{True}) & :: & (\mathit{Bool}, \mathit{Bool}) \\
(\mathit{False}, \texttt{'a'}, \mathit{True}) & :: & (\mathit{Bool}, \mathit{Char}, \mathit{Bool}) \\
(\texttt{"Yes"}, \mathit{True}, \texttt{'a'}) & :: & (\mathit{String}, \mathit{Bool}, \mathit{Char})
\end{array}
$$

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 evaluation order explicit, such as in $(1 + 2) * 3$.

As with 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:

$$
\begin{array}{lcl}
(\texttt{'a'}, (\mathit{False}, \texttt{'b'})) & :: & (\mathit{Char}, (\mathit{Bool}, \mathit{Char})) \\
([\texttt{'a'}, \texttt{'b'}], [\mathit{False}, \mathit{True}]) & :: & ([\mathit{Char}], [\mathit{Bool}]) \\
[(\texttt{'a'}, \mathit{False}), (\texttt{'b'}, \mathit{True})] & :: & [(\mathit{Char}, \mathit{Bool})]
\end{array}
$$

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

3.5 Function types

A *function* is a mapping from arguments of one type to results of another type. We write $T1 \rightarrow T2$ for the type of all functions that map arguments of type *T1* to results of type *T2*. For example:

$$
\begin{array}{lcl}
\neg & :: & \mathit{Bool} \rightarrow \mathit{Bool} \\
\mathit{isDigit} & :: & \mathit{Char} \rightarrow \mathit{Bool}
\end{array}
$$

(The library function *isDigit* decides if a character is a numeric digit.) 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 result is already sufficient to handle 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:

$$
\begin{array}{lcl}
\mathit{add} & :: & (\mathit{Int}, \mathit{Int}) \rightarrow \mathit{Int} \\
\mathit{add}\ (x, y) & = & x + y \\
\mathit{zeroto} & :: & \mathit{Int} \rightarrow [\mathit{Int}] \\
\mathit{zeroto}\ n & = & [0 \mathinner{..} n]
\end{array}
$$

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 of a function is not defined. For example, the result of library function *head* that selects the first element of a list is undefined if the list is empty.

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:

$$\begin{array}{lcl} add' & :: & Int \rightarrow (Int \rightarrow Int) \\ add'\ x\ y & = & x + y \end{array}$$

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 $Int \rightarrow Int$. 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:

$$\begin{array}{lcl} add & :: & (Int, Int) \rightarrow Int \\ add' & :: & Int \rightarrow (Int \rightarrow Int) \end{array}$$

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, one at a time, and returns their product, can be defined as follows:

$$\begin{array}{lcl} mult & :: & Int \rightarrow (Int \rightarrow (Int \rightarrow Int)) \\ mult\ x\ y\ z & = & x * y * z \end{array}$$

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*. 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 is given by the partial application $add'\ 1 :: Int \rightarrow Int$ 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 $\rightarrow$ in types is assumed to associate to the right. For example,

$$Int \rightarrow Int \rightarrow Int \rightarrow Int$$

means

$$Int \rightarrow (Int \rightarrow (Int \rightarrow Int))$$

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

$$mult\ x\ y\ z$$

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.

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* $[isDigit, isLower, isUpper]$
3

The idea that the function *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:

$$length :: [a] \to Int$$

That is, for any type a, the function length has type $[a] \to Int$. A type that contains one or more type variables is called *polymorphic* ("of many forms"), as is an expression with such a type. Hence, $[a] \to Int$ 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:

$$\begin{array}{lll} fst & :: & (a, b) \to a \\ head & :: & [a] \to a \\ take & :: & Int \to [a] \to [a] \\ zip & :: & [a] \to [b] \to [(a, b)] \\ id & :: & a \to a \end{array}$$

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

$> \ 1.1 + 2.2$
3.3

The idea that the operator + 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 + is as follows:

$$(+) \ :: \ \mathit{Num} \ a \Rightarrow a \rightarrow a \rightarrow a$$

That is, for any type a that is a *instance* of the class *Num* of numeric types, the function $(+)$ has type $a \rightarrow a \rightarrow a$. (Parenthesising an operator converts it into a curried function, and is explained in more detail in the next chapter.) A type that contains one or more class constraints is called *overloaded*, as is an expression with such a type. Hence, $\mathit{Num} \ a \Rightarrow a \rightarrow a \rightarrow a$ is an overloaded type and $(+)$ is an overloaded function. More generally, most of the numeric functions provided in the standard prelude are overloaded. For example:

$$\begin{array}{lll} (-) & :: & \mathit{Num} \ a \Rightarrow a \rightarrow a \rightarrow a \\ (*) & :: & \mathit{Num} \ a \Rightarrow a \rightarrow a \rightarrow a \\ \mathit{negate} & :: & \mathit{Num} \ a \Rightarrow a \rightarrow a \\ \mathit{abs} & :: & \mathit{Num} \ a \Rightarrow a \rightarrow a \\ \mathit{signum} & :: & \mathit{Num} \ a \Rightarrow a \rightarrow a \end{array}$$

Moreover, numbers themselves are also overloaded. For example, $3 :: \mathit{Num} \ a \Rightarrow a$ means that for any numeric type a, the number 3 has type a.

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.

Eq - equality types

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

$$\begin{array}{lll} (==) & :: & a \rightarrow a \rightarrow \mathit{Bool} \\ (\neq) & :: & a \rightarrow a \rightarrow \mathit{Bool} \end{array}$$

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

$$\begin{array}{lll} (<) & :: & a \to a \to Bool \\ (\leqslant) & :: & a \to a \to Bool \\ (>) & :: & a \to a \to Bool \\ (\geqslant) & :: & a \to a \to Bool \\ min & :: & a \to a \to a \\ max & :: & a \to a \to a \end{array}$$

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

$$show \quad :: \quad a \rightarrow String$$

All the basic types *Bool*, *Char*, *String*, *Int*, *Integer*, and *Float* are instances of the *Show* class, as are list types and tuple types, provided that their element and component types are instances of the class. 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 converted from strings of characters using the following method:

$$read \quad :: \quad String \rightarrow a$$

All the basic types *Bool*, *Char*, *String*, *Int*, *Integer*, and *Float* are instances of the *Read* class, as are list types and tuple types, provided that their element and component types are instances of the class. 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. In practice, however, the necessary type information can usually be inferred automatically from the context. For example, the expression $\neg$ (*read* `"False"`) requires no explicit type information, because the application of the logical negation function $\neg$ 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 $\neg$ (*read* `"hello"`) produces an error when evaluated, because `"hello"` cannot be read as a logical value.

Num - numeric types

This class contains types that are instances of the equality class *Eq* and showable class *Show*, but in addition whose values are numeric, and as such can be processed using the following six methods:

$$\begin{array}{lll} (+) & :: & a \to a \to a \\ (-) & :: & a \to a \to a \\ (*) & :: & a \to a \to a \\ \mathit{negate} & :: & a \to a \\ \mathit{abs} & :: & a \to a \\ \mathit{signum} & :: & a \to a \end{array}$$

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

$> 1 + 2$
3

$> \; 1.1 + 2.2$
3.3

$> \; \mathit{negate} \; 3.3$
-3.3

$> \; \mathit{abs} \; (-3)$
3

$> \; \mathit{signum} \; (-3)$
-1

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:

$$\begin{array}{lll} \mathit{div} & :: & a \to a \to a \\ \mathit{mod} & :: & a \to a \to a \end{array}$$

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:

$> \; 7 \; `\mathit{div}` \; 2$
3

$> \; 7 \; `\mathit{mod}` \; 2$
1

For efficiency reasons, a number of prelude functions that involve both lists and integers (such as *length*, *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 *List.hs*.

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:

$$\begin{array}{lll} (/) & :: & a \to a \to a \\ \mathit{recip} & :: & a \to a \end{array}$$

The basic type *Float* is an instance of the *Fractional* class. 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. A more detailed account of the type system is given in Haskell Report [26], while formal descriptions for specialists can be found in [21, 6].

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. 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 if the functions are defined using overloaded operators.

3. Check your answers to the preceding two questions using Hugs.

4. 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.

Chapter 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 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 number of library functions that are defined in this way are shown below.

- Decide if a character is a digit:

$$\begin{array}{lcl} \mathit{isDigit} & :: & \mathit{Char} \rightarrow \mathit{Bool} \\ \mathit{isDigit}\ c & = & c \geqslant \texttt{'0'} \wedge c \leqslant \texttt{'9'} \end{array}$$

- Decide if an integer is even:

$$\begin{array}{lcl} \mathit{even} & :: & \mathit{Integral}\ a \Rightarrow a \rightarrow \mathit{Bool} \\ \mathit{even}\ n & = & n\ \text{`}\mathit{mod}\text{`}\ 2 == 0 \end{array}$$

- Split a list at the nth element:

$$\begin{array}{lcl} \mathit{splitAt} & :: & \mathit{Int} \rightarrow [\,a\,] \rightarrow ([\,a\,], [\,a\,]) \\ \mathit{splitAt}\ n\ \mathit{xs} & = & (\mathit{take}\ n\ \mathit{xs}, \mathit{drop}\ n\ \mathit{xs}) \end{array}$$

- Reciprocation:

$$\begin{array}{lcl} \mathit{recip} & :: & \mathit{Fractional}\ a \Rightarrow a \rightarrow a \\ \mathit{recip}\ n & = & 1\ /\ n \end{array}$$

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

$$\begin{array}{lcl} abs & :: & Int \rightarrow Int \\ abs\ n & = & \textbf{if}\ n \geqslant 0\ \textbf{then}\ n\ \textbf{else} - n \end{array}$$

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:

$$\begin{array}{lcl} signum & :: & Int \rightarrow Int \\ signum\ n & = & \textbf{if}\ n < 0\ \textbf{then} - 1\ \textbf{else} \\ & & \quad \textbf{if}\ n == 0\ \textbf{then}\ 0\ \textbf{else}\ 1 \end{array}$$

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 as follows:

$$\begin{array}{lcl} abs\ n \mid n \geqslant 0 & = & n \\ \quad\quad \mid otherwise & = & -n \end{array}$$

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 clearly avoiding the possibility that none of the guards in the sequence are *True*, which would 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:

$$\begin{array}{lll} signum\ n \mid n < 0 & = & -1 \\ \quad\quad\quad\ \ \mid n == 0 & = & 0 \\ \quad\quad\quad\ \ \mid otherwise & = & 1 \end{array}$$

4.4 Pattern matching

Many functions have a particularly 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 $\neg$ that returns the negation of a logical value is defined as follows:

$$\begin{array}{lll} \neg & :: & Bool \rightarrow Bool \\ \neg\ False & = & True \\ \neg\ True & = & False \end{array}$$

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 $\wedge$ that returns the conjunction of two logical values can be defined as follows:

$$\begin{array}{lll} (\wedge) & :: & Bool \rightarrow Bool \rightarrow Bool \\ True \wedge True & = & True \\ True \wedge False & = & False \\ False \wedge True & = & False \\ False \wedge False & = & False \end{array}$$

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:

$$\begin{array}{lll} True \wedge True & = & True \\ _ \wedge _ & = & False \end{array}$$

This version also has the benefit that, under lazy evaluation as discussed in chapter 12, 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 $\wedge$ using equations that have this same property, but make the choice about which equation applies using the value of the first argument only:

$$\begin{array}{lll} True \wedge b & = & b \\ False \wedge _ & = & False \end{array}$$

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 for technical reasons, the same name may not be used for more than one argument in a single equation. For example, the following definition for the operator $\wedge$ is based upon the observation that if the two arguments are equal then the result is the same value, otherwise the result is *False*, but is invalid because of the above naming requirement:

$$\begin{array}{lcl} b \wedge b & = & b \\ _ \wedge _ & = & \mathit{False} \end{array}$$

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

$$\begin{array}{lcl} b \wedge c \mid b == c & = & b \\ \quad\;\; \mid \mathit{otherwise} & = & \mathit{False} \end{array}$$

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 three 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 select the first and second components of a pair are defined as follows:

$$\begin{array}{lcl} \mathit{fst} & :: & (a, b) \rightarrow a \\ \mathit{fst}\ (x, _) & = & x \\ \mathit{snd} & :: & (a, b) \rightarrow b \\ \mathit{snd}\ (_, y) & = & y \end{array}$$

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 `'a'` can be defined as follows:

$$\begin{array}{lcl} \mathit{test} & :: & [\mathit{Char}] \rightarrow \mathit{Bool} \\ \mathit{test}\ [\texttt{'a'}, _, _] & = & \mathit{True} \\ \mathit{test}\ _ & = & \mathit{False} \end{array}$$

Up to this point we have viewed lists as a primitive notion in Haskell. In fact they are not primitive as such, but are actually 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:

$$
\begin{array}{ll}
 & [1,2,3] \\
= & \quad \{ \text{ list notation } \} \\
 & 1:[2,3] \\
= & \quad \{ \text{ list notation } \} \\
 & 1:(2:[3]) \\
= & \quad \{ \text{ list notation } \} \\
 & 1:(2:(3:[\,]))
\end{array}
$$

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 `'a'`:

$$
\begin{array}{lcl}
\mathit{test} & :: & [\mathit{Char}] \rightarrow \mathit{Bool} \\
\mathit{test}\ (\texttt{'a'} : _) & = & \mathit{True} \\
\mathit{test}\ _ & = & \mathit{False}
\end{array}
$$

Similarly, the library functions *null*, *head* and *tail* that decide if a list is empty, select the first element of a non-empty list, and remove the first element of a non-empty list are defined as follows:

$$
\begin{array}{lcl}
\mathit{null} & :: & [a] \rightarrow \mathit{Bool} \\
\mathit{null}\ [\,] & = & \mathit{True} \\
\mathit{null}\ (_ : _) & = & \mathit{False} \\
\mathit{head} & :: & [a] \rightarrow a \\
\mathit{head}\ (x : _) & = & x \\
\mathit{tail} & :: & [a] \rightarrow [a] \\
\mathit{tail}\ (_ : \mathit{xs}) & = & \mathit{xs}
\end{array}
$$

Note that cons patterns must be parenthesised, because function application has higher priority than all other operators. For example, the definition $\mathit{tail}\ _ : \mathit{xs} = \mathit{xs}$ without parentheses means $(\mathit{tail}\ _) : \mathit{xs} = \mathit{xs}$, which is both the incorrect meaning and an invalid definition.

Integer patterns

As a special case that is sometimes useful, Haskell also allows integer patterns of the form $n{+}k$, where n is an integer variable and $k{>}0$ is an integer constant. For example, a function *pred* that maps zero to itself and any strictly positive integer to its predecessor can be defined as follows:

$$
\begin{array}{lcl}
\mathit{pred} & :: & \mathit{Int} \rightarrow \mathit{Int} \\
\mathit{pred}\ 0 & = & 0 \\
\mathit{pred}\ (n+1) & = & n
\end{array}
$$

There are two points to note about $n + k$ patterns. First of all, they only match integers $\geqslant k$. For example, evaluating $\mathit{pred}\ (-1)$ produces an error, because neither of the two patterns in the definition for *pred* matches negative integers. Secondly, for the same reason as cons patterns, integer patterns must be parenthesised. For example, the definition $\mathit{pred}\ n + 1 = n$ without parentheses means $(\mathit{pred}\ n) + 1 = n$, which is 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, a nameless function that takes a single number x as its argument, and produces the result $x + x$, can be constructed as follows:

$$\lambda x \rightarrow x + x$$

The symbol λ is the lower-case Greek letter "lambda". 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:

$$> \ (\lambda x \rightarrow x + x)\ 2$$
$$4$$

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

$$\mathit{add}\ x\ y \ = \ x + y$$

can be understood as meaning

$$\mathit{add} \ = \ \lambda x \rightarrow (\lambda y \rightarrow x + y)$$

which makes precise that *add* is a function that takes a number x and returns a function, which in turn takes a number y and returns the result $x + y$.

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:

$$\begin{aligned} \mathit{const} \quad &:: \quad a \rightarrow b \rightarrow a \\ \mathit{const}\ x\ _ \quad &= \quad x \end{aligned}$$

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:

$$\begin{array}{lcl} \mathit{const} & :: & a \rightarrow (b \rightarrow a) \\ \mathit{const}\ x & = & \lambda_ \rightarrow x \end{array}$$

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

$$\begin{array}{lcl} \mathit{odds} & :: & \mathit{Int} \rightarrow [\mathit{Int}] \\ \mathit{odds}\ n & = & \mathit{map}\ f\ [0\mathinner{..}n-1] \\ & & \quad \mathbf{where}\ f\ x = x * 2 + 1 \end{array}$$

(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:

$$\mathit{odds}\ n \quad = \quad \mathit{map}\ (\lambda x \rightarrow x * 2 + 1)\ [0\mathinner{..}n-1]$$

4.6 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 $\oplus$ is an operator then expressions of the form $(\oplus)$, $(x\ \oplus)$ and $(\oplus\ y)$ for arguments x and y are called *sections*, whose meaning as functions can be formalised using lambda expressions as follows:

$$\begin{array}{lcl} (\oplus) & = & \lambda x \rightarrow (\lambda y \rightarrow x \ \oplus\ y) \\ (x\ \oplus) & = & \lambda y \rightarrow x \ \oplus\ y \\ (\oplus\ y) & = & \lambda x \rightarrow x \ \oplus\ y \end{array}$$

Sections have three main 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 $\lambda x \rightarrow (\lambda y \rightarrow x + y)$
(1+)	is the successor function $\lambda y \rightarrow 1 + y$
(1/)	is the reciprocation function $\lambda y \rightarrow 1\ /\ y$
(∗2)	is the doubling function $\lambda x \rightarrow x * 2$
(/2)	is the halving function $\lambda x \rightarrow x\ /\ 2$

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 logical conjunction operator $\wedge$ is stated as follows:

$$(\wedge) \quad :: \quad \mathit{Bool} \rightarrow \mathit{Bool} \rightarrow \mathit{Bool}$$

Finally, sections are also necessary when using operators as arguments to other functions. For example, the library function *and* that decides if all logical values in a list are *True* is defined by using the operator $\wedge$ as an argument to the library function *foldr*, which is itself discussed in chapter 7:

$$\begin{aligned} \mathit{and} \quad &:: \quad [\mathit{Bool}] \rightarrow \mathit{Bool} \\ \mathit{and} \quad &= \quad \mathit{foldr}\ (\wedge)\ \mathit{True} \end{aligned}$$

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 [26]. The Greek letter λ used when defining nameless functions comes from the *lambda calculus*, the mathematical theory of functions upon which Haskell is founded.

4.8 Exercises

1. Using library functions, define a function $\mathit{halve} :: [a] \rightarrow ([a],[a])$ that splits an even-lengthed list into two halves. For example:

   $> \mathit{halve}\ [1,2,3,4,5,6]$
   $([1,2,3],[4,5,6])$

2. Consider a function $\mathit{safetail} :: [a] \rightarrow [a]$ that behaves as the library function *tail*, except that *safetail* maps the empty list to itself, whereas *tail* produces an error in this case. Define *safetail* using:

   (a) a conditional expression;
   (b) guarded equations;
   (c) pattern matching.

   Hint: make use of the library function *null*.

3. In a similar way to $\wedge$, show how the logical disjunction operator $\vee$ can be defined in four different ways using pattern matching.

4. Redefine the following version of the conjunction operator using conditional expressions rather than pattern matching:

$$\begin{aligned} \mathit{True} \wedge \mathit{True} \quad &= \quad \mathit{True} \\ _ \wedge _ \quad &= \quad \mathit{False} \end{aligned}$$

5. Do the same for the following version, and note the difference in the number of conditional expressions required:

$$\begin{array}{lcl} \mathit{True} \wedge b & = & b \\ \mathit{False} \wedge _ & = & \mathit{False} \end{array}$$

6. Show how the curried function definition $\mathit{mult}\ x\ y\ z = x * y * z$ can be understood in terms of lambda expressions.

Chapter 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 Generators

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:

$$> \ [x \uparrow 2 \mid x \leftarrow [1 \mathinner{..} 5]]$$
$$[1, 4, 9, 16, 25]$$

The symbols $\mid$ and $\leftarrow$ are read as "such that" and "is drawn from" respectively, and the expression $x \leftarrow [1 \mathinner{..} 5]$ 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 $[4, 5]$ can be produced as follows:

$$> \ [(x, y) \mid x \leftarrow [1, 2, 3], y \leftarrow [4, 5]]$$
$$[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]$$

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

$$> \ [(x, y) \mid y \leftarrow [4, 5], x \leftarrow [1, 2, 3]]$$
$$[(1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5)]$$

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:

$$> \ [(x, y) \mid x \leftarrow [1\,..\,3], y \leftarrow [x\,..\,3]]$$
$$[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]$$

As another 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:

$$\begin{aligned} &\mathit{concat} && :: && [[a]] \rightarrow [a] \\ &\mathit{concat}\ \mathit{xss} && = && [x \mid \mathit{xs} \leftarrow \mathit{xss}, x \leftarrow \mathit{xs}] \end{aligned}$$

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:

$$\begin{aligned} &\mathit{firsts} && :: && [(a, b)] \rightarrow [a] \\ &\mathit{firsts}\ \mathit{ps} && = && [x \mid (x, _) \leftarrow \mathit{ps}] \end{aligned}$$

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:

$$\begin{aligned} &\mathit{length} && :: && [a] \rightarrow \mathit{Int} \\ &\mathit{length}\ \mathit{xs} && = && \mathit{sum}\ [1 \mid _ \leftarrow \mathit{xs}] \end{aligned}$$

In this case, the generator $_ \leftarrow \mathit{xs}$ 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, otherwise they are discarded. For example, the comprehension $[x \mid x \leftarrow [1\,..\,10], \mathit{even}\ x]$ 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 as follows:

$$\begin{aligned} &\mathit{factors} && :: && \mathit{Int} \rightarrow [\mathit{Int}] \\ &\mathit{factors}\ n && = && [x \mid x \leftarrow [1\,..\,n], n\ \text{`}\mathit{mod}\text{`}\ x == 0] \end{aligned}$$

For example:

$> factors\ 15$
$[1, 3, 5, 15]$

$> factors\ 7$
$[1, 7]$

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

$$\begin{array}{lcl} prime & :: & Int \rightarrow Bool \\ prime\ n & = & factors\ n == [1, n] \end{array}$$

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:

$$\begin{array}{lcl} primes & :: & Int \rightarrow [Int] \\ primes\ n & = & [x \mid x \leftarrow [2 \mathinner{..} n], prime\ x] \end{array}$$

For example:

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

In chapter 12 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.

As a final example concerning guards, suppose that we represent a lookup table by a list of pairs comprising keys and values. Then for any type of keys that is an equality type, 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:

$$\begin{array}{lcl} find & :: & Eq\ a \Rightarrow a \rightarrow [(a, b)] \rightarrow [b] \\ find\ k\ t & = & [v \mid (k', v) \leftarrow t, k == k'] \end{array}$$

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 are exhausted. For example:

$> \; zip \; [\texttt{'a'}, \texttt{'b'}, \texttt{'c'}] \; [1,2,3,4]$
$[(\texttt{'a'},1),(\texttt{'b'},2),(\texttt{'c'},3)]$

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

$$\begin{array}{lcl} pairs & :: & [a] \to [(a,a)] \\ pairs\;xs & = & zip\;xs\;(tail\;xs) \end{array}$$

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:

$$\begin{array}{lcl} sorted & :: & Ord\;a \Rightarrow [a] \to Bool \\ sorted\;xs & = & and\;[x \leqslant y \mid (x,y) \leftarrow pairs\;xs] \end{array}$$

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:

$$\begin{array}{lcl} positions & :: & Eq\;a \Rightarrow a \to [a] \to [Int] \\ positions\;x\;xs & = & [i \mid (x',i) \leftarrow zip\;xs\;[0\,..\,n], x == x'] \\ & & \textbf{where}\;n = length\;xs - 1 \end{array}$$

For example:

$> \; positions \; False \; [True, False, True, False]$
$[1,3]$

5.4 String comprehensions

Up to this point we have viewed strings as a primitive notion in Haskell. In fact they are not primitive as such, but are actually constructed as lists of characters. For example, `"abc"` :: *String* is just an abbreviation for [`'a'`, `'b'`, `'c'`] :: [*Char*]. Because strings are just special kinds of 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, isLower x]
count       :: Char → String → Int
count x xs  =  length [x′ | x′ ← xs, x == x′]
```

For example:

```
> lowers "Haskell␣98"
6

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

5.5 The Caesar cipher

We conclude this chapter with an extended example. Consider the problem of encoding a string in order to disguise its contents from unintended readers. A well-known encoding method is the *Caesar cipher*, named after its use by Julius Caesar. 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 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

```
"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.

Encoding and decoding

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 $ord :: Char \rightarrow Int$ and $chr :: Int \rightarrow Char$ convert between characters and their Unicode representation as integers.) 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
```

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

$$
\begin{array}{lcl}
encode & :: & Int \to String \to String \\
encode\ n\ xs & = & [shift\ n\ x \mid x \leftarrow xs]
\end{array}
$$

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

Some letters in English are used more frequently than others. By analysing a large volume of text, one can derive the following table of approximate percentage frequencies of the twenty-six letters of alphabet:

$$
\begin{array}{lcl}
table & :: & [Float] \\
table & = & [8.2, 1.5, 2.8, 4.3, 12.7, 2.2, 2.0, 6.1, 7.0, 0.2, 0.8, 4.0, 2.4, \\
 & & \ 6.7, 7.5, 1.9, 0.1, 6.0, 6.3, 9.1, 2.8, 1.0, 2.4, 0.2, 2.0, 0.1]
\end{array}
$$

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:

$$
\begin{array}{lcl}
percent & :: & Int \to Int \to Float \\
percent\ n\ m & = & (fromInt\ n\ /\ fromInt\ m) * 100
\end{array}
$$

(The library function *fromInt* :: *Int* → *Float* converts an integer into the corresponding floating-point number.) Using *percent* within a string comprehension, together with the functions *lowers* and *count* from the previous section, we can define a function that returns the frequency table for any string:

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

For example:

```
> freqs "abbcccddddeeeee"
[6.7, 13.3, 20.0, 26.7, 33.3, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ···, 0.0]
```

That is, the letter `'a'` occurs with a frequency of 6.7%, 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 string 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 formulae 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 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 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 $table' = freqs$ "kdvnhoo␣lv␣ixq" then

$$[\,chisqr\ (rotate\ n\ table')\ table \mid n \leftarrow [0\,..\,25]\,]$$

gives the result

$$[1408.8, 640.3, 612.4, 202.6, 1439.8, 4247.2, 651.3, \cdots, 626.7]$$

in which the minimum value, 202.6, appears at position three in this list. Hence, we conclude that three is the most likely shift factor that can be used to decode 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 "the␣five␣boxing␣wizards␣jump␣quickly")
"dro␣psfo␣lyhsxq␣gsjkbnc␣tewz␣aesmuvi"
```

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 satisfying a particular property. A formal meaning for list comprehensions by translation using more primitive features of the language is given in the Haskell Report [26]. A popular account of the Caesar cipher, and many other famous cryptographic methods, is given in *The Code Book* [30].

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. In a similar way to the function *length*, show how the library function $replicate :: Int \rightarrow a \rightarrow [a]$ that produces a list of identical elements can be defined using a list comprehension. For example:

   $>$ *replicate* 3 *True*
   [*True*, *True*, *True*]

3. A triple (x, y, z) of positive integers can be termed *pythagorean* if $x^2 + y^2 = z^2$. Using a list comprehension, define a function $pyths :: Int \rightarrow [(Int, Int, Int)]$ that returns the list of all pythagorean triples whose components are at most a given limit. For example:

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

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

   $>$ *perfects* 500
   $[6,28,496]$

5. Show how the single comprehension $[(x, y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5,6]]$ with two generators can be re-expressed using two comprehensions with single generators. Hint: make use of the library function *concat* and nest one comprehension within the other.

6. Redefine the function *positions* using the function *find*.

7. 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 the function *chisqr*, show how a list comprehension can be used to define a function $scalarproduct :: [Int] \to [Int] \to Int$ that returns the scalar product of two lists. For example:

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

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