Recent changes:

- 27 Jul 2004: revised and extended chapter 5 on list comprehensions.
- 3 Aug 2004: completed chapter 7 on higher-order functions.
- 10 Jan 2005: drafted part of chapter 13 on reasoning about programs.
To Annette, Callum and Tom
Contents

0 Preface 9

1 Introduction 11
  1.1 Functions ........................................... 11
  1.2 Functional programming ................................ 12
  1.3 Features of Haskell ....................................... 14
  1.4 Historical background .................................... 16
  1.5 A taste of Haskell ....................................... 17
  1.6 Chapter remarks .......................................... 19
  1.7 Exercises ................................................ 19

2 First Steps 21
  2.1 The Hugs system ........................................ 21
  2.2 The standard prelude ..................................... 21
  2.3 Function application .................................... 24
  2.4 Haskell scripts ........................................ 24
    2.4.1 My first script .................................... 24
    2.4.2 Naming requirements .................................. 26
    2.4.3 The layout rule ..................................... 26
    2.4.4 Comments ........................................... 27
  2.5 Chapter remarks .......................................... 27
  2.6 Exercises ................................................ 27

3 Types and Classes 29
  3.1 Basic concepts .......................................... 29
  3.2 Basic types ............................................ 30
  3.3 List types ................................................ 32
  3.4 Tuple types ............................................. 32
  3.5 Function types .......................................... 33
  3.6 Curried functions ....................................... 34
  3.7 Polymorphic types ...................................... 35
  3.8 Overloaded types ....................................... 36
  3.9 Basic classes ............................................ 36
  3.10 Chapter remarks ......................................... 41
  3.11 Exercises ............................................... 41
4 Defining Functions
4.1 New from old ........................................ 43
4.2 Conditional expressions .............................. 44
4.3 Guarded equations ................................... 44
4.4 Pattern matching ..................................... 45
   4.4.1 Tuple patterns .................................... 46
   4.4.2 List patterns ...................................... 46
   4.4.3 Integer patterns .................................. 47
4.5 Lambda expressions .................................. 48
4.6 Sections .............................................. 49
4.7 Chapter remarks ..................................... 50
4.8 Exercises ............................................ 50

5 List Comprehensions ................................. 53
5.1 Generators ........................................... 53
5.2 Guards ................................................ 54
5.3 The zip function .................................... 56
5.4 String comprehensions .............................. 57
5.5 The Caesar cipher .................................. 57
   5.5.1 Encoding and decoding ......................... 58
   5.5.2 Frequency tables ................................. 59
   5.5.3 Cracking the cipher ............................. 60
5.6 Chapter remarks ..................................... 61
5.7 Exercises ............................................ 62

6 Recursive Functions ............................... 65
6.1 Basic concepts ...................................... 65
6.2 Recursion on lists .................................. 66
6.3 Multiple arguments ................................ 69
6.4 Multiple recursion ................................ 70
6.5 Mutual recursion .................................. 71
6.6 Advice on recursion ................................. 72
6.7 Chapter remarks ..................................... 77
6.8 Exercises ............................................ 77

7 Higher-Order Functions ........................... 79
7.1 Basic concepts ...................................... 79
7.2 Processing lists .................................... 80
7.3 The foldr function ................................ 82
7.4 The foldl function ................................ 85
7.5 The composition operator ......................... 87
7.6 String transmitter ................................ 88
   7.6.1 Binary numbers ................................. 88
   7.6.2 Base conversion ................................ 88
   7.6.3 Transmission .................................... 90
7.7 Chapter remarks ..................................... 91
7.8 Exercises .............................................. 91

8 Functional Parsers .................................. 93
  8.1 Parsers ............................................. 93
  8.2 The parser type ................................... 94
  8.3 Basic parsers ..................................... 94
  8.4 Sequencing ....................................... 96
  8.5 Choice ............................................ 97
  8.6 Derived primitives ............................... 97
  8.7 Handling spacing ................................. 100
  8.8 Arithmetic expressions ......................... 101
  8.9 Chapter remarks ................................ 105
  8.10 Exercises ....................................... 105

9 Interactive Programs ............................... 107
  9.1 Interaction ....................................... 107
  9.2 The input/output type ......................... 108
  9.3 Basic actions .................................... 109
  9.4 Sequencing ....................................... 109
  9.5 Derived primitives ............................... 110
  9.6 Desktop calculator ............................... 112
  9.7 Game of life ..................................... 114
  9.8 Chapter remarks ................................ 117
  9.9 Exercises ....................................... 118

10 Defining Types and Classes ...................... 119

11 The Countdown Problem .......................... 121
  11.1 Introduction .................................... 121
  11.2 Formalising the problem ....................... 122
  11.3 Brute force solution ......................... 124
  11.4 Combining generation and evaluation ........ 126
  11.5 Exploiting algebraic properties ............... 127
  11.6 Chapter remarks ................................ 128
  11.7 Exercises ....................................... 128

12 Lazy Evaluation .................................. 131
  12.1 Introduction .................................... 131
  12.2 Innermost evaluation ......................... 133
  12.3 Outermost evaluation ......................... 134
  12.4 Lambda expressions ............................ 134
  12.5 Termination ..................................... 135
  12.6 Number of reductions ......................... 136
  12.7 Infinite structures ............................ 138
  12.8 Modular programming ........................... 139
  12.9 Strict application ............................. 142
  12.10 Chapter remarks .............................. 145
Chapter 0

Preface

This book is an introduction to the functional style of computer programming, using the modern functional language Haskell. The functional style is quite different to that promoted by most current programming languages, such as Visual Basic, C, C++ and Java. In particular, most current languages are closely linked to the underlying computer 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 or mathematics at university level, but may also be of interest to a broader spectrum of readers who would like to learn about programming in Haskell. No previous programming experience is assumed, but some experience with the basic concepts of discrete mathematics — in particular, sets, functions, propositional logic and predicate logic — will be helpful. However, all the concepts required are introduced as they are needed.

The version of Haskell used in this book is Haskell 98, the recently defined stable version of the language that is the culmination of fifteen years of work by its designers. Haskell itself will continue to evolve, but implementors for the language are committed to supporting Haskell 98 for the foreseeable future. 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 volume of the text is dedicated to introducing the main features of the language, while the other half comprises examples and case studies of programming of Haskell.

For lecturers interested in teaching students how to program in Haskell using this book, lecture slides based upon each chapter are freely available from the author. Most of the material could be covered in around twenty hours of lectures, supported by a total of around forty hours of private study, practical sessions in a supervised laboratory, and take-home programming courseworks. However, additional time would be required to study some of the later chapters in more detail, along with some of the later case studies.
Acknowledgements

In preparation.
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

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 single number $x$ as its argument and produces the result $x + x$ can be defined by the following equation:

$$\text{double } x = x + x$$

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

$$\begin{align*}
\text{double } 3 \\
&= \{ \text{applying } \text{double } \} \\
&= \{ \text{applying } + \} \\
&= 6
\end{align*}$$

Similarly, the result of the nested application double (double 2) in which the function double is applied twice can be calculated as follows:
\(\text{double (double 2)}\)
\[
= \{ \text{applying the inner double} \}
\]
\(\text{double (2 + 2)}\)
\[
= \{ \text{applying +} \}
\]
\(\text{double 4}\)
\[
= \{ \text{applying double} \}
\]
\(4 + 4\)
\[
= \{ \text{applying +} \}
\]
\(8\)

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

\(\text{double (double 2)}\)
\[
= \{ \text{applying the outer double} \}
\]
\((\text{double 2}) + (\text{double 2})\)
\[
= \{ \text{applying the first double} \}
\]
\((2 + 2) + (\text{double 2})\)
\[
= \{ \text{applying the first +} \}
\]
\(4 + (\text{double 2})\)
\[
= \{ \text{applying double} \}
\]
\(4 + (2 + 2)\)
\[
= \{ \text{applying the second +} \}
\]
\(4 + 4\)
\[
= \{ \text{applying +} \}
\]
\(8\)

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 \textbf{repeat} and \textbf{until} to repeatedly execute a sequence of instructions until a condition is satisfied, then the following sequence of instructions computes the required sum:

\begin{verbatim}
  count := 0
  total := 0
  repeat
    count := count + 1
    total := total + count
  until
  count = n
\end{verbatim}

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 \textit{total} is the required sum:

\begin{verbatim}
  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{verbatim}

In general, programming languages in which the basic method of computation is changing stored values are called \textit{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 \([\ldots]\) used to produce the list of numbers between one and \(n\), and the other called \textit{sum} used to produce the sum of this list:

\[
\textit{sum} \ [1 \ldots n]
\]

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:

\[
\textit{sum} \ [1 \ldots 5]
\]
Most imperative languages support some form of programming with functions, so the Haskell program \( \text{sum} \ [1..n] \) 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, interactive programming, and parsing.

- **Monadic effects** (chapters 8 and 9)
  Functions in Haskell are pure functions that take all their inputs as arguments and produce all their outputs as results. However, most real-life programs require some form of side effect that would appear to be at odds with purity, such as reading data from files, interacting with the user, or changing stored values. Haskell provides a uniform framework for handling side 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 \textit{lambda calculus}, a simple
  but powerful mathematical theory of functions.

- In the 1950s, John McCarthy developed \textit{Lisp} (“LIST Processor”), gener-
  ally 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 \textit{ISWIM} (“If you See What I
  Mean”), the first purely functional programming language, based strongly
  on the lambda calculus and having no variable assignments.

- In the 1970s, John Backus developed \textit{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 \textit{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 func-
  tional programming languages, culminating in the commercially pro-
  duced language \textit{Miranda} (meaning “admirable”).

- In 1987, an international committee of researchers initiated the develop-
  ment of \textit{Haskell} (named after the logician Haskell Curry), a standard
  lazy functional programming language.

- In 2003, the committee published the definition of \textit{Haskell 98}, a stable
  version of Haskell that is the culmination of fifteen years of revisions and
  extensions to 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 \textit{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:

\[
\begin{align*}
\text{sum} \; [] & = 0 \\
\text{sum} \; (x : xs) & = x + \text{sum} \; xs
\end{align*}
\]

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 \(\text{sum} \; [1, 2, 3]\) can be calculated as follows:

\[
\begin{align*}
\text{sum} \; [1, 2, 3] \\
&= \{ \text{applying sum} \}
1 + \text{sum} \; [2, 3] \\
&= \{ \text{applying sum} \}
1 + (2 + \text{sum} \; [3]) \\
&= \{ \text{applying sum} \}
1 + (2 + (3 + \text{sum} \; [])) \\
&= \{ \text{applying sum} \}
1 + (2 + (3 + 0)) \\
&= \{ \text{applying +} \}
6
\end{align*}
\]

Note that even though the function \textit{sum} is defined in terms of itself and is hence recursive, it does not loop forever. In particular, each application of \textit{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 \textit{identity} for addition. That is, \(0 + x = x\) and \(x + 0 = x\) for any number \(x\).

In Haskell, every function has a \textit{type} that specifies the nature of its arguments and results, which is automatically inferred from the definition of the function. For example, the function \textit{sum} has the following type:

\[
\text{Num} \; a \Rightarrow [a] \rightarrow a
\]

This type states that for any type \(a\) of numbers, \textit{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, rationals such as \(\frac{2}{3}\), and “floating-point” numbers such as 3.14159. Hence, for example, \textit{sum} could be applied to a list of integers to produce another integer, as in the calculation above, or it could be applied to a list of rationals to produce another rational.

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:

\[
\begin{align*}
quartesort \; [] & = [] \\
quartesort \; (x : xs) & = \text{qsort smaller} \; ++ \; [x] \; ++ \; \text{qsort larger} \\
\text{where} & \\
\text{smaller} & = \{ a \mid a \leftarrow xs, a \leq x \} \\
\text{larger} & = \{ b \mid b \leftarrow xs, b > x \}
\end{align*}
\]

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

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

\[
\begin{align*}
quartesort \; [x] \\
& = \{ \text{applying \text{qsort} } \} \\
quartesort \; [] + [x] + qsort \; [] \\
& = \{ \text{applying qsort } \} \\
[] + [x] + [] \\
& = \{ \text{applying } + \} \\
[x]
\end{align*}
\]

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

\[
\begin{align*}
quartesort \; [3, 5, 1, 4, 2] \\
& = \{ \text{applying qsort } \} \\
quartesort \; [1, 2] + [3] + qsort \; [5, 4] \\
& = \{ \text{applying qsort } \} \\
& = \{ \text{applying } + \} \\
& = \{ \text{applying } + \} \\
[1, 2] + [3] + [4, 5] \\
& = \{ \text{applying } + \} \\
[1, 2, 3, 4, 5]
\end{align*}
\]
In summary, \textit{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 \textit{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 \textit{smaller} and \textit{larger} than this number. This method of sorting is called \textit{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 \textit{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

\[ qsort :: \text{Ord} \ a \Rightarrow [a] \rightarrow [a] \]

states that for any type \( a \) of ordered values, \textit{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 "\texttt{abcde}". Hence, for example, the function \textit{qsort} could also be used to sort a list of characters, or a list of strings.

\section{Chapter remarks}

The definition of Haskell 98 is freely available on the web from the Haskell home page, \texttt{www.haskell.org}, and has also been published as a book [18]. A more detailed historical account of the development of functional programming languages is given in Hudak’s survey article [7].

\section{Exercises}

1. Give another possible calculation for the result of \textit{double} (\textit{double} 2).

2. Show that \textit{sum} \([x] = x\) for any number \( x \).

3. Define a function \textit{product} that produces the product of a list of numbers, and show using your definition that \textit{product} \([2, 3, 4] = 24\).

4. How should the definition of the function \textit{qsort} be modified so that it produces a \textit{reverse} sorted version of a list?

5. What would be the effect of replacing \( \leq \) by \( < \) in the original definition of \textit{qsort}? Hint: consider the example \textit{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 many applications. However, if greater performance or a stand-alone executable version of a Haskell program is required, a number of optimising compilers for Haskell 98 are 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 \texttt{'div'} 2
3

> 2 ↑3
8

Note that the integer division operator is written as \texttt{'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, \(2 ∗ 3↑4\) means \(2 ∗ (3↑4)\), 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, \(2 ↑3 ↑4\) means \(2 ↑ (3 ↑ 4)\), while \(2 - 3 + 4\) means \((2 - 3) + 4\). In practice, however, it is often clearer to use explicit parentheses in 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:

  > \texttt{head \{1, 2, 3, 4, 5\}}
  1

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

  > \texttt{tail \{1, 2, 3, 4, 5\}}
  \{2, 3, 4, 5\}

- Select the \(n\)th element of list (counting from zero):

  > \{1, 2, 3, 4, 5\} ++ 2
  3

- Select the first \(n\) elements of a list:

  > \texttt{take \(3\) \{1, 2, 3, 4, 5\}}
  \{1, 2, 3\}

- Remove the first \(n\) elements from a list:

  > \texttt{drop \(3\) \{1, 2, 3, 4, 5\}}
  \{4, 5\}
• Calculate the length of a list:

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

• Calculate the sum of a list of numbers:

\[ \text{sum } [1, 2, 3, 4, 5] \]
15

• Calculate the product of a list of numbers:

\[ \text{product } [1, 2, 3, 4, 5] \]
120

• Append two lists:

\[ [1, 2, 3] ++ [4, 5] \]
[1, 2, 3, 4, 5]

• Reverse a list:

\[ \text{reverse } [1, 2, 3, 4, 5] \]
[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:

\[ 1 \div 0 \]
Error

\[ \text{head } [] \]
Error

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

For reference, Appendix A shows how special symbols such as ↑ and → are typed using a normal keyboard, and Appendix B presents some of the most commonly used definitions from the standard prelude.
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

\[ f(a, b) + c \times d \]

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 \( \ast \). For example, the expression above would be written in Haskell as follows:

\[ f a \ b + c \ast d \]

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:

<table>
<thead>
<tr>
<th>Mathematics</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) )</td>
<td>( f )</td>
</tr>
<tr>
<td>( f(x, y) )</td>
<td>( f \ x \ y )</td>
</tr>
<tr>
<td>( f(g(x)) )</td>
<td>( f \ (g \ x) )</td>
</tr>
<tr>
<td>( f(x, g(y)) )</td>
<td>( f \ x \ (g \ y) )</td>
</tr>
<tr>
<td>( f(x) g(y) )</td>
<td>( f \ x \ast g \ y )</td>
</tr>
</tbody>
</table>

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 in the standard prelude, it is also possible to define new functions. New functions cannot be defined at the \( \geq \) prompt within Hugs, but must be defined within a Haskell 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.

2.4.1 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:

\[
\text{double } x = x + x \\
\text{quadruple } x = \text{double (double } x) \\
\]

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:

\[> \text{:load test.hs}\]

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

\[> \text{quadruple 10}\]
\[40\]

\[> \text{take (double 2) [1, 2, 3, 4, 5, 6]}\]
\[[1, 2, 3, 4]\]

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:

\[
\text{factorial } n = \text{product } [1..n] \\
\text{average } ns = \text{sum } ns \text{ 'div' length } ns \\
\]

We could equally well have defined \(\text{average } ns = \text{div (sum } ns) \text{ (length } ns)\), but writing \(\text{div}\) between its two arguments is more natural. In general, any such 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:

\[> \text{:reload}\]

\[> \text{factorial 10}\]
\[3628800\]

\[> \text{average [1, 2, 3, 4, 5]}\]
\[3\]

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, \(\text{:load}\) can be abbreviated by \(:l\). The command \(\text{:type}\) is explained in more detail in the next chapter.
2.4.2 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:

\[
\text{myFun} \quad \text{fun1} \quad \text{arg_2} \quad x'
\]

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

\[
\text{case} \quad \text{class} \quad \text{data} \quad \text{default} \quad \text{deriving} \quad \text{do} \quad \text{else} \\
\text{if} \quad \text{import} \quad \text{infix} \quad \text{infixl} \quad \text{infixr} \quad \text{instance} \\
\text{let} \quad \text{module} \quad \text{newtype} \quad \text{of} \quad \text{then} \quad \text{type} \quad \text{where}
\]

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`.

2.4.3 The layout rule

When defining new definitions in 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

\[
a \quad = \quad b + c \\
\text{where} \\
\quad b \quad = \quad 1 \\
\quad c \quad = \quad 2 \\
\quad d \quad = \quad a \times 2
\]

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

\[
\begin{align*}
  a &= b + c \\
  \text{where} & \quad \{ b = 1; \\
  & \quad c = 2 \} \\
  d &= a \ast 2
\end{align*}
\]

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

### 2.4.4 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 \times \text{div} \text{ 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:

```
{-
  \text{double} x \quad = \quad x + x
  \text{quadruple} x \quad = \quad \text{double} (\text{double} x)
-}
```

### 2.5 Chapter remarks

The Hugs system is freely available on the web from the Haskell home page, [www.haskell.org](http://www.haskell.org), which also contains a wealth of other useful resources.

### 2.6 Exercises

1. Parenthesise the following arithmetic expressions:

   \[
   \begin{align*}
   2 \uparrow 3 \ast 4 \\
   2 \ast 3 + 4 \ast 5 \\
   2 + 3 \ast 4 \uparrow 5
   \end{align*}
   \]
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.

\[
N \equiv a \ 'div' \ length \ xs \\
\text{where} \\
a = 10 \\
xs = [1, 2, 3, 4, 5]
\]

4. Show how the library function \texttt{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 \texttt{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 \( \text{Bool} \) contains the two logical values \( \text{False} \) and \( \text{True} \), while the type \( \text{Bool} \rightarrow \text{Bool} \) contains all functions that map arguments from \( \text{Bool} \) to results from \( \text{Bool} \), 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:

\[
\begin{align*}
\text{False} &:: \text{Bool} \\
\text{True} &:: \text{Bool} \\
\neg &:: \text{Bool} \rightarrow \text{Bool}
\end{align*}
\]

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:

\[
\begin{align*}
\neg \text{False} &:: \text{Bool} \\
\neg \text{True} &:: \text{Bool} \\
\neg (\neg \text{False}) &:: \text{Bool}
\end{align*}
\]

In Haskell, every expression must have a type, which is calculated prior to evaluating the expression by a process called type inference. In particular, there are a set of typing rules that are used to calculate the type of expressions from the types of their components. The key such rule concerns function application, and 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 of \( f \) to \( e \) has type \( B \). That is, we have the following rule:
if \( f :: A \rightarrow B \) and \( e :: A \), then \( f \ e :: B \)

For example, the typing \( \neg \ False :: \text{Bool} \) can be inferred from this rule using the
fact that \( \neg :: \text{Bool} \rightarrow \text{Bool} \) and \( \text{False} :: \text{Bool} \). 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 \( 3 :: \text{Bool} \), 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
\textit{type error}, and are deemed to be invalid expressions.

Because type inference precedes evaluation, Haskell programs are \textit{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 ‘\texttt{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
\textbf{if} \( \text{True} \) \textbf{then} 1 \textbf{else} \text{False} \textit{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, \textit{False}, is a logical
value. In practice, however, programmers quickly learn how to work within
the limits of the typing rules and avoid such problems.

In the Hugs system, the type of any expression can be displayed by pre-
ceeding the expression by the command \textit{:type}. For example:

\begin{verbatim}
> :type \neg
\neg :: \text{Bool} \rightarrow \text{Bool}

> :type \neg \text{False}
\neg \text{False} :: \text{Bool}

> :type \neg 3
\text{Error}
\end{verbatim}

\section*{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.

\textit{Bool} - \textit{logical values}

This type contains the two logical values \textit{False} and \textit{True}.

\textit{Char} - \textit{single characters}

This type contains all single characters that are available from a normal key-
board, such as ‘a’, ‘A’, ‘3’ and ‘_’, as well as a number of \textit{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 "". 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 be a value of type 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 operations for handling fixed-precision integers with great speed, whereas arbitrary-precision integers must be processed using the slower medium of software.

**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 $\sqrt{2}$ 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 interesting 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:

\[
\begin{align*}
[False, True, False] &:: [Bool] \\
[\text{\textquoteleft}a\text{\textquoteleft}, \text{\textquoteleft}b\text{\textquoteleft}, \text{\textquoteleft}c\text{\textquoteleft}, \text{\textquoteleft}d\text{\textquoteleft}] &:: [Char] \\
[\text{\textquoteleft}One\text{\textquoteleft}, \text{\textquoteleft}Two\text{\textquoteleft}, \text{\textquoteleft}Three\text{\textquoteleft}] &:: [String]
\end{align*}
\]

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 \([False]\) and \([\text{\textquoteleft}a\text{\textquoteleft}\)], 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 \([False, True]\) and \([False, True, False]\) both have type \([Bool]\), even though they have different lengths. Secondly, there are no restrictions on the type of the elements of a list. At present we are limited in the range of examples that we can give because the only non-basic type that we have introduced at this point is list types, but we can have lists of lists, such as:

\[
[[\text{\textquoteleft}a\text{\textquoteleft}, \text{\textquoteleft}b\text{\textquoteleft}],[\text{\textquoteleft}c\text{\textquoteleft}, \text{\textquoteleft}d\text{\textquoteleft}, \text{\textquoteleft}e\text{\textquoteleft}] &:: [[Char]]
\]

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 \(i\)th components have type \(T_i\) for any \(i\) in the range 1 to \(n\). For example:

\[
\begin{align*}
(False, True) &:: (Bool, Bool) \\
(False, 'a', True) &:: (Bool, Char, Bool) \\
("Yes", True, 'a') &:: (String, Bool, Char)
\end{align*}
\]

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 \((\text{Bool}, \text{Char})\) contains all pairs comprising a first component of type \text{Bool} and a second component of type \text{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:

\[
(\text{'a'}, (\text{False}, \text{'b'})) \quad :: \quad (\text{Char}, (\text{Bool}, \text{Char}))
\]

\[
([\text{'a'}, \text{'b'}], [\text{False}, \text{True}]) \quad :: \quad ([\text{Char}], [\text{Bool}])
\]

\[
([\text{'a'}, \text{False}], ([\text{'b'}], \text{True})) \quad :: \quad ([\text{Char}, \text{Bool})]
\]

Finally, 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:

\[
\neg \quad :: \quad \text{Bool} \rightarrow \text{Bool}
\]

\[
isDigit \quad :: \quad \text{Char} \rightarrow \text{Bool}
\]

(The library function \text{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 \text{add} that calculates the sum of a pair of integers, and a function \text{zeroto} that returns the list of integers from zero to a given limit, as follows:

\[
\text{add} \quad :: \quad (\text{Int}, \text{Int}) \rightarrow \text{Int}
\]

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

\[
isDigit \quad :: \quad \text{Int} \rightarrow [\text{Int}]
\]

\[
\text{zeroto} \ n \ = \ [0\ldots n]
\]

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

Note that there is no restriction that functions must be total on their argument type, in the sense that there may be some arguments for which the result of a function is not defined. For example, the result of library function \text{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:

\[
\begin{align*}
\text{add}' & : \text{Int} \to (\text{Int} \to \text{Int}) \\
\text{add}' \ x \ y & = x + y
\end{align*}
\]

The type states that \text{add}' is a function that takes an argument of type \text{Int}, and returns a result that is a function of type \text{Int} \to \text{Int}. The definition itself states that \text{add}' takes an integer \(x\) followed by an integer \(y\) and returns the result \(x + y\). More precisely, \text{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 \text{add}' produces the same final result as the function \text{add} from the previous section, but whereas \text{add} takes its two arguments at the same time packaged as a pair, \text{add}' takes its two arguments \textit{one at a time}, as reflected in the different types of the two functions:

\[
\begin{align*}
\text{add} & : (\text{Int,Int}) \to \text{Int} \\
\text{add}' & : \text{Int} \to (\text{Int} \to \text{Int})
\end{align*}
\]

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 \text{mult} that takes three integers, one at a time, and returns their product, can be defined as follows:

\[
\begin{align*}
\text{mult} & : \text{Int} \to (\text{Int} \to (\text{Int} \to \text{Int})) \\
\text{mult} \ x \ y \ z & = x * y * z
\end{align*}
\]

This definition states that \text{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 \text{add}' and \text{mult} that take their arguments one at a time are called \textit{curried functions}. As well as being interesting in their own right, curried functions are also more flexible than functions on tuples, because useful functions can often be made by \textit{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 \text{add}' \(1 : \text{Int} \to \text{Int}\) of the curried function \text{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 \(\to\) in types is assumed to associate to the right. For example,

\[
\text{Int} \to \text{Int} \to \text{Int} \to \text{Int}
\]

means

\[
\text{Int} \to (\text{Int} \to (\text{Int} \to \text{Int}))
\]

34
Consequently, function application, which is denoted silently using spacing, is assumed to associate to the left. For example,

\[ \text{mult } x \ y \ z \]

means

\[ ((\text{mult } x) \ y) \ z \]

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 \( \text{length} \) 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:

\[
> \text{length} \ [1, 3, 5, 7] \\
4
\]

\[
> \text{length} \ ["Yes", "No"] \\
2
\]

\[
> \text{length} \ [\text{isDigit}, \text{isLower}, \text{isUpper}] \\
3
\]

The idea that the function \( \text{length} \) can be applied to lists whose elements have any type is made precise in its type by the inclusion of a \textit{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 \( \text{length} \) is as follows:

\[
\text{length} :: [a] \to \text{Int}
\]

That is, for any type \( a \), the function \( \text{length} \) has type \([a] \to \text{Int}\). A type that contains one or more type variables is called \textit{polymorphic} (“of many forms”), as is an expression with such a type. Hence, \([a] \to \text{Int}\) is a polymorphic type and \( \text{length} \) is a polymorphic function. More generally, many of the functions provided in the standard prelude are polymorphic. For example:

\[
\begin{align*}
\text{fst} & :: (a, b) \to a \\
\text{head} & :: [a] \to a \\
\text{take} & :: \text{Int} \to [a] \to [a] \\
\text{zip} & :: [a] \to [b] \to [(a, b)] \\
\text{id} & :: a \to a
\end{align*}
\]
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, in which case the result is another integer, or the sum of two floating-point numbers, in which case the result is another floating-point number:

\[
\begin{align*}
> & \ 1 + 2 \\
& \ 3 \\
> & \ 1.1 + 2.2 \\
& \ 3.3
\end{align*}
\]

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:

\[
(+) \ :: \ Num\ a \Rightarrow a \to a \to a
\]

That is, for any type \( a \) that is a instance of the class \( Num \) of numeric types, the function \((+)\) has type \( a \to a \to a \). (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, \( Num\ a \Rightarrow a \to a \to a \) 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:

\[
\begin{align*}
(--) & :: \ Num\ a \Rightarrow a \to a \to a \\
(*) & :: \ Num\ a \Rightarrow a \to a \to a \\
\text{negate} & :: \ Num\ a \Rightarrow a \to a \\
\text{abs} & :: \ Num\ a \Rightarrow a \to a \\
\text{signum} & :: \ Num\ a \Rightarrow a \to a
\end{align*}
\]

Moreover, numbers themselves are also overloaded. For example, \( 3::\ Num\ a \Rightarrow a \) 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 differen
t using the following two methods:

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

All the basic types boolean, 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:

\[ (<) \:: \ a \to \ a \to \text{Bool} \]  
\[ (\leq) \:: \ a \to \ a \to \text{Bool} \]  
\[ (>) \:: \ a \to \ a \to \text{Bool} \]  
\[ (\geq) \:: \ a \to \ a \to \text{Bool} \]  
\[ \text{min} \:: \ a \to \ a \to \text{a} \]  
\[ \text{max} \:: \ a \to \ a \to \text{a} \]

All the basic types boolean, 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
> \texttt{min 'a' 'b' 'a'}

> "elegant" < "elephant"
\texttt{True}

> \texttt{[1,2,3] < [1,2]}
\texttt{False}

> \texttt{( 'a', 2) < ( 'b', 1) }
\texttt{True}

> \texttt{( 'a', 2) < ( 'a', 1) }
\texttt{False}

Note that strings, lists and tuples are ordered \textit{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.

\textit{Show - showable types}

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

\[ \text{show} :: a \rightarrow \text{String} \]

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

> \texttt{show False}
\texttt{"False"}

> \texttt{show 'a'}
\texttt{"'a'"}

> \texttt{show 123}
\texttt{"123"}

> \texttt{show [1,2,3]}
\texttt{"[1,2,3]"}

> \texttt{show ('a',False)}
\texttt{"( '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:

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

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

\begin{verbatim}
> 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)
\end{verbatim}

The use of :: in these examples resolves the type of the result. In practice, however, the necessary type information can often be inferred automatically from the context. For example, the expression \( \neg (\text{read } \texttt{"False"}) \) requires no explicit type information, because the application of the logical negation function \( \neg \) implies that \text{read } \texttt{"False"} must have type \texttt{Bool}.

Note that the result of \text{read} is undefined if its argument is not syntactically valid. For example, the expression \( \neg (\text{read } \texttt{"hello"}) \) produces an error when evaluated, because \texttt{"hello"} cannot be read as a logical value.

Num - numeric types

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

\[
\begin{align*}
\text{(+)} & :: a \rightarrow a \rightarrow a \\
\text{(-)} & :: a \rightarrow a \rightarrow a \\
\text{(\star)} & :: a \rightarrow a \rightarrow a \\
\text{negate} & :: a \rightarrow a \\
\text{abs} & :: a \rightarrow a \\
\text{signum} & :: a \rightarrow a
\end{align*}
\]
(The method \texttt{negate} returns the negation of a number, \texttt{abs} returns the absolute value, while \texttt{signum} returns the sign.) The basic types \textit{Int}, \textit{Integer} and \textit{Float} are instances of the \textit{Num} class. For example:

\begin{verbatim}
> 1 + 2
3

> 1.1 + 2.2
3.3

> negate 3.3
-3.3

> abs (-3)
3

> signum (-3)
-1
\end{verbatim}

Note that the \textit{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.

\textit{Integral} - \textbf{integral types}

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

\begin{verbatim}
div :: a -> a -> a
mod :: a -> a -> a
\end{verbatim}

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

\begin{verbatim}
> 7 'div' 2
3

> 7 'mod' 2
1
\end{verbatim}

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

40
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:

```plaintext
(/) :: a → a → a
recip :: a → a
```

The basic type `Float` is an instance of the `Fractional` class. For example:

```plaintext
> 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 the Haskell Report [18], while formal descriptions for specialists can be found in [14, 3].

3.11 Exercises

1. What are the types of the following values?

```plaintext
['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?

```plaintext
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)
```

Hine: 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 to make function types 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 questions, 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:

  
  \[
  \text{isDigit} :: \text{Char} \rightarrow \text{Bool} \\
  \text{isDigit } c = \text{'}0' \leq c \leq \text{'}9'
  \]

- Decide if an integer is even:

  
  \[
  \text{even} :: \text{Integral } a \Rightarrow a \rightarrow \text{Bool} \\
  \text{even } n = n \mod 2 =\equiv 0
  \]

- Split a list at the \text{n}th element:

  \[
  \text{splitAt} :: \text{Int} \rightarrow [a] \rightarrow ([a], [a]) \\
  \text{splitAt } n \text{ } xs = (\text{take } n \text{ } xs, \text{drop } n \text{ } xs)
  \]

- Reciprocation:

  \[
  \text{recip} :: \text{Fractional } a \Rightarrow a \rightarrow a \\
  \text{recip } n = 1 / n
  \]

Note the use of the class constraints in the types for \text{even} and \text{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:

\[
\text{abs} \quad :: \quad \text{Int} \to \text{Int} \\
\text{abs} \; n \quad = \quad \text{if} \; n \geq 0 \; \text{then} \; n \; \text{else} \; -n
\]

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

\[
\text{signum} \quad :: \quad \text{Int} \to \text{Int} \\
\text{signum} \; n \quad = \quad \text{if} \; n < 0 \; \text{then} \; -1 \; \text{else} \\
\phantom{\text{signum} \; n \; = \quad \text{if} \; n < 0 \; \text{then} \; -1 \; \text{else}} \text{if} \; n == 0 \; \text{then} \; 0 \; \text{else} \; 1
\]

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

4.3 Guarded equations

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

\[
\text{abs} \; n \; | \; n \geq 0 \; = \; n \\
\phantom{\text{abs} \; n \; | \; n \geq 0 \; = \; n} \text{otherwise} \; = \; -n
\]

The symbol `|` is read as “such that”, and the guard `otherwise` is defined in the library file 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 \textit{signum} is easier to understand when defined as follows:

\[
\text{signum} \ n \ |
\begin{align*}
\text{if } n < 0 & \quad \Rightarrow -1 \\
\text{if } n == 0 & \quad \Rightarrow 0 \\
\text{otherwise} & \quad \Rightarrow 1
\end{align*}
\]

4.4 Pattern matching

Many functions have a particularly simple and intuitive definition using \textit{pattern matching}, in which a sequence of syntactic expressions called \textit{patterns} is used to choose between a sequence of results of the same type. If the first pattern is \textit{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 \texttt{\texttt{\neg}} that returns the negation of a logical value is defined as follows:

\[
\neg \quad :: \quad \texttt{Bool} \rightarrow \texttt{Bool} \\
\neg \texttt{False} \quad = \quad \texttt{True} \\
\neg \texttt{True} \quad = \quad \texttt{False}
\]

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

\[
\texttt{(\&\&)} \quad :: \quad \texttt{Bool} \rightarrow \texttt{Bool} \rightarrow \texttt{Bool} \\
\texttt{True \&\& True} \quad = \quad \texttt{True} \\
\texttt{True \&\& False} \quad = \quad \texttt{False} \\
\texttt{False \&\& True} \quad = \quad \texttt{False} \\
\texttt{False \&\& False} \quad = \quad \texttt{False}
\]

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

\[
\texttt{True \&\& True} \quad = \quad \texttt{True} \\
\texttt{\_\_ \&\& \_\_} \quad = \quad \texttt{False}
\]

This version also has the benefit that, under lazy evaluation as discussed in chapter 12, if the first argument is \texttt{False} then the result \texttt{False} is returned without the need to evaluate the second argument. In practice, the library file defines \texttt{\&\&} using equations that have this same property, but make the choice about which equation applies using the value of the first argument only:

\[
\texttt{True \&\& \_} \quad = \quad \texttt{b} \\
\texttt{False \&\& \_} \quad = \quad \texttt{False}
\]

That is, if the first argument is \texttt{True} then the result is the value of the second argument, otherwise if the first argument is \texttt{False} then the result is \texttt{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 $\land$ 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 convention:

\[
b \land b = b \\
\bot \land \bot = False
\]

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

\[
b \land c \mid b == c = b \\
\mid otherwise = False
\]

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

### 4.4.1 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 \texttt{fst} and \texttt{snd} that select the first and second components of a pair are defined as follows:

\[
\begin{align*}
\texttt{fst} & :: (a, b) \rightarrow a \\
\texttt{fst} (x, \_ ) & = x \\
\texttt{snd} & :: (a, b) \rightarrow b \\
\texttt{snd} (\_, y) & = y
\end{align*}
\]

### 4.4.2 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 \texttt{test} that decides if a list contains precisely three characters beginning with the letter \texttt{a} can be defined as follows:

\[
\begin{align*}
\texttt{test} & :: [Char] \rightarrow Bool \\
\texttt{test} ['a', \_ , \_] & = True \\
\texttt{test} \_ & = False
\end{align*}
\]

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 \texttt{cons} (abbreviating “construct”) that produces a new list by prepending a new element to the start of an existing list. For example, the following calculation shows how the list \([1, 2, 3]\) can be understood in this way:
\[ [1, 2, 3] \]
\[ = \{ \text{applying cons} \} \]
\[ 1 : [2, 3] \]
\[ = \{ \text{applying cons} \} \]
\[ 1 : (2 : [3]) \]
\[ = \{ \text{applying cons} \} \]
\[ 1 : (2 : (3 : [])) \]

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

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

\[
\begin{align*}
\text{test} & \quad :: \ [\text{Char}] \to \text{Bool} \\
\text{test} \ ('a' : _) & = \ True \\
\text{test} \ _ & = \ False
\end{align*}
\]

Similarly, the library functions \text{null}, \text{head} and \text{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:

\[
\begin{align*}
\text{null} & \quad :: \ [a] \to \text{Bool} \\
\text{null} \ [] & = \ True \\
\text{null} \ (_ : _) & = \ False \\
\text{head} & \quad :: \ [a] \to a \\
\text{head} \ (x : _) & = \ x \\
\text{tail} & \quad :: \ [a] \to [a] \\
\text{tail} \ (_ : xs) & = \ xs
\end{align*}
\]

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

### 4.4.3 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 \text{pred} that maps zero to itself and any strictly positive integer to its predecessor can be defined as follows:

\[
\begin{align*}
\text{pred} & \quad :: \ \text{Int} \to \text{Int} \\
\text{pred} \ 0 & = 0 \\
\text{pred} \ (n + 1) & = n
\end{align*}
\]
There are two points to note about \( n + k \) patterns. First of all, they only match integers \( \geq k \). For example, evaluating \textsf{pred} \((-1)\) produces an error, because neither of the two patterns in the definition for \textsf{pred} matches negative integers. Secondly, for the same reason as \textsf{cons} patterns, integer patterns must be parenthesised. For example, the definition \textsf{pred} \( n + 1 = n \) without parentheses means \((\textsf{pred} n) + 1 = n\), which is an invalid definition.

4.5 Lambda expressions

As an alternative to defining functions using equations, functions can also be constructed using \textit{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:

\[
\lambda x \to x + x
\]

The symbol \( \lambda \) is the lower-case Greek letter “lambda”. Despite the fact they have no names, functions constructed using lambda expressions can be used in the same way as any other functions. For example:

\[
> (\lambda x \to x + x) \ 2 \\
4
\]

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

\[
\text{add} \ x \ y = x + y
\]

can be understood as meaning

\[
\text{add} = \lambda x \to (\lambda y \to x + y)
\]

which makes precise that \textit{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 \textit{const} that returns a constant function that always produces a given value can be defined as follows:

\[
\begin{align*}
\text{const} &:: a \to b \to a \\
\text{const} \ x &= x
\end{align*}
\]
However, it is more appealing to define \textit{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:

\[
\begin{align*}
\text{const} & \quad :: \quad a \to (b \to a) \\
\text{const} \ x & \quad = \quad \lambda \_ \to x
\end{align*}
\]

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

\[
\begin{align*}
\text{odds} & \quad :: \quad \text{Int} \to [\text{Int}] \\
\text{odds} \ n & \quad = \quad \text{map} \ f \ [0\ldots n - 1] \\
\text{where} & \\
\quad f \ x & \quad = \quad x \ast 2 + 1
\end{align*}
\]

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

\[
\text{odds} \ n \quad = \quad \text{map} \ (\lambda x \to x \ast 2 + 1) \ [0\ldots n - 1]
\]

\section{Sections}

Functions such as + that are written between their two arguments are called \textit{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 \textquoteleft div\textquoteleft 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 \textit{sections}, whose meaning as functions can be formalised using lambda expressions as follows:

\[
\begin{align*}
(\oplus) & \quad = \lambda x \to (\lambda y \to x \oplus y) \\
(x \oplus) & \quad = \lambda y \to x \oplus y \\
(\oplus \ y) & \quad = \lambda x \to x \oplus y
\end{align*}
\]

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:

\[
\begin{align*}
+ & \quad \text{is the addition function } \lambda x \to (\lambda y \to x + y) \\
1+ & \quad \text{is the successor function } \lambda y \to 1 + y \\
1/ & \quad \text{is the reciprocation function } \lambda y \to 1 / y \\
*2 & \quad \text{is the doubling function } \lambda x \to x \ast 2 \\
/2 & \quad \text{is the halving function } \lambda x \to x / 2
\end{align*}
\]
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 \( \land \) is stated as follows:

\[
(\land) :: \text{Bool} \to \text{Bool} \to \text{Bool}
\]

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

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

\[
\text{and} = \text{foldr} (\land) \text{True}
\]

Note that the definition \( \text{and} = \text{foldr} \land \text{True} \) without parentheses would mean that \texttt{and} was defined by applying the operator \( \land \) to the arguments \texttt{foldr} and \texttt{True}, which is both the incorrect meaning and an invalid definition.

### 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 [18]. The Greek letter \( \lambda \) 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 \( \text{halve} :: [a] \to ([a],[a]) \) that splits an even-lengthed list into two halves. For example:

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

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

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

   Hint: make use of the library function \texttt{null}.

3. In a similar way to \( \land \), show how the logical disjunction operator \( \lor \) 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:

\[
\begin{align*}
\text{True} \land \text{True} &= \text{True} \\
\text{False} \land \text{False} &= \text{False}
\end{align*}
\]

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

\[
\begin{align*}
\text{True} \land b &= b \\
\text{False} \land \text{False} &= \text{False}
\end{align*}
\]

6. Show how the curried function definition \( \text{mult} \ x \ y \ z = x \ast y \ast z \) 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 \textit{zip} and the idea of string comprehensions, and conclude by developing a program to crack the Caesar cipher.

5.1 Generators

In mathematics, the \textit{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:

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

The symbols | and \(\leftarrow\) are read as “such that” and “is drawn from” respectively, and the expression \(x \leftarrow \{1..5\}\) is called a \textit{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 4,4,5,5,5,5), in the previous case the \(y\) components change more frequently than the \(x\) components.
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 \([1 \ldots 3]\) can be produced as follows:

\[
\begin{align*}
> & [(x, y) \mid x \leftarrow [1 \ldots 3], y \leftarrow [x \ldots 3]] \\
& [(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
\end{align*}
\]

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:

\[
\begin{align*}
\text{concat} & :: [[a]] \rightarrow [a] \\
\text{concat} \ xss & = [x \mid xss \leftarrow xss, x \leftarrow x]
\end{align*}
\]

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:

\[
\begin{align*}
\text{firsts} & :: [(a, b)] \rightarrow [a] \\
\text{firsts} \ ps & = [x \mid (x, \_) \leftarrow ps]
\end{align*}
\]

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:

\[
\begin{align*}
\text{length} & :: [a] \rightarrow \text{Int} \\
\text{length} \ xs & = \text{sum} [1 \mid \_ \leftarrow xs]
\end{align*}
\]

In this case, the generator \(\_ \leftarrow xs\) 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 \([x \mid x \leftarrow [1 \ldots 10], \text{even} \ x]\) produces the list \([2, 4, 6, 8, 10]\) of all even numbers from the list \([1 \ldots 10]\). Similarly, a function that maps a positive integer to its list of positive `factors` can be defined as follows:

\[
\begin{align*}
\text{factors} & :: \text{Int} \rightarrow [\text{Int}] \\
\text{factors} \ n & = [x \mid x \leftarrow [1 \ldots n], n \mod x == 0]
\end{align*}
\]

For example:
> factors 15
[1, 3, 5, 15]

> factors 7
[1, 7]

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

\[
\begin{align*}
\text{prime} & : \text{Int} \rightarrow \text{Bool} \\
\text{prime } n & = \text{factors } n \implies [1, n]
\end{align*}
\]

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:

\[
\begin{align*}
\text{primes} & : \text{Int} \rightarrow [\text{Int}] \\
\text{primes } n & = [x \mid x \leftarrow [2..n], \text{prime } x]
\end{align*}
\]

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:

\[
\begin{align*}
\text{find} & : \text{Eq } a \Rightarrow a \rightarrow [(a, b)] \rightarrow [b] \\
\text{find } k \ t & = [v \mid (k', v) \leftarrow t, k \equiv k']
\end{align*}
\]

For example:

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

55
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:

\[
\begin{align*}
> \text{zip [\text{'a'}, \text{'b'}, \text{'c'}]} & \ [1, 2, 3, 4] \\
& \ [(\text{'a'}, 1), (\text{'b'}, 2), (\text{'c'}, 3)]
\end{align*}
\]

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:

\[
\begin{align*}
pairs & \quad :: \ [a] \rightarrow [(a, a)] \\
pairs \ \text{xs} & \quad = \ \text{zip xs (tail xs)}
\end{align*}
\]

For example:

\[
\begin{align*}
> \text{pairs [1, 2, 3, 4]} \\
& \ [(1, 2), (2, 3), (3, 4)]
\end{align*}
\]

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:

\[
\begin{align*}
sorted & \quad :: \ \text{Ord a} \Rightarrow [a] \rightarrow \text{Bool} \\
sorted \ \text{xs} & \quad = \ \text{and} \ [x \leq y \mid (x, y) \gets \text{pairs xs}]
\end{align*}
\]

For example:

\[
\begin{align*}
> \text{sorted [1, 2, 3, 4]} \\
& \ True
\end{align*}
\]

\[
\begin{align*}
> \text{sorted [1, 3, 2, 4]} \\
& \ False
\end{align*}
\]

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 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:

\[
\begin{align*}
\text{positions} & \quad :: \ \text{Eq a} \Rightarrow a \rightarrow [a] \rightarrow [\text{Int}] \\
\text{positions} \ \text{xs} & \quad = \ [i \mid (x', i) \gets \text{zip xs [0..n]}, x == x'] \\
& \quad \text{where} \ n = \text{length xs} - 1
\end{align*}
\]

For example:

\[
\begin{align*}
> \text{positions False [True, False, True, False]} \\
& \ [1, 3]
\end{align*}
\]

56
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]
[('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 "haskell498"
7

> 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 message

"haskell is fun"
would be encoded as

“kdvnho 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.

5.5.1 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:

\[
\begin{align*}
\text{let2int} & :: \text{Char} \rightarrow \text{Int} \\
\text{let2int} \ x &= \text{ord} \ x - \text{ord} \ ('a') \\
\text{int2let} & :: \text{Int} \rightarrow \text{Char} \\
\text{int2let} \ n &= \text{chr} \ (\text{ord} \ ('a') + n)
\end{align*}
\]

(The library functions \(\text{ord} :: \text{Char} \rightarrow \text{Int}\) and \(\text{chr} :: \text{Int} \rightarrow \text{Char}\) convert between characters and their Unicode representation as integers.) For example:

\[
\begin{align*}
> \ \text{let2int} \ 'a' \\
& 0 \\
> \ \text{int2let} \ 0 \\
& 'a'
\end{align*}
\]

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:

\[
\begin{align*}
\text{shift} & :: \text{Int} \rightarrow \text{Char} \rightarrow \text{Char} \\
\text{shift} \ n \ x \mid \text{isLower} \ x &= \text{int2let} \ ((\text{let2int} \ x + n) \mod 26) \\
\mid \text{otherwise} &= x
\end{align*}
\]

Note that this function accepts both positive and negative shift factors, and that only lower-case letters are changed. For example:
Using `shift` within a string comprehension, it is now easy to define a function that encodes a string using a given shift factor:

\[
\text{encode} \quad :: \quad \text{Int} \to \text{String} \to \text{String} \\
\text{encode } n \ x s = [\text{shift } n \ x | x \leftarrow x s]
\]

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

\[
> \text{encode } 3 \ "haskell} \_\text{i} \_\text{s} \_\text{fun}" \hspace{1cm} "\text{kdvnhoo} \_\text{l} \_\text{v} \_\text{ixq}"
\]

\[
> \text{encode } (-3) \ "\text{kdvnhoo} \_\text{l} \_\text{v} \_\text{ixq}" \hspace{1cm} "\text{haskell} \_\text{i} \_\text{s} \_\text{fun}"
\]

### 5.5.2 Frequency tables

In English text, some letters 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:

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

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 do this, we first define a function that calculates the percentage of one integer with respect to another, returning the result as a floating-point number:

\[
\begin{align*}
\text{percent} \quad :: \quad & \text{Int} \to \text{Int} \to \text{Float} \\
\text{percent } n \ m &= (\text{fromInt } n / \text{fromInt } m) * 100
\end{align*}
\]

(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:

\[
\begin{align*}
\text{freqs} & \quad :: \quad \text{String} \rightarrow \left[ \text{Float} \right] \\
\text{freqs} \; xs & \quad = \quad \left[ \text{percent} \left( \text{count} \; x \; xs \right) \; n \mid x \leftarrow ['a' \ldots 'z'] \right] \\
& \quad \quad \text{where} \quad n = \text{lowers} \; xs
\end{align*}
\]

For example:

\[
\begin{align*}
> \text{freqs} \; "\text{abbc}\cdots\text{ddee}" \\
[6.7, 13.3, 20.0, 26.7, 33.3, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, \ldots, 0.0]
\end{align*}
\]

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 \(n = \text{lowers} \; xs\) within \text{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.

### 5.5.3 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 \(i\)th 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 \text{zip} and a list comprehension, it is easy to translate the above formula into a function definition:

\[
\begin{align*}
\text{chisqr} & \quad :: \quad \left[ \text{Float} \right] \rightarrow \left[ \text{Float} \right] \rightarrow \text{Float} \\
\text{chisqr} \; os \; es & \quad = \quad \text{sum} \left[ ((o - e) \uparrow 2) / e \mid (o, e) \leftarrow \text{zip} \; os \; es \right]
\end{align*}
\]

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:

\[
\begin{align*}
\text{rotate} & \quad :: \quad \text{Int} \rightarrow \left[ \text{a} \right] \rightarrow \left[ \text{a} \right] \\
\text{rotate} \; n \; xs & \quad = \quad \text{drop} \; n \; xs \; \uplus \; \text{take} \; n \; xs
\end{align*}
\]

For example:

\[
\begin{align*}
> \text{rotate} \; 3 \; [1, 2, 3, 4, 5] \\
[4, 5, 1, 2, 3]
\end{align*}
\]
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 \(table' = \text{freqs} \ "kdvhoo_\_lv_\_ixq"\) then

\[
[\text{chisqr (rotate n table')} \mid n \leftarrow [0..25]]
\]
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 \textit{positions} from earlier in this chapter, this procedure can be implemented as follows:

\[
\begin{align*}
\text{\textit{crack}} & \quad \text{:: String \rightarrow String} \\
\text{\textit{crack xs}} &= \text{encode \ (-factor)} \text{ xs} \\
\text{\textbf{where}} \\
\text{\quad factor} &= \text{head (positions (minimum chitab) chitab)} \\
\text{\quad chitab} &= [\text{chisqr (rotate n table')} \mid n \leftarrow [0..25]] \\
\text{\quad table'} &= \text{freqs xs}
\end{align*}
\]

For example:

\[
\begin{align*}
> \text{\textit{crack}} \ "kdvhoo_\_lv_\_ixq" \\
&"\text{haskell_\_is_\_fun}"
\end{align*}
\]

\[
\begin{align*}
> \text{\textit{crack}} \ "vscd_\_mywzboroxcsyc_\_kbo_\_ecopev" \\
&"\text{list_\_comprehensions_\_are_\_useful}"
\end{align*}
\]

More generally, the \textit{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:

\[
\begin{align*}
> \text{\textit{crack}} (\text{\textit{encode}} \ 3 \ "\text{haskell}\"") \\
&"\text{piasmmt}"
\end{align*}
\]

\[
\begin{align*}
> \text{\textit{crack}} (\text{\textit{encode}} \ 3 \ "\text{the_\_five_\_boxing_\_wizards_\_jump_\_quickly}\"") \\
&"\text{dro_\_psfo_\_lyhsxq_\_gsjkbncl_\_tewz_\_aesmuvi}"
\end{align*}
\]

### 5.6 Chapter remarks

The term \textit{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 [18]. A popular account of the Caesar cipher, any many other famous cryptographic methods, is given in The Code Book [22].

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 $\text{length}$, show how the library function $\text{replicate} :: \text{Int} \rightarrow a \rightarrow [a]$ 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 is pythagorean if $x^2 + y^2 = z^2$. Using a list comprehension, define a function $\text{pythagoreans} :: \text{Int} \rightarrow [(\text{Int}, \text{Int}, \text{Int})]$ that returns the list of all pythagorean triples whose components are at most a given limit. For example:

   $>$ pythagoreans 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 $\text{factors}$, define a function $\text{perfects} :: \text{Int} \rightarrow [\text{Int}]$ 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 $\text{concat}$ and nest one comprehension within the other.

6. Redefine the function $\text{positions}$ using the function $\text{find}$.

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:

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

   In a similar manner to the function $\text{chisqr}$, show how a list comprehension can be used to define a function $\text{scalarproduct} :: [\text{Int}] \rightarrow [\text{Int}] \rightarrow \text{Int}$ that returns the scalar product of two lists. For example:
8. Modify the Caesar cipher program to also handle upper-case letters.
Chapter 6

Recursive Functions

In this chapter we introduce recursion, the basic mechanism of repetition in Haskell. We start with recursion on integers, then extend the idea to recursion on lists, consider multiple arguments, multiple recursion, and mutual recursion, and conclude with some advice on defining recursive functions.

6.1 Basic concepts

As we have seen in previous chapters, many functions can naturally be defined in terms of other functions. For example, a function that returns the factorial of a non-negative integer can be defined by using library functions to calculate the product of the integers between one and the number itself:

\[
\text{factorial} \quad :: \quad \text{Int} \rightarrow \text{Int} \\
\text{factorial} \ n \quad = \quad \text{product} \ [1..n]
\]

In Haskell, it is also permissible to define functions in terms of themselves, in which case the functions are called recursive. For example, the factorial function can be defined in this manner as follows:

\[
\text{factorial} \ 0 \quad = \quad 1 \\
\text{factorial} \ (n + 1) \quad = \quad (n + 1) \times \text{factorial} \ n
\]

The first equation states that the factorial of zero is one, and is called a base case. The second equation states that the factorial of any strictly positive integer is the product of that number and the factorial of its predecessor, and is called a recursive case. For example, the following calculation shows how the factorial of three is computed using this definition:

\[
\begin{align*}
\text{factorial} \ 3 \\
= & \quad \{ \text{applying } \text{factorial} \} \\
= & \quad 3 \times \text{factorial} \ 2 \\
= & \quad \{ \text{applying } \text{factorial} \} \\
= & \quad 3 \times (2 \times \text{factorial} \ 1) \\
= & \quad \{ \text{applying } \text{factorial} \}
\end{align*}
\]
\[ 3 \ast (2 \ast (1 \ast \text{factorial} \ 0)) = \{ \text{applying factorial} \} \]
\[ 3 \ast (2 \ast (1 \ast 1)) = \{ \text{applying *} \} \]
\[ 6 \]

Note that even though the \textit{factorial} function is defined in terms of itself, it does not loop forever. In particular, each application of \textit{factorial} reduces the integer argument by one, until it eventually reaches zero at which point the recursion stops and the multiplications are performed. Returning one as the factorial of zero is appropriate because one is the identity for multiplication. That is, \(1 \ast x = x\) and \(x \ast 1 = x\) for any integer \(x\).

For the case of the \textit{factorial} function, the original definition using library functions is simpler than the definition using recursion. However, as we shall see in the remainder of this book, many functions have a simple and natural definition using recursion. For example, many of the library functions in Haskell are defined in this way. Moreover, as we shall see in chapter 13, defining functions using recursion also allows properties of those functions to be proved using the powerful mathematical technique of \textit{induction}.

As another example of recursion on integers, consider the multiplication operator \ast used above. For efficiency reasons, this operator is provided as a primitive in Haskell. However, for non-negative integers it can also be defined using recursion on either of its two arguments, such as the second:

\[
(*) \quad : \quad \text{Int} \to \text{Int} \to \text{Int}
\]
\[
m \ast 0 = 0
\]
\[
m \ast (n + 1) = m + (m \ast n)
\]

For example:

\[ 4 \ast 3 = \{ \text{applying *} \} \]
\[ 4 + (4 \ast 2) = \{ \text{applying *} \} \]
\[ 4 + (4 + (4 \ast 1)) = \{ \text{applying *} \} \]
\[ 4 + (4 + (4 + (4 \ast 0))) = \{ \text{applying *} \} \]
\[ 4 + (4 + (4 + 0)) = \{ \text{applying +} \} \]
\[ 12 \]

That is, the recursive definition for the \ast operator formalises the idea that multiplication can be reduced to repeated addition.

### 6.2 Recursion on lists

Recursion is not restricted to functions on integers, but can also be used to define functions on lists. For example, the library function \textit{product} used in the
preceding section can be defined as follows:

\[
\begin{align*}
\text{product} &:: \text{Num } a \Rightarrow [a] \rightarrow a \\
\text{product} [] & = 1 \\
\text{product} (n : ns) & = n \times \text{product} ns
\end{align*}
\]

The first equation states that the product of the empty list is one, which is appropriate because one is the identity for multiplication. The second equation states that the product of any non-empty list is given by multiplying the first number and the product of the remaining list of numbers. For example:

\[
\begin{align*}
\text{product} [2, 3, 4] & = \{\text{applying product}\} \\
& = 2 \times \text{product} [3, 4] \\
& = \{\text{applying product}\} \\
& = 2 \times (3 \times \text{product} [4]) \\
& = \{\text{applying product}\} \\
& = 2 \times (3 \times (4 \times \text{product} [])) \\
& = \{\text{applying product}\} \\
& = 2 \times (3 \times (4 \times 1)) \\
& = \{\text{applying }\times\} \\
& = 24
\end{align*}
\]

Recall that lists in Haskell are actually constructed one element at a time using the \texttt{cons} operator. Hence, \([2, 3, 4]\) is just an abbreviation for \([2 : (3 : (4 : []))].\) As another simple example of recursion on lists, the library function \texttt{length} can be defined using the same pattern of recursion as \texttt{product}:

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

That is, the length of the empty list is zero, and the length of any non-empty list is the successor of the length of its tail. Note the use of the wildcard pattern \(\_\) in the recursive case, which reflects the fact that the length of a list does not depend upon the value of its elements.

Now let us consider the library function that reverses a list. This function can be defined using recursion as follows:

\[
\begin{align*}
\text{reverse} &:: [a] \rightarrow [a] \\
\text{reverse} [] & = [] \\
\text{reverse} (x : xs) & = \text{reverse} xs \mathbin{\&}\{x\}
\end{align*}
\]

That is, the reverse of the empty list is simply the empty list, and the reverse of any non-empty list is given by appending the reverse of its tail to a singleton list comprising the head of the list. For example:

\[
\text{reverse} [1, 2, 3]
\]
= \{\text{applying }\text{reverse}\\}
reverse\ [2,3] \oplus [1]
= \{\text{applying }\text{reverse}\\}
(reverse\ [3] \oplus [2]) \oplus [1]
= \{\text{applying }\text{reverse}\\}
(((reverse\ [] \oplus [3]) \oplus [2]) \oplus [1]
= \{\text{applying }\oplus\\}
[3,2,1]

In turn, the append operator \(\oplus\) used in the above definition of \(\text{reverse}\) can itself be defined using recursion on its first argument:

\[
(\oplus) \quad : \quad [a] \to [a] \to [a]
\]

\[
[] \oplus ys = ys
\]

\[
(x:xs) \oplus ys = x:(xs \oplus ys)
\]

For example:

\[
[1,2,3] \oplus [4,5,6]
= \{\text{applying }\oplus\\}
1 : (\{\text{[2,3] \oplus [4,5,6]}\})
= \{\text{applying }\oplus\\}
1 : (2 : (\{\text{[3] \oplus [4,5,6]}\}))
= \{\text{applying }\oplus\\}
1 : (2 : (3 : (\{\text{[1] \oplus [4,5,6]}\})))
= \{\text{applying }\oplus\\}
1 : (2 : (3 : [4,5,6]))
= \{\text{list notation}\\}
\{\text{[1,2,3,4,5,6]}\}
\]

That is, the recursive definition for \(\oplus\) formalises the idea that two lists can be appended by copying elements from the first list until it is exhausted, at which point the second list is joined on at the end.

We conclude this section with two examples of recursion on sorted lists. First of all, a function that inserts a new element of any ordered type into a sorted list to give another sorted list can be defined as follows:

\[
\begin{align*}
\text{insert} & \quad : \quad \text{Ord}\ a \Rightarrow a \to [a] \to [a] \\
\text{insert}\ x\ [] & = [x] \\
\text{insert}\ x\ (y:ys) & \mid x \leq y = x : y : ys \\
& \mid \text{otherwise} = y : \text{insert}\ x\ ys \\
\end{align*}
\]

That is, inserting a new element into the empty list gives a singleton list, while for a non-empty list the result depends upon the ordering of the new element \(x\) and the head of the list \(y\). In particular, if \(x \leq y\) then the new element \(x\) is simply prepended to the start of the list, otherwise the head \(y\) becomes the first element of the resulting list, and we then proceed to insert the new element into the tail of the given list. For example:
\[\text{insert} \, 3 \, [1, 2, 4, 5] = \{ \text{applying insert } \} [1 : \text{insert} \, 3 \, [2, 4, 5] = \{ \text{applying insert } \} [1 : 2 : \text{insert} \, 3 \, [4, 5] = \{ \text{applying insert } \} 1 : 2 : 3 : [4, 5] = \{ \text{list notation } \} [1, 2, 3, 4, 5]\]

Using \text{insert} we can now define a function that implements \textit{insertion sort}, in which the empty the empty list is already sorted, and any non-empty list is sorted by inserting its head into the list that results from sorting its tail:

\[
\begin{align*}
\text{isort} & :: \text{Ord } a \Rightarrow [a] \rightarrow [a] \\
\text{isort} \, [] & = [] \\
\text{isort} \, (x : xs) & = \text{insert} \, x \, (\text{isort} \, xs)
\end{align*}
\]

For example:

\[
\begin{align*}
\text{isort} \, [3, 2, 1, 4] = \{ \text{applying isort } \} & \\
\text{insert} \, 3 \, (\text{insert} \, 2 \, (\text{insert} \, 1 \,(\text{insert} \, 4 \,[[]]))) & = \{ \text{applying insert } \} \\
\text{insert} \, 3 \, (\text{insert} \, 2 \, ([1, 4])) & = \{ \text{applying insert } \} \\
\text{insert} \, 3 \, [1, 2, 4] & = \{ \text{applying insert } \} \\
[1, 2, 3, 4] &
\end{align*}
\]

### 6.3 Multiple arguments

Functions with multiple arguments can also be defined using recursion on more than one argument at the same time. For example, the library function \text{zip} that takes two lists and produces a list of pairs is defined as follows:

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

For example:

\[
\begin{align*}
\text{zip} \, ['a', 'b', 'c'] \,[1, 2, 3, 4] = \{ \text{applying zip } \}
\end{align*}
\]
Note that two base cases are required in the definition of `zip`, because either of the two argument lists may be empty. As another example of recursion on multiple arguments, the library function `drop` that removes a given number of elements from the start of a list is defined as follows:

\[
\begin{align*}
drop &: \text{Int} \to [a] \to [a] \\
drop 0 \; xs &= xs \\
drop (n + 1) \; [] &= [] \\
drop (n + 1) \; (_\cdot; \; xs) &= \text{drop} \; n \; xs
\end{align*}
\]

Again, two base cases are required, one for removing zero elements, and one for attempting to remove one or more elements from the empty list.

### 6.4 Multiple recursion

Functions can also be defined using *multiple recursion*, in which a function is applied more than once in its own definition. For example, recall the Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, …, in which the first two numbers are 0 and 1, and each subsequent number is given by adding the preceding two numbers in the sequence. In Haskell, a function that calculates the \( n \)th Fibonacci number for any integer \( n \geq 0 \) can be defined using double recursion as follows:

\[
\begin{align*}
fibonacci & : \text{Int} \to \text{Int} \\
fibonacci \; 0 &= 0 \\
fibonacci \; 1 &= 1 \\
fibonacci \; (n + 2) &= \text{fibonacci} \; n + \text{fibonacci} \; (n + 1)
\end{align*}
\]

As another example, in chapter 1 we showed how to implement another well-known method of sorting a list, called *quicksort*:

\[
\begin{align*}
\text{qsort} &: \text{Ord} \; a \Rightarrow [a] \to [a] \\
\text{qsort} \; [] &= [] \\
\text{qsort} \; (x : xs) &= \text{qsort} \; \text{smaller} + [x] + \text{qsort} \; \text{larger} \\
\quad \text{where} \\
\quad \text{smaller} &= [a \mid a \leftarrow xs, a \leq x] \\
\quad \text{larger} &= [b \mid b \leftarrow xs, b > x]
\end{align*}
\]

That is, the empty list is already sorted, and any non-empty list can be sorted by placing its head between the two lists that result from sorting those elements of its tail that are `smaller` and `larger` than the head.
6.5 Mutual recursion

Functions can also be defined using *mutual recursion*, in which two or more functions are all defined in terms of each other. For example, consider the library functions *even* and *odd*. For efficiency, these functions are normally defined using the remainder after dividing by two. However, for non-negative integers they can also be defined using mutual recursion:

\[
\begin{align*}
\text{even} & \quad :: \quad \text{Int} \rightarrow \text{Bool} \\
\text{even } 0 & \quad = \quad \text{True} \\
\text{even } (n + 1) & \quad = \quad \text{odd } n \\
\text{odd} & \quad :: \quad \text{Int} \rightarrow \text{Bool} \\
\text{odd } 0 & \quad = \quad \text{False} \\
\text{odd } (n + 1) & \quad = \quad \text{even } n
\end{align*}
\]

That is, zero is even but not odd, and any strictly positive integer is even if its predecessor is odd, and odd if its predecessor is even. For example:

\[
\begin{align*}
\text{even } 4 & \quad = \quad \{ \text{applying \textit{even} } \} \\
\text{odd } 3 & \quad = \quad \{ \text{applying \textit{odd} } \} \\
\text{even } 2 & \quad = \quad \{ \text{applying \textit{even} } \} \\
\text{odd } 1 & \quad = \quad \{ \text{applying \textit{odd} } \} \\
\text{even } 0 & \quad = \quad \{ \text{applying \textit{even} } \} \\
\text{True}
\end{align*}
\]

Similarly, functions that select the elements from a list at all even and odd positions (counting from zero) can be defined as follows:

\[
\begin{align*}
\text{evens} & \quad :: \quad [a] \rightarrow [a] \\
\text{evens } [] & \quad = \quad [] \\
\text{evens } (x : xs) & \quad = \quad x : \text{odds } xs \\
\text{odds} & \quad :: \quad [a] \rightarrow [a] \\
\text{odds } [] & \quad = \quad [] \\
\text{odds } (\_ : xs) & \quad = \quad \text{evens } xs
\end{align*}
\]

For example:

\[
\begin{align*}
\text{evens } "abcde" & \quad = \quad \{ \text{applying \textit{evens} } \} \\
\text{\textquotesingle a\textquotesingle : odds } "bcde" & \quad = \quad \{ \text{applying \textit{odds} } \} \\
\text{\textquotesingle a\textquotesingle : evens } "cde"
\end{align*}
\]
= \{ \text{applying } \textit{evens} \} \\
'a' : 'c' : odds "de"
= \{ \text{applying } \textit{odds} \} \\
'a' : 'c' : evens "e"
= \{ \text{applying } \textit{evens} \} \\
'a' : 'c' : 'e' : odds []
= \{ \text{applying } \textit{odds} \} \\
'a' : 'c' : 'e' : []
= \{ \text{string notation} \}
"ace"

Recall that strings in Haskell are actually constructed as lists of characters. Hence, "abcde" is just an abbreviation for ['a', 'b', 'c', 'd', 'e'].

6.6 Advice on recursion

Defining recursive functions is like riding a bicycle: it looks easy when someone else is doing it, may seem impossible when you first try to do it yourself, but becomes simple and natural with practice. In this section we offer some advice for defining functions in general, and recursive functions in particular, using a five step process that we introduce by means of examples.

Example - \textit{product}

As a simple first example, we show how the definition given earlier in this chapter for the library function that calculates the \textit{product} of a list of numbers can be constructed in a stepwise manner.

Step 1: define the type

Thinking about types is very helpful when defining functions, so it is good practice to define the type of a function prior to starting to define the function itself. In this case, we begin with the type

\[
\textit{product} :: [\textit{Int}] \rightarrow \textit{Int}
\]

that states that \textit{product} takes a list of integers and produces a single integer. As in this example, it is often useful to begin with a simple type, which can be refined or generalised later on as appropriate.

Step 2: enumerate the cases

For most types of argument, there are a number of standard cases to consider. For lists, the standard cases are the empty list and non-empty lists, so we can write down to the following \textit{skeleton} definition using pattern matching:

\[
\textit{product} [] = \\
\textit{product} (n : ns) =
\]
For non-negative integers, the standard cases are 0 and \( n + 1 \), for logical values they are \( \text{False} \) and \( \text{True} \), and so on. As with the type, we may need to refine the cases later on, but it is useful to begin with the standard cases.

**Step 3: define the simple cases**

By definition, the product of zero integers is one, because one is the identity for multiplication. Hence it is straightforward to define the empty list case:

\[
\begin{align*}
\text{product} \; [\;] & = 1 \\
\text{product} \; (n : \text{ns}) & =
\end{align*}
\]

As in this example, the simple cases often become base cases.

**Step 4: define the other cases**

How can we calculate the product of a non-empty list of integers? For this step, it is useful to first consider the *ingredients* that can be used, such as the function itself (\( \text{product} \)), the arguments (\( n \) and \( \text{ns} \)), and library functions of relevant types (+, −, *, and so on.) In this case, we simply multiply the first integer and the product of the remaining list of integers:

\[
\begin{align*}
\text{product} \; [\;] & = 1 \\
\text{product} \; (n : \text{ns}) & = n \times \text{product} \; \text{ns}
\end{align*}
\]

As in this example, the other cases often become recursive cases.

**Step 5: generalise and simplify**

Once a function has been defined using the above process, it often becomes clear that it can be generalised and simplified. For example, the function \( \text{product} \) does not depend on the precise kind of numbers to which it is applied, so its type can be generalised from integers to any numeric type:

\[
\text{product} :: \text{Num} \; a \Rightarrow [a] \rightarrow a
\]

In terms of simplification, we will see in chapter 7 that the pattern of recursion used in \( \text{product} \) is encapsulated by a library function called \( \text{foldr} \), using which \( \text{product} \) can be redefined by a single equation:

\[
\text{product} = \text{foldr} (\times) \; 1
\]

In conclusion, our final definition for \( \text{product} \) is as follows:

\[
\begin{align*}
\text{product} :: \text{Num} \; a \Rightarrow [a] \rightarrow a \\
\text{product} & = \text{foldr} (\times) \; 1
\end{align*}
\]

This is precisely the definition from the standard prelude in Appendix B, except that for efficiency reasons the use of \( \text{foldr} \) is replaced by the related library function \( \text{foldl} \), which is discussed in chapter 7.
Example - \textit{drop}

As a more substantial example, we now show how the definition given earlier for the library function \textit{drop} that removes a given number of elements from the start of a list can be constructed using the five step process.

\textbf{Step 1: define the type}

Let us begin with a type that states that \textit{drop} takes an integer and a list of values of some type \( a \), and produces another list of such values:

\[
drop \quad : \quad \text{Int} \rightarrow [a] \rightarrow [a]
\]

Note that we have made four decisions in defining this type: using integers rather than a more general numeric type, for simplicity; using currying rather than taking the arguments as a pair, for flexibility (see section 3.6); supplying the integer argument before the list argument, for readability (\textit{drop} \( n \) \( xs \) can be read as “\textit{drop} \( n \) elements from \( xs \)”); and finally, making the function polymorphic in the type of the list elements, for generality.

\textbf{Step 2: enumerate the cases}

As there are two standard cases for the integer argument (0 and \( n + 1 \)) and two for the list argument ([ ] and \( x : xs \)), writing down a skeleton definition using pattern matching requires four cases in total:

\[
\begin{align*}
\text{drop} \hspace{1em} 0 \hspace{1em} [ ] & = [ ] \\
\text{drop} \hspace{1em} 0 \hspace{1em} (x : xs) & = \\
\text{drop} \hspace{1em} (n + 1) \hspace{1em} [ ] & = [ ] \\
\text{drop} \hspace{1em} (n + 1) \hspace{1em} (x : xs) & =
\end{align*}
\]

\textbf{Step 3: define the simple cases}

By definition, removing zero elements from the start of any list gives the same list, so it is straightforward to define the first two cases:

\[
\begin{align*}
\text{drop} \hspace{1em} 0 \hspace{1em} [ ] & = [ ] \\
\text{drop} \hspace{1em} 0 \hspace{1em} (x : xs) & = x : xs \\
\text{drop} \hspace{1em} (n + 1) \hspace{1em} [ ] & = [ ] \\
\text{drop} \hspace{1em} (n + 1) \hspace{1em} (x : xs) & =
\end{align*}
\]

Attempting to removing one or more elements from the empty list is invalid, so the third case could be omitted, which would result in an error being produced if this situation arises. In practice, however, we choose the avoid the production of an error by returning the empty list in this case:

\[
\begin{align*}
\text{drop} \hspace{1em} 0 \hspace{1em} [ ] & = [ ] \\
\text{drop} \hspace{1em} 0 \hspace{1em} (x : xs) & = x : xs \\
\text{drop} \hspace{1em} (n + 1) \hspace{1em} [ ] & = [ ] \\
\text{drop} \hspace{1em} (n + 1) \hspace{1em} (x : xs) & =
\end{align*}
\]
Step 4: define the other cases

How can we remove one or more elements from a non-empty list? By simply removing one less element from the tail of the list:

\[
drop 0 \; [ ] \quad = \quad [ ] \\
drop 0 \; (x : xs) \quad = \quad x : xs \\
drop (n + 1) \; [ ] \quad = \quad [ ] \\
drop (n + 1) \; (x : xs) \quad = \quad drop \; n \; xs
\]

Step 5: generalise and simplify

Because the function \( \text{drop} \) does not depend on the precise kind of integers to which it is applied, its type can be generalised to any integral type, of which \( \text{Int} \) and \( \text{Integer} \) are the standard instances:

\[
drop \quad :: \quad \text{Integral} \; b \Rightarrow b \rightarrow [a] \rightarrow [a]
\]

For efficiency reasons, however, this generalisation is not in fact made in the standard prelude, as already mentioned in section 3.9. In terms of simplification, the first two equations for \( \text{drop} \) can be combined into a single equation that states that removing zero elements from any list gives the same list:

\[
drop 0 \; xs \quad = \quad xs \\
drop (n + 1) \; [ ] \quad = \quad [ ] \\
drop (n + 1) \; (x : xs) \quad = \quad drop \; n \; xs
\]

Moreover, the variable \( x \) in the last equation can be replaced by the wildcard pattern \( _\_ \), because this variable is not used in the body of the equation:

\[
drop 0 \; xs \quad = \quad xs \\
drop (n + 1) \; [ ] \quad = \quad [ ] \\
drop (n + 1) \; (_ : xs) \quad = \quad drop \; n \; xs
\]

We might similarly expect \( n \) in the second equation to be replaced by \( _\_ \) but this would make the definition invalid, because patterns of the form \( n + k \) require that \( n \) is a variable. This constraint could be avoided by replacing the entire pattern \( n + 1 \) in the second equation by \( _\_ \) but this would change the behaviour of the function. For example, evaluating \( \text{drop} \; (-1) \; [ ] \) would then produce the empty list whereas it currently produces an error, because \( _\_ \) can match any integer whereas \( n + 1 \) only matches integers \( \geq 1 \).

In conclusion, our final definition for \( \text{drop} \) is as follows, which is precisely the definition from the standard prelude in Appendix B:

\[
drop \quad :: \quad \text{Int} \rightarrow [a] \rightarrow [a]
\]

\[
drop 0 \; xs \quad = \quad xs \\
drop (n + 1) \; [ ] \quad = \quad [ ] \\
drop (n + 1) \; (_ : xs) \quad = \quad drop \; n \; xs
\]
Example - \textit{init}

As a final example, let us consider how the definition for library function \textit{init}
that removes the last element from a non-empty list can be constructed.

\textbf{Step 1: define the type}

We begin with a type that states that \textit{init} takes a list of values of some type \(a\),
and produces another list of such values:

\[
\textit{init} :: [a] \rightarrow [a]
\]

\textbf{Step 2: enumerate the cases}

As the empty list is not a valid argument for \textit{init}, writing down a skeleton
definition using pattern matching requires just one case:

\[
\textit{init} (x : xs) =
\]

\textbf{Step 3: define the simple cases}

Whereas in the previous two examples this step was straightforward, a little
more thought is required for the function \textit{init}. By definition, however, removing
the last element from a list with one element gives the empty list, so we
can introduce a guard to handle this simple case:

\[
\textit{init} (x : xs) \mid \text{null } xs = []
\]

\[
\mid \text{otherwise } =
\]

Recall that the library function \textit{null} decides if a list is empty or not.

\textbf{Step 4: define the other cases}

How can we remove the last element from a list with at least two elements?
By simply retaining the head and removing the last element from the tail:

\[
\textit{init} (x : xs) \mid \text{null } xs = []
\]

\[
\mid \text{otherwise } = x : \text{init } xs
\]

\textbf{Step 5: generalise and simplify}

The type for \textit{init} is already as general as possible, but the definition itself can
now be simplified by using pattern matching rather than guards, and by using
a wildcard pattern in the first equation rather than a variable:

\[
\textit{init} :: [a] \rightarrow [a]
\]

\[
\textit{init} [\_] = []
\]

\[
\textit{init} (x : xs) = x : \text{init } xs
\]

Again, this is precisely the definition from the standard prelude.
6.7 Chapter remarks

The recursive definitions presented in this chapter emphasise clarity, but many can be improved in terms of efficiency, as discussed in chapter 13. The five step process for defining functions is based upon [6].

6.8 Exercises

1. Define the exponentiation operator ↑ for non-negative integers using the same pattern of recursion as the multiplication operator *, and show how $2 ↑ 3$ is evaluated using your definition.

2. Using the definitions given in this chapter, show how length $[1, 2, 3]$, drop $3 [1, 2, 3, 4, 5]$, and init $[1, 2, 3]$ are evaluated.

3. Without looking at the definitions from the standard prelude, define the following library functions using recursion:

   - Decide if all logical values in a list are True:
     \[
     \text{and} :: [\text{Bool}] \to \text{Bool}
     \]

   - Concatenate a list of lists:
     \[
     \text{concat} :: [[a]] \to [a]
     \]

   - Produce a list with $n$ identical elements:
     \[
     \text{replicate} :: \text{Int} \to a \to [a]
     \]

   - Select the $n$th element of a list:
     \[
     (!!) :: [a] \to \text{Int} \to a
     \]

   - Decide if a value is an element of a list:
     \[
     \text{elem} :: \text{Eq } a \Rightarrow a \to [a] \to \text{Bool}
     \]

   Note: most of these functions are in fact defined in the prelude using other library functions, rather than using explicit recursion.

4. Define a recursive function \text{merge} :: \text{Ord } a \Rightarrow [a] \to [a] \to [a] that merges two sorted lists to give a single sorted list. For example:

   \[
   > \text{merge} \ [2, 5, 6] \ [1, 3, 4] \\
   [1, 2, 3, 4, 5, 6]
   \]

   Note: your definition should not use other functions on sorted lists such as \text{insert} or \text{isort}, but should be defined using explicit recursion.
5. Using `merge`, define a recursive function `msort :: Ord a ⇒ [a] → [a]` that implements `merge sort`, in which the empty list and lists with one element are already sorted, and any other list is sorted by merging together the two lists that result from sorting the two halves of the list separately.

Hint: first define a function `halve :: [a] → [[a],[a]]` that splits a list into two halves whose length differs by at most one.

6. Using the five step process, define the library functions that calculate the `sum` of a list of numbers, `take` a given number of elements from the start of a list, and select the `last` element of a non-empty list.
Chapter 7

Higher-Order Functions

In this chapter we introduce higher-order functions, which allow common programming patterns to be encapsulated as functions. We start by explaining what higher-order functions are and why they are useful, then introduce a number of standard higher-order functions for processing lists, consider function composition, and conclude by developing a string transmitter.

7.1 Basic concepts

As we have seen in previous chapters, functions with multiple arguments are usually defined in Haskell using the notion of currying. That is, the arguments are taken one at a time by exploiting the fact that functions can return functions as results. For example, the definition

\[
\text{add} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
\text{add} \ x \ y = x + y
\]

means

\[
\text{add} :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \\
\text{add} = \lambda x \rightarrow (\lambda y \rightarrow x + y)
\]

and states that \text{add} is a function that takes an integer \(x\) and returns a function, which in turn takes another integer \(y\) and returns their sum \(x + y\). In Haskell, it is also permissible to define functions that take functions as arguments. For example, a function that takes a function and a value, and returns the result of applying the function twice to the value, can be defined as follows:

\[
\text{twice} :: (a \rightarrow a) \rightarrow a \rightarrow a \\
\text{twice} \ f \ x = f (f \ x)
\]

For example:

\[
> \text{twice} \ (**2) \ 3 \\
12
\]

\[
> \text{twice} \ \text{reverse} \ [1,2,3] \\
[1,2,3]
\]
Moreover, because \textit{twice} is a curried function, it can be partially applied with just one argument to build other useful functions. For example, a function that quadruples a number is given by \textit{twice} \((\ast 2)\), and the fact that reversing a (finite) list twice has no effect is given by the functional equation \(\textit{twice} \textit{reverse} = \textit{id}\), where \textit{id} is the identity function defined by \(\textit{id} \ x = x\).

Formally speaking, a function that takes a function as an argument or returns a function as a result is called \textit{higher-order}. In practice, however, because the term curried already exists for returning functions as results, the term higher-order is often just used for taking functions as arguments. It is this latter interpretation that is the subject of this chapter.

Using higher-order functions considerably increases the power of Haskell, by allowing common programming patterns, such as applying a function twice, to be encapsulated as functions within the language itself. More generally, special purpose or “domain specific” languages can often be defined as collections of higher-order functions. For example, in this chapter we present a simple language for processing lists, and in chapter 8 we do the same for building parsers. Moreover, as we shall see in chapter 13, using higher-order functions can also simplify the process of reasoning about programs.

## 7.2 Processing lists

The standard prelude defines a number of useful higher-order functions for processing lists. For example, the function \textit{map} applies a function to all elements of a list, and can be defined using a list comprehension as follows:

\[
\begin{align*}
\textit{map} & \quad :: \quad (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\textit{map} \ f \ xs & \quad = \quad [f \ x \mid x \leftarrow xs]
\end{align*}
\]

That is, \textit{map} \(f\) \(xs\) returns the list of all values \(f\) \(x\) such that \(x\) is an element of the list \(xs\). For example, we have:

\[
\begin{align*}
> \ \textit{map} \ (+1) \ [1, 3, 5, 7] & \quad [2, 4, 6, 8] \\
> \ \textit{map} \ \textit{isDigit} \ [\texttt{a}', \texttt{1}', \texttt{b}', \texttt{2}'] & \quad [\texttt{False}, \texttt{True}, \texttt{False}, \texttt{True}] \\
> \ \textit{map} \ \textit{reverse} \ [\texttt{one}'', \texttt{two}'', \texttt{three}''] & \quad [\texttt{eno}'', \texttt{owt}'', \texttt{eerht}'']
\end{align*}
\]

There are three further points to note about \textit{map}. First of all, it is a polymorphic function that can be applied to lists of any type, as are most higher-order functions on lists. Secondly, it can be applied to itself to process nested lists. For example, the function \textit{map} \((\textit{map} \ (+1))\) increments each number in a list of lists of numbers, as shown in the following calculation:
\[\text{map} \ (\text{map} \ (+1)) \ [[1, 2, 3], [4, 5]]\]
\[= \begin{cases} \text{applying the outer map} \\ \text{map} \ (+1) \ [1, 2, 3], \ \text{map} \ (+1) \ [4, 5] \end{cases}\]
\[= \begin{cases} \text{applying map} \\ [[2, 3, 4], [5, 6]] \end{cases}\]

And finally, the function \text{map} can also be defined using recursion:

\[
\begin{align*}
\text{map} \ f \ [] &= [] \\
\text{map} \ f \ (x : xs) &= f \ x : \text{map} \ f \ xs
\end{align*}
\]

That is, applying a function to all elements of the empty list gives the empty list, while for a non-empty list the function is simply applied to the head of the list, and we then proceed to apply the function to all elements of the tail. The original definition for \text{map} using a list comprehension is simpler, but the recursive definition is preferable for reasoning purposes in chapter 13. Another useful library function is \text{filter}, which selects all elements of a list that satisfy a predicate, where a \textit{predicate} (or property) is a function that returns a logical value. As with \text{map}, the function \text{filter} also has a simple definition using a list comprehension:

\[
\begin{align*}
\text{filter} &:: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
\text{filter} \ p \ xs &= [x \mid x \leftarrow xs, p \ x]
\end{align*}
\]

That is, \text{filter} \ p \ xs returns the list of all values \(x\) such that \(x\) is an element of the list \(xs\) and the value of \(p \ x\) is \text{True}. For example:

\[
\begin{align*}
> & \text{filter} \ \text{even} \ [1 \ldots 10] \\
& [2, 4, 6, 8, 10] \\
> & \text{filter} \ (>5) \ [1 \ldots 10] \\
& [6, 7, 8, 9, 10] \\
> & \text{filter} \ (\neq \ 'a') \ "\text{one\two\three}" \\
& "\text{one\two\three}"
\end{align*}
\]

As with \text{map}, the function \text{filter} can be applied to lists of any type, and can be defined using recursion for the purposes of reasoning:

\[
\begin{align*}
\text{filter} \ p \ [] &= [] \\
\text{filter} \ p \ (x : xs) \mid p \ x &= x : \text{filter} \ p \ xs \\
\text{filter} \ p \ (x : xs) \mid \text{otherwise} &= \text{filter} \ p \ xs
\end{align*}
\]

That is, selecting all elements that satisfy a predicate from the empty list gives the empty list, while for a non-empty list the result depends upon whether the head satisfies the predicate. If it does then the head is retained and we then proceed to filter elements from the tail of the list, otherwise the head is discarded and we simply filter elements from the tail.
The functions \textit{map} and \textit{filter} are often used together in programs, with \textit{filter} being used to select certain elements from a list, each of which is then transformed using \textit{map}. For example, a function that returns the sum of the squares of the even integers from a list could be defined as follows:

\begin{align*}
\text{sums square even} & : \quad [\text{Int}] \to \text{Int} \\
\text{sums square even ns} & = \quad \text{sum} (\text{map} (\lambda x. x^2) (\text{filter even ns}))
\end{align*}

We conclude this section by illustrating a number of other higher-order functions for processing lists that are defined in the standard prelude:

- Decide if all elements of a list satisfy a predicate:

  \begin{verbatim}
> all even [2, 4, 6, 8] True
\end{verbatim}

- Decide if any element of a list satisfies a predicate:

  \begin{verbatim}
> any odd [2, 4, 6, 8] False
\end{verbatim}

- Select elements from a list while they satisfy a predicate:

  \begin{verbatim}
> takeWhile isLower "hello, there" "hello"
\end{verbatim}

- Remove elements from a list while they satisfy a predicate:

  \begin{verbatim}
> dropWhile isLower "hello, there" "there"
\end{verbatim}

\subsection{7.3 The \textit{foldr} function}

Many functions that take a list as their argument can be defined using the following simple pattern of recursion on lists:

\begin{align*}
f [] & = v \\
f (x : xs) & = x \oplus f xs
\end{align*}

That is, the function maps the empty list to some value \(v\), and any non-empty list to some operator \(\oplus\) applied to the head of the list and the result of recursively processing the tail. For example, a number of familiar library
functions on lists can be defined using this pattern:

\[
\begin{align*}
\text{sum} \; \text{[]} & = 0 \\
\text{sum} \; (x : xs) & = x + \text{sum} \; xs \\
\text{product} \; \text{[]} & = 1 \\
\text{product} \; (x : xs) & = x \ast \text{product} \; xs \\
\text{or} \; \text{[]} & = \text{False} \\
\text{or} \; (x : xs) & = x \lor \text{or} \; xs \\
\text{and} \; \text{[]} & = \text{True} \\
\text{and} \; (x : xs) & = x \land \text{and} \; xs
\end{align*}
\]

The higher-order library function \texttt{foldr} (abbreviating “fold right”) encapsulates this pattern of recursion for defining functions on lists, with the operator \( \oplus \) and the value \( v \) as arguments. For example, using \texttt{foldr} the four definitions above can be rewritten more compactly as follows:

\[
\begin{align*}
\text{sum} & = \text{foldr} \; (+) \; 0 \\
\text{product} & = \text{foldr} \; (*) \; 1 \\
\text{or} & = \text{foldr} \; (\lor) \; \text{False} \\
\text{and} & = \text{foldr} \; (\land) \; \text{True}
\end{align*}
\]

(Recall that operators must be parenthesised when used as arguments.) These new definitions could also include explicit list arguments, as in \( \text{sum} \; xs = \text{foldr} \; (+) \; 0 \; xs \), but we prefer the above definitions in which these arguments are made implicit using partial application because they are simpler.

The \texttt{foldr} function itself is defined using recursion:

\[
\begin{align*}
\text{foldr} & : (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\
\text{foldr} \; f \; v \; \text{[]} & = v \\
\text{foldr} \; f \; v \; (x : xs) & = f \; x \; (\text{foldr} \; f \; v \; xs)
\end{align*}
\]

That is, the function \( \text{foldr} \; f \; v \) maps the empty list to the value \( v \), and any non-empty list to the function \( f \) applied to the head of the list and the recursively processed tail. In practice, however, it is best to think of the behaviour of \( \text{foldr} \; f \; v \) in a non-recursive manner, as simply replacing each cons operator in a list by the function \( f \), and the empty list at the end by the value \( v \). For example, applying the function \( \text{foldr} \; (+) \; 0 \) to the list

\[
1 : (2 : (3 : []))
\]

gives the result

\[
1 + (2 + (3 + 0))
\]

in which : and [] have been replaced by + and 0, respectively. Hence, the definition \( \text{sum} = \text{foldr} \; (+) \; 0 \) states that summing a list of numbers amounts to replacing each cons by addition and the empty list by zero.
Even though foldr encapsulates a simple pattern of recursion, it can be used to define many more functions than might first be expected. First of all, recall the following definition for the library function length:

\[
\begin{align*}
\text{length} & : [a] \to \text{Int} \\
\text{length} \; [] & = 0 \\
\text{length} \; (\text{\textup{:}} \; x : \text{xs}) & = 1 + \text{length} \; \text{xs}
\end{align*}
\]

For example, applying length to the list

\[1 : (2 : (3 : []))]\]

gives the result

\[1 + (1 + (1 + 0))\]

That is, calculating the length of a list amounts to replacing each cons by the function that adds one to its second argument, and the empty list by zero. Hence, the definition for length can be rewritten using foldr:

\[
\text{length} = \text{foldr} \; (\lambda n \to 1 + n) \; 0
\]

Now let us consider the library function that reverses a list, which can be defined using explicit recursion as follows:

\[
\begin{align*}
\text{reverse} & : [a] \to [a] \\
\text{reverse} \; [] & = [] \\
\text{reverse} \; (x : \text{xs}) & = \text{reverse} \; \text{xs} \; \text{\textup{\textbf{:+}}} \; [x]
\end{align*}
\]

For example, applying reverse to the list

\[1 : (2 : (3 : []))]\]

gives the result

\[(([] \; \text{\textup{\textbf{:+}}} \; [3]) \; \text{\textup{\textbf{:+}}} \; [2]) \; \text{\textup{\textbf{:+}}} \; [1]\]

It is perhaps not immediately clear from the definition, or the example, how reverse can be defined using foldr. However, if we define a function snoc \(x \; \text{xs} = \text{xs} \; \text{\textup{\textbf{:+}}} \; [x]\) that adds a new element at the end of a list rather than at the start (“snoc” is cons backwards), then reverse can be redefined as

\[
\begin{align*}
\text{reverse} \; [] & = [] \\
\text{reverse} \; (x : \text{xs}) & = \text{snoc} \; x \; (\text{reverse} \; \text{xs})
\end{align*}
\]

from which a definition using foldr is then immediate:

\[
\text{reverse} = \text{foldr} \; \text{snoc} \; []
\]
That is, reversing a list amounts to replacing each cons by \textit{snoc} and the empty list by itself. The append operator \texttt{+} used in the definition of \textit{snoc} can itself be defined in a particularly compact manner using \textit{foldr}:

\[
(\texttt{+} \ ys) = \textit{foldr} (:) ys
\]

That is, appending \textit{ys} to the end of a list amounts to replacing each cons in the list by itself, and the empty list at the end by \textit{ys}.

We conclude this section by noting that the name “fold \textit{right}” reflects the use of an operator that is assumed to associate to the right. For example, evaluating \textit{foldr} \texttt{(+)} \texttt{0} [1, 2, 3] gives the result \((1 + (2 + (3 + 0)))\), in which the bracketing specifies that addition here is assumed to associate to the right. More generally, the behaviour of \textit{foldr} can be summarised as follows:

\[
\textit{foldr} (\texttt{+}) \texttt{v} [x_0, x_1, \ldots, x_n] = x_0 \texttt{+} (x_1 \texttt{+} (\ldots (x_n \texttt{+} \texttt{v}) \ldots))
\]

### 7.4 The \textit{foldl} function

It is also possible to define a recursive function on lists using an operator that is assumed to associate to the \textit{left}. For example, the function \textit{sum} can be redefined in this manner by using an auxiliary function \textit{sum'} that takes an extra argument \texttt{v} that is used to accumulate the final result:

\[
\textit{sum} = \textit{sum'} 0 \\
\textbf{where} \\
\text{sum'} \texttt{v} [] = \texttt{v} \\
\text{sum'} \texttt{v} (x : xs) = \text{sum'} (\texttt{v + x}) xs
\]

For example:

\[
\text{sum} \ [1, 2, 3] \\
= \{ \text{applying } \text{sum} \} \\
\text{sum'} 0 \ [1, 2, 3] \\
= \{ \text{applying } \text{sum'} \} \\
\text{sum'} (0 + 1) \ [2, 3] \\
= \{ \text{applying } \text{sum'} \} \\
\text{sum'} ((0 + 1) + 2) \ [3] \\
= \{ \text{applying } \text{sum'} \} \\
\text{sum'} (((0 + 1) + 2) + 3) [] \\
= \{ \text{applying } \text{sum'} \} \\
((0 + 1) + 2) + 3
\]

The bracketing in the final result specifies that addition is now assumed to associate to the left. Of course, the order of association does not affect the value of the result in this case, because addition is \textit{associative}. That is, \(x + (y + z) = (x + y) + z\) for any numbers \(x, y\) and \(z\). 

85
Generalising from the \textit{sum} example, many functions on lists can be defined using the following simple pattern of recursion:

\[
\begin{align*}
    f \ v \ [] & = v \\
    f \ v \ (x : xs) & = f \ (v \oplus x) \ xs
\end{align*}
\]

That is, the function maps the empty list to the \textit{accumulator} value \(v\), and any non-empty list to the result of recursively processing the tail using a new accumulator value obtained by applying some operator \(\oplus\) to the current value and the head of the list. The higher-order library function \textit{foldl} (abbreviating “fold left”) encapsulates this pattern of recursion, with the operator \(\oplus\) and the accumulator \(v\) as arguments. For example, using \textit{foldl} the above definition for \textit{sum} can be rewritten more compactly as follows:

\[
\text{sum} = \text{foldl} \ (+) \ 0
\]

Similarly, we have:

\[
\begin{align*}
    \text{product} & = \text{foldl} \ (*) \ 1 \\
    \text{or} & = \text{foldl} \ (\lor) \ \text{False} \\
    \text{and} & = \text{foldl} \ (\land) \ \text{True}
\end{align*}
\]

The other \textit{foldr} examples from the previous section can also be redefined using \textit{foldl}, by supplying the appropriate operators:

\[
\begin{align*}
    \text{length} & = \text{foldl} \ (\lambda n \rightarrow n + 1) \ 0 \\
    \text{reverse} & = \text{foldl} \ (\lambda xs \ x \rightarrow x : xs) \ [] \\
    (xs \oplus) & = \text{foldl} \ (\lambda ys \ y \rightarrow ys \oplus [y]) \ xs
\end{align*}
\]

For example, with these new definitions:

\[
\begin{align*}
    \text{length} \ [1, 2, 3] & = ((0 + 1) + 1) + 1 \\
    \text{reverse} \ [1, 2, 3] & = 3 : (2 : (1 : [])) \\
    [1, 2, 3] \oplus [4, 5, 6] & = (([1, 2, 3] \oplus [4]) \oplus [5]) \oplus [6]
\end{align*}
\]

When a function can be defined using both \textit{foldr} and \textit{foldl}, as in the above examples, the choice of which definition is preferable is usually made on grounds of efficiency, and requires careful consideration of the evaluation mechanism underlying Haskell. We defer a discussion on this issue until we consider the general problem of reasoning about programs in chapter 13.

The \textit{foldl} function itself is defined using recursion:

\[
\begin{align*}
    \text{foldl} & :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \\
    \text{foldl} \ f \ v \ [] & = v \\
    \text{foldl} \ f \ v \ (x : xs) & = \text{foldl} \ f \ (f \ v \ x) \ xs
\end{align*}
\]

In practice, however, as with \textit{foldr} it is best to think of the behaviour of \textit{foldl} in a non-recursive manner, in terms of an operator \(\oplus\) that is assumed to associate to the left, as summarised by the following equation:

\[
\text{foldl} \ (\oplus) \ v \ [x_0, x_1, \ldots, x_n] = (\cdots ((v \oplus x_0) \oplus x_1) \cdots) \oplus x_n
\]
7.5 The composition operator

The higher-order library operator \( \circ \) returns the composition of two functions as a single function, and can be defined as follows:

\[
(\circ) :: (b \to c) \to (a \to b) \to (a \to c)
\]

\[
f \circ g = \lambda x \to f (g \, x)
\]

That is, \( f \circ g \) (read as “\( f \) composed with \( g \)” ) is the function that takes an argument \( x \), applies the function \( g \) to this argument, and applies the function \( f \) to the result. This operator could also be defined by \( (f \circ g) \, x = f (g \, x) \). However, we prefer the above definition in which the \( x \) argument is shunted to the body of the definition using a lambda expression, because it makes explicit the idea that composition returns a function as its result.

Composition can be used to simplify nested function applications, by reducing parentheses and avoiding the need to explicitly refer to the initial argument. For example, using composition the definitions

\[
\begin{align*}
\text{odd} \ n &= \neg (\text{even} \ n) \\
\text{twice} \ f \ x &= f \ (f \ x) \\
\text{sumsqr} \ \text{even} \ ns &= \text{sum} \ (\text{map} \ (\uparrow 2) \ (\text{filter} \ \text{even} \ ns))
\end{align*}
\]

can be rewritten more simply:

\[
\begin{align*}
\text{odd} &= \neg \circ \ \text{even} \\
\text{twice} \ f &= f \circ f \\
\text{sumsqr} &= \text{sum} \circ \text{map} \ (\uparrow 2) \circ \text{filter} \ \text{even}
\end{align*}
\]

The last definition exploits the fact that composition is associative. That is, \( f \circ (g \circ h) = (f \circ g) \circ h \) for any functions \( f, \, g \) and \( h \) of the appropriate types. Hence, in a composition of three of more functions, as in \( \text{sumsqr} \) , there is no need to include parentheses to indicate the order of association, because associativity ensures that this does not affect the result.

Composition also has an identity, given by the identity function:

\[
\begin{align*}
id :: a \to a \\
id &= \lambda x \to x
\end{align*}
\]

That is, \( id \) is the function that simply returns its argument unchanged, and has the property that \( id \circ f = f \) and \( f \circ id = f \) for any function \( f \). The identity function is often useful when reasoning about programs (see chapter 13), and in programming itself provides an appropriate starting point for a sequence of compositions For example, using the identity function, the composition of a list of functions can be defined as follows:

\[
\begin{align*}
\text{compose} :: [a \to a] \to (a \to a) \\
\text{compose} &= \text{foldr} \ (\circ) \ \text{id}
\end{align*}
\]
7.6 String transmitter

We conclude this chapter with an extended example. Consider the problem of simulating the transmission of a string using zeros and ones. More precisely, we seek to define a function that converts a string of characters into a list of zeros and ones, together with a function that performs the opposite conversion.

7.6.1 Binary numbers

As a consequence of having ten fingers, people normally find it most convenient to use numbers written in base-ten or decimal notation. A decimal number comprises a sequence of digits in the range zero to nine, in which the last digit has a weight of one, and successive digits as we move to the left in the number increase in weight by a factor of ten. For example, the decimal number 2345 can be understood in these terms as follows:

\[
2345 = (1000 \times 2) + (100 \times 3) + (10 \times 4) + (1 \times 5)
\]

That is, 2345 represents the sum of the products of the weights 1000,100,10,1 with the digits 2,3,4,5, which evaluates to the integer 2345.

In contrast, computers normally find it more convenient to use numbers written in the more primitive base-two or binary notation. A binary number comprises a sequence of zeros and ones, called binary digits or bits, in which successive bits as we move to the left increase in weight by a factor of two. For example, the binary number 1101 can be understood as follows:

\[
1101 = (8 \times 1) + (4 \times 1) + (2 \times 0) + (1 \times 1)
\]

That is, 1101 represents the sum of the products of the weights 8,4,2,1 with the bits 1,1,0,1, which evaluates to the integer 13.

To simplify the definition of certain functions, we assume for the remainder of this example that binary numbers are written in reverse order to normal. For example, 1101 would now be written as 1011, with successive bits as we move to the right increasing in weight by a factor of two:

\[
1011 = (1 \times 1) + (2 \times 0) + (4 \times 1) + (8 \times 1)
\]

7.6.2 Base conversion

To make the types of the functions that we define more meaningful, we define a type for bits as a synonym for the type of integers:

\[
\text{type } Bit = Int
\]

A binary number, represented as a list of such bits, can be converted into an integer by simply evaluating the required weighted sum:

\[
\begin{align*}
\text{bin2int} & \quad :: \quad [Bit] \rightarrow Int \\
\text{bin2int bits} & \quad = \quad \text{sum} \left\{ w \times b \mid (w, b) \leftarrow \text{zip weights bits} \right\} \\
\text{where} \quad & \quad \text{weights} = \text{iterate} \ (\times 2) \ 1
\end{align*}
\]
The higher-order library function *iterate* behaves as follows, producing a list by applying a function an increasing number of times to a value:

\[ \text{iterate } f \ x = [x, f \ x, f (f \ x), f (f (f \ x)), \cdots] \]

Hence the expression *iterate* (*\(\ast\)2) 1 used within the definition of *bin2int* produces the list of weights \([1, 2, 4, 8, \cdots]\). This list is notionally infinite, but under lazy evaluation as discussed in chapter 12, only as many elements as required by the context in which it is used — in this case zipping with the list of bits — will actually be produced. For example:

\[
> \text{bin2int} \ [1, 0, 1, 1] \\
13
\]

There is, however, a simpler way to define *bin2int*, which can be revealed with the aid of a little algebra. Consider an arbitrary four-bit binary number \([a, b, c, d]\). Applying *bin2int* to this list will produce the weighted sum

\[(1 \ast a) + (2 \ast b) + (4 \ast c) + (8 \ast d)\]

which can be restructured as follows:

\[
(1 \ast a) + (2 \ast b) + (4 \ast c) + (8 \ast d) \\
= \{ \text{simplifying } 1 \ast a \} \\
a + (2 \ast b) + (4 \ast c) + (8 \ast d) \\
= \{ \text{factoring out } 2 \ast \} \\
a + 2 \ast (b + (2 \ast c) + (4 \ast d)) \\
= \{ \text{factoring out } 2 \ast \} \\
a + 2 \ast (b + 2 \ast (c + (2 \ast d))) \\
= \{ \text{complicating } d \} \\
a + 2 \ast (b + 2 \ast (c + 2 \ast (d + (2 \ast 0))))
\]

The final result shows that converting a list of bits \([a, b, c, d]\) into an integer amounts to replacing each cons by the function that adds its first argument to twice its second argument, and replacing the empty list by zero. More generally, we conclude that *bin2int* can be rewritten using *foldr*:

\[
\text{bin2int} = \text{foldr } (\lambda x \ y \to x + 2 \ast y) \text{ 0}
\]

Now let us consider the opposite conversion, from a non-negative integer into a binary number. This can be achieved by repeatedly dividing the integer by two and taking the remainder, until the integer becomes zero. For example, starting with the integer 13 we proceed as follows:

\[
\begin{align*}
13 & \text{ divided by } 2 = 6 \text{ remainder } 1 \\
6 & \text{ divided by } 2 = 3 \text{ remainder } 0 \\
3 & \text{ divided by } 2 = 1 \text{ remainder } 1 \\
1 & \text{ divided by } 2 = 0 \text{ remainder } 1
\end{align*}
\]
The sequence of remainders, 1011, provides the binary representation of the integer 13. It is easy to implement this procedure using recursion:

\[
\begin{align*}
\text{int2bin} & \quad :: \quad \text{Int} \rightarrow [\text{Bit}] \\
\text{int2bin} \ 0 & \quad = \quad [] \\
\text{int2bin} \ n & \quad = \quad n \ mod \ 2 \ : \ \text{int2bin} \ (n \ div \ 2)
\end{align*}
\]

For example:

\[
> \text{int2bin} \ 13 \\
[1,0,1,1]
\]

We will ensure that all our binary numbers have the same length, eight bits, by using a function \text{make8} that truncates or extends a binary number as appropriate to make it precisely eight bits:

\[
\begin{align*}
\text{make8} & \quad :: \quad [\text{Bit}] \rightarrow [\text{Bit}] \\
\text{make8} \ \text{bits} & \quad = \quad \text{take} \ 8 \ (\text{bits} + \ \text{repeat} \ 0)
\end{align*}
\]

The library function \text{repeat} produces an infinite list of copies of a value, but once again lazy evaluation ensures that only as many elements as required by the context will actually be produced. For example:

\[
> \text{make8} \ [1,0,1,1] \\
[1,0,1,1,0,0,0,0]
\]

### 7.6.3 Transmission

We can now define a function that encodes a string of characters as a list of bits by converting each character into a Unicode number, converting each such number into an eight-bit binary number, and concatenating each of these numbers together to produce a list of bits. Using the higher-order functions \text{map} and composition, this conversion can be implemented as follows:

\[
\begin{align*}
\text{encode} & \quad :: \quad \text{String} \rightarrow [\text{Bit}] \\
\text{encode} & \quad = \quad \text{concat} \circ \text{map} \ (\text{make8} \circ \text{int2bin} \circ \text{ord})
\end{align*}
\]

For example:

\[
> \text{encode} \ "abc" \\
[1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,0,0,0,1,1,0]
\]

To decode a list of bits produced using \text{encode}, we first define a function \text{chop8} that chops such a list up into eight-bit binary numbers:

\[
\begin{align*}
\text{chop8} & \quad :: \quad [\text{Bit}] \rightarrow [[\text{Bit}]] \\
\text{chop8} \ [] & \quad = \quad [] \\
\text{chop8} \ \text{bits} & \quad = \quad \text{take} \ 8 \ \text{bits} : \text{chop8} \ (\text{drop} \ 8 \ \text{bits})
\end{align*}
\]
It is now easy to define a function that decodes a list of bits as a string of characters by chopping the list up, and converting each resulting binary number into a Unicode number and then a character:

\[
\begin{align*}
\text{decode} & : [\text{Bit}] \rightarrow \text{String} \\
\text{decode} & = \text{map} \ (\text{chr} \circ \text{bin2int}) \circ \text{chop8}
\end{align*}
\]

For example:

\[
> \text{decode} \ [1,0,0,0,1,1,0,0,1,0,0,1,0,0,1,1,0,1,0,0,1,1,0] \\
"\text{abc}\"
\]

Finally, we define a function \textit{transmit} that simulates the transmission of a string of characters as a list of bits, using a perfect communication channel that we model using the identity function:

\[
\begin{align*}
\text{transmit} & : \text{String} \rightarrow \text{String} \\
\text{transmit} & = \text{decode} \circ \text{channel} \circ \text{encode} \\
\text{channel} & : [\text{Bit}] \rightarrow [\text{Bit}] \\
\text{channel} & = \text{id}
\end{align*}
\]

For example:

\[
> \text{transmit} \ "\text{higher-order functions are easy}\"
"\text{higher-order functions are easy}\"
\]

### 7.7 Chapter remarks

Further applications of higher-order functions, including the production of computer music, financial contracts, graphical images, hardware descriptions, logic programs, and pretty printers can be found in The Fun of Programming [4]. A more in-depth tutorial on \textit{foldr} is given in [10].

### 7.8 Exercises

1. Show how the list comprehension \([f \; x \mid x \leftarrow xs, p \; x]\) can be re-expressed using the higher-order functions \textit{map} and \textit{filter}.

2. Without looking at the definitions from the standard prelude, define the higher-order functions \textit{all}, \textit{any}, \textit{takeWhile} and \textit{dropWhile}.

3. Redefine the functions \textit{map} \textit{f} and \textit{filter} \textit{p} using \textit{foldr}.

4. Using \textit{foldl}, define a function \textit{dec2int} : [\text{Int}] \rightarrow \text{Int} that converts a decimal number into an integer. For example:

\[
> \text{dec2int} \ [2,3,4,5] \\
2345
\]
5. Explain why the following definition is invalid:

   \[ \text{sumsqa"even} \ = \ \text{compose} \ [\text{sum}, \text{map} \ (\dagger \! 2), \text{filter} \ \text{even}] \]

6. Without looking at the definitions from the standard prelude, define the higher-order function \( \text{curry} \) that converts a function on pairs into a curried function, and conversely, the higher-order function \( \text{uncurry} \) that converts a curried function with two arguments into a function on pairs.

   Hint: first write down the types of the two functions.

7. A higher-order function \( \text{unfold} \) that encapsulates a simple pattern of recursion for producing a list can be defined as follows:

   \[
   \text{unfold} \; p \; h \; t \; x \mid p\; x \begin{array}{c}
   = \ [] \\
   \text{otherwise} \ = \ h\; x : \text{unfold} \; p \; h \; t \; (t\; x)
   \end{array}
   \]

   That is, the function \( \text{unfold} \; p \; h \; t \) produces the empty list if the predicate \( p \) is true of the argument, and otherwise produces a non-empty list by applying the function \( h \) to give the head, and the function \( t \) to generate another argument that is recursively processed in the same way to produce the tail of the list. For example, the function \( \text{int2bin} \) can be rewritten more compactly using \( \text{unfold} \) as follows:

   \[
   \text{int2bin} \ = \ \text{unfold} \ (==\ 0) \ (\text{\textquoteleft mod\textquoteleft} 2) \ (\text{\textquoteleft div\textquoteleft} 2)
   \]

   Redefine the functions \( \text{chop8} \), \( \text{map} \; f \) and \( \text{iterate} \; f \) using \( \text{unfold} \).

8. Modify the string transmitter program to detect simple transmission errors using parity bits. That is, each eight-bit binary number produced during encoding is extended with a parity bit, set to one if the number contains an odd number of ones, and to zero otherwise. In turn, each resulting nine-bit binary number consumed during decoding is checked to ensure that its parity bit is correct, with the parity bit being discarded if this is the case, and a parity error reported otherwise.

   Hint: the library function \( \text{error :: String} \rightarrow a \) terminates evaluation and displays the given string as an error message.

9. Test your new string transmitter program from the previous exercise using a faulty communication channel that forgets the first bit, which can be modelled using the \( \text{tail} \) function on lists of bits.
Chapter 8

Functional Parsers

In this chapter we show how Haskell can be used to program simple parsers. We start by explaining what parsers are and why they are useful, show how parsers can naturally be viewed as functions, define a number of basic parsers and higher-order functions for building larger parsers by combining smaller parsers, and conclude by developing a parser for arithmetic expressions.

8.1 Parsers

A parser is a program that takes a string of characters, and produces some form of tree that makes the syntactic structure of the string explicit. For example, given the string \(2 \times 3 + 4\), a parser for arithmetic expressions might produce a tree of the following form, in which the numbers appear at the leaves of the tree, and the operators appear at the branches:

```
    +
   / \  \
  *   4
 / \   \
2   3
```

The structure of this tree makes explicit that \(+\) and \(\times\) are operators with two arguments, and that \(\times\) has higher priority than \(+\).

Parsers are an important topic in computing, because most real-life programs use a parser to pre-process their input. For example, a calculator program parses numeric expressions prior to evaluating them, the Hugs system parses Haskell programs prior to executing them, and a web browser parses hypertext documents prior to displaying them. In each case, making the structure of the input explicit considerably simplifies its further processing. For example, once a numeric expression has been parsed into a tree structure such as in the example above, evaluating the expression is straightforward.
8.2 The parser type

In Haskell, a parser can naturally be viewed directly as a function that takes
a string and produces a tree. Hence, given a suitable type Tree of trees, the
notion of a parser can be represented as a function of type String → Tree,
which we abbreviate as Parser using the following definition:

\[ \text{type} \quad \text{Parser} \quad = \quad \text{String} \rightarrow \text{Tree} \]

In general, however, a parser might not always consume its entire argument
string. For example, a parser for numbers might be applied to a string com-
prising a number followed by a word. For this reason, we generalise our type
for parsers to also return any unconsumed part of the argument string:

\[ \text{type} \quad \text{Parser} \quad = \quad \text{String} \rightarrow (\text{Tree}, \text{String}) \]

Similarly, a parser might not always succeed. For example, a parser for num-
bers might be applied to a string comprising a word. To handle this, we further
generalise our type for parsers to return a list of results, with the convention
that the empty list denotes failure, and a singleton list denotes success:

\[ \text{type} \quad \text{Parser} \quad = \quad \text{String} \rightarrow [(\text{Tree}, \text{String})] \]

Returning a list also opens up the possibility of returning more than one result
if the argument string can be parsed in more than one way. For simplicity,
however, we only consider parsers that return at most one result.

Finally, different parsers will likely return different kinds of trees, or more
generally, any kind of value. For example, a parser for numbers might return
an integer value. Hence, it is useful to abstract from the specific type Tree of
result values, and make this into a parameter of the Parser type:

\[ \text{type} \quad \text{Parser} \quad a \quad = \quad \text{String} \rightarrow [(a, \text{String})] \]

In summary, this definition states that a parser of type \( a \) is a function that
takes an input string and produces a list of results, each of which is a pair
comprising a result value of type \( a \) and an output string. Alternatively, the
parser type can also be read as a rhyme in the style of Dr Seuss!

\[
\begin{align*}
A \, \text{parser} \, \text{for} \, \text{things} \\
\text{Is a function from} \, \text{strings} \\
\text{To lists of pairs} \\
\text{Of things and strings}
\end{align*}
\]

8.3 Basic parsers

We now define three basic parsers that will be used as the building blocks for
all other parsers. First of all, the parser \( \text{return} \, v \) always succeeds with the
result value \( v \), without consuming any of the input string:

\[
\begin{align*}
\text{return} & \quad :: \quad a \rightarrow \text{Parser} \ a \\
\text{return} \ v & \quad = \quad \lambda \text{inp} \rightarrow [(v, \text{inp})]
\end{align*}
\]

This function could equally well be defined by \( \text{result} \ v \ \text{inp} = [(v, \text{inp})] \). However, we prefer the above definition in which the second argument \( \text{inp} \) is slotted to the body of the definition using a lambda expression, because it makes explicit that \( \text{return} \) is a function that takes a single argument and returns a parser, as expressed by the type \( a \rightarrow \text{Parser} \ a \).

Whereas \( \text{return} \ v \) always succeeds, the dual parser \( \text{failure} \) always fails, regardless of the contents of the input string:

\[
\begin{align*}
\text{failure} & \quad :: \quad \text{Parser} \ a \\
\text{failure} & \quad = \quad \lambda \text{inp} \rightarrow []
\end{align*}
\]

Our final basic parser is \( \text{item} \), which fails if the input string is empty, and succeeds with the first character as the result value otherwise:

\[
\begin{align*}
\text{item} & \quad :: \quad \text{Parser} \ \text{Char} \\
\text{item} & \quad = \quad \lambda \text{inp} \rightarrow \text{case} \ \text{inp} \ \text{of} \\
& \quad \quad \quad \quad \quad \quad \quad [\ ] \rightarrow [] \\
& \quad \quad \quad \quad \quad \quad \quad (x : xs) \rightarrow [(x, xs)]
\end{align*}
\]

The \text{case} mechanism of Haskell used in this definition allows pattern matching to be used in the body of a definition, in this example by matching the string \( \text{inp} \) against two patterns to choose between two possible results. The \text{case} mechanism is not used much in this book, but can sometimes be useful.

Because parsers are functions, they could be applied to a string using normal function application, but we prefer to abstract from the representation of parsers by defining our own application function:

\[
\begin{align*}
\text{parse} & \quad :: \quad \text{Parser} \ a \rightarrow \text{String} \rightarrow [(a, \text{String})] \\
\text{parse} \ p \ \text{inp} & \quad = \quad p \ \text{inp}
\end{align*}
\]

Using \( \text{parse} \), we conclude this section with some examples that illustrate the behaviour of the three basic parsers defined above:

\[
\begin{align*}
> \ \text{parse} \ (\text{return} \ 1) \ "\text{hello}"
& \quad [1,"\text{hello}"] \\
> \ \text{parse} \ \text{failure} \ "\text{hello}"
& \quad [] \\
> \ \text{parse} \ \text{item} \ "\" \\
& \quad [] \\
> \ \text{parse} \ \text{item} \ "\text{hello}"
& \quad ["h","\text{ello}""]
\end{align*}
\]

Note that for technical reasons, the second example actually produces an error, but this does not occur when \( \text{failure} \) is used in non-trivial examples.
8.4 Sequencing

Perhaps the simplest way of combining two parsers is to apply one after the other in sequence, with the output string returned by the first parser becoming the input string to the second. But how should the result values from the two parsers be handled? One approach would be to combine the two values as a pair, using a sequencing operator for parsers with the following type:

\[ \text{Parser } a \to \text{Parser } b \to \text{Parser } (a, b) \]

In practice, however, it turns out to be more convenient to integrate the sequencing of parsers with the processing of their result values, by means of a sequencing operator \(\gg\gg\) (read as “then”) defined as follows:

\[
\begin{align*}
(\gg\gg) &:: \text{Parser } a \to (a \to \text{Parser } b) \to \text{Parser } b \\
\operatorname{p} \gg\gg \operatorname{f} &\equiv \lambda \text{inp} . \text{case } \text{parse } \operatorname{p} \ \text{inp} \ \text{of} \\
& \quad | [] \to [] \\
& \quad | [(\text{v}, \text{out})] \to \text{parse } (f \ \text{v}) \ \text{out}
\end{align*}
\]

That is, the parser \(\operatorname{p} \gg\gg \operatorname{f}\) fails if the application of the parser \(\operatorname{p}\) to the input string fails, and otherwise applies the function \(\operatorname{f}\) to the result value to give a second parser, which is then applied to the output string to give the final result. In this manner, the result value produced by the first parser is made directly available for processing by the second.

A typical parser built using \(\gg\gg\) has the following structure:

\[
\begin{align*}
p1 &\gg\gg \lambda \text{v1} \to \\
\operatorname{p2} &\gg\gg \lambda \text{v2} \to \\
& \vdots \\
\operatorname{pn} &\gg\gg \lambda \text{vn} \to \\
\text{return} &\ (f \ \text{v1} \ \text{v2} \ \cdots \ \text{vn})
\end{align*}
\]

That is, apply the parser \(\operatorname{p1}\) and call its result value \(\text{v1}\); then apply the parser \(\operatorname{p2}\) and call its result value \(\text{v2}\); \(\vdots\); then apply the parser \(\operatorname{pn}\) and call its result value \(\text{vn}\); and finally, combine all the results into a single value by applying the function \(f\). Haskell provides a special syntax for such parsers, allowing them to be expressed in the following, more appealing, form:

\[
\begin{align*}
\text{do } \text{v1} &\leftarrow \operatorname{p1} \\
\text{v2} &\leftarrow \operatorname{p2} \\
& \vdots \\
\text{vn} &\leftarrow \operatorname{pn} \\
\text{return} &\ (f \ \text{v1} \ \text{v2} \ \cdots \ \text{vn})
\end{align*}
\]

As with list comprehensions, the expressions \(\text{vi} \leftarrow \operatorname{pi}\) are called generators. If the result value produced by a generator \(\text{vi} \leftarrow \operatorname{pi}\) is not required, the generator can be abbreviated simply by \(\operatorname{pi}\). Note also that the layout rule applies to
the do notation for sequencing parsers, in the sense that each parser in the
sequence must begin in precisely the same column.

For example, a parser that consumes three characters, discards the second,
and returns the first and third as a pair can now be defined as follows:

\[
p :: Parser (Char, Char)
p = do x ← item
     y ← item
     return (x, y)
\]

Note that \( p \) only succeeds if every parser in its defining sequence succeeds,
which requires at least three characters in the input string:

\[
> \text{parse} \ p \ "abcdef"
(('a', 'c'), "def")
\]

\[
> \text{parse} \ p \ "ab"
[]
\]

### 8.5 Choice

Another natural way of combining two parsers is to apply the first parser to
the input string, and if this fails then apply the second instead. Such a choice
operator \( \text{+++} \) (read as “or else”) can be defined as follows:

\[
(\text{+++}) :: Parser a → Parser a → Parser a
p +++ q = \lambda \text{inp} \to \text{case} \ \text{parse} \ p \ \text{inp} \ \text{of}
\quad \ [] \to \text{parse} \ q \ \text{inp}
\quad [(v, out)] \to [(v, out)]
\]

For example:

\[
> \text{parse} \ (\text{item} +++ \text{return 'a'}) "hello"
[('h', "ello")]
\]

\[
> \text{parse} \ (\text{failure} +++ \text{return 'a'}) "hello"
[('a', "hello")]
\]

\[
> \text{parse} \ (\text{failure} +++ \text{failure}) "hello"
[]
\]

### 8.6 Derived primitives

Using the three basic parsers together with sequencing and choice, we can now
define a number of other useful parsing primitives. First of all, we define a
parser \( sat \) \( p \) for single characters that satisfy the predicate \( p \), where a predicate (or property) is a function that returns a logical value:

\[
sat \quad :: \quad (\text{Char} \rightarrow \text{Bool}) \rightarrow \text{Parser Char}
\]

\[
sat \ p \ = \ \text{do} \ x \leftarrow \text{item}
\quad \text{if} \ p \ x \ \text{then} \ \text{return} \ x \ \text{else} \ \text{failure}
\]

Using \( sat \) and appropriate predicates from the standard prelude, we can define parsers for single digits, lower-case letters, upper-case letters, arbitrary letters, alphanumeric characters, and specific characters:

\[
digit \quad :: \quad \text{Parser Char}
\]

\[
digit \ = \ sat \ isDigit
\]

\[
lower \quad :: \quad \text{Parser Char}
\]

\[
lower \ = \ sat \ isLower
\]

\[
upper \quad :: \quad \text{Parser Char}
\]

\[
upper \ = \ sat \ isUpper
\]

\[
letter \quad :: \quad \text{Parser Char}
\]

\[
letter \ = \ sat \ isAlpha
\]

\[
alphanum \quad :: \quad \text{Parser Char}
\]

\[
alphanum \ = \ sat \ isAlphaNum
\]

\[
char \quad :: \quad \text{Char} \rightarrow \text{Parser Char}
\]

\[
char \ x \ = \ sat \ (== \ x)
\]

For example:

\[
> \ \text{parse} \ \text{digit} \ "123"
\ = \ [(\text{"1"}, \text{"23"})]
\]

\[
> \ \text{parse} \ \text{digit} \ "abc"
\ = \ []
\]

\[
> \ \text{parse} \ (\text{char} \ \text{\textquoteleft a\textquoteright}) \ "abc"
\ = \ [(\text{\textquoteleft a\textquoteright}, \text{"bc"})]
\]

\[
> \ \text{parse} \ (\text{char} \ \text{\textquoteleft a\textquoteright}) \ "123"
\ = \ []
\]

In turn, using \( char \) we can define a parser \( string \) \( xs \) for the string of characters \( xs \), with the string itself returned as the result value:

\[
string \quad :: \quad \text{String} \rightarrow \text{Parser String}
\]

\[
string \ [\] \ = \ \text{return} \ [\]
\]

\[
string \ (x : xs) \ = \ \text{do} \ char \ x
\quad \text{string} \ xs
\quad \text{return} \ (x : xs)
\]
Note that \textit{string} is defined using recursion, and only succeeds if the entire
target string is consumed. The base case states that the empty string can
always be parsed. The recursive case states that a non-empty string can be
parsed by parsing the first character, parsing the remaining characters, and
returning the entire string as the result value. For example:

\[
\begin{array}{l}
> \ \text{parse (string "abc") "abcdef"} \\
\quad (["abc","def"]) \\
> \ \text{parse (string "abc") "ab1234"} \\
\quad []
\end{array}
\]

Our next two parsers, \textit{many} \( p \) and \textit{many1} \( p \), apply a parser \( p \) as many
times as possible until it fails, with the result values from by each successful
application of \( p \) being combined as a list. The difference between these two
repetition primitives is that \textit{many} permits zero or more applications of \( p \),
whereas \textit{many1} requires at least one successful application:

\[
\begin{array}{l}
\text{many} \quad :: \quad \text{Parser} \ a \rightarrow \text{Parser} \ [a] \\
\text{many} \ p \quad = \quad \text{many1} \ p \ \text{+++ return} \ [] \\
\text{many1} \quad :: \quad \text{Parser} \ a \rightarrow \text{Parser} \ [a] \\
\text{many1} \ p \quad = \quad \text{do} \ \ v \leftarrow \ p \\
\quad \hspace{2em} \quad \text{vs} \leftarrow \ \text{many} \ p \\
\quad \hspace{2em} \quad \text{return} \ (v : \ vs)
\end{array}
\]

Note that \textit{many} and \textit{many1} are defined using mutual recursion, as introduced
in section 6.5. In particular, the definition for \textit{many} \( p \) states that \( p \) can either
be applied at least once or not at all, while the definition for \textit{many1} \( p \) states
that \( p \) can be applied once and then zero or more times. For example:

\[
\begin{array}{l}
> \ \text{parse (many digit) "123abc"} \\
\quad (["123","abc"]) \\
> \ \text{parse (many digit) "abcdef"} \\
\quad (["","abcdef"]) \\
> \ \text{parse (many1 digit) "abcdef"} \\
\quad []
\end{array}
\]

Using \textit{many} and \textit{many1} we can define parsers for \textit{identifiers} (or names) compri-
sing a lower-case letter followed by zero or more alphanumeric characters,
natural numbers comprising one or more digits, and spacing comprising zero
or more spaces, tabs, and newline characters:

\[
\begin{align*}
\text{ident} & :: \(Parser\ \text{String} \\
\text{ident} & = \text{do } x \leftarrow \text{lower} \> \\
& \quad \text{xs} \leftarrow \text{many alphanumeric} \> \\
& \quad \text{return } (x : \text{xs}) \\
\text{nat} & :: \(Parser\ \text{Int} \\
\text{nat} & = \text{do } xs \leftarrow \text{many digit} \> \\
& \quad \text{return } (\text{read } \text{xs}) \\
\text{space} & :: \(Parser\ ()) \> \\
\text{space} & = \text{do } \text{many } (\text{sat } \text{isSpace}) \> \\
& \quad \text{return } ()
\end{align*}
\]

For example:

\[
\begin{align*}
> \text{parse ident "hello\_there"} \\
& (["hello","\_there"]) \\
> \text{parse nat "123\_pounds"} \\
& ([123,"\_pounds"]) \\
> \text{parse space "\_\_\_hello"} \\
& ([(),"hello"]) \\
\end{align*}
\]

Note that \text{space} returns the empty tuple () as a dummy result value, reflecting the fact that the details of spacing are not usually important.

### 8.7 Handling spacing

Most real-life parsers allow spacing to be freely used around the basic tokens in their input string. For example, the strings 1+2 and 1 + 2 are both parsed in the same way by Hugs. To handle such spacing, we define a new primitive that ignores any space before and after applying a parser for a token:

\[
\begin{align*}
\text{token} & :: \Parser\ a \rightarrow \Parser\ a \\
\text{token } p & = \text{do } \text{space} \> \\
& \quad \text{v} \leftarrow p \> \\
& \quad \text{space} \> \\
& \quad \text{return } v
\end{align*}
\]

Using \text{token}, it is now easy to define parsers that ignore spacing around identifiers, natural numbers, and special symbols:

\[
\begin{align*}
\text{identifier} & :: \Parser\ \text{String} \\
\text{identifier} & = \text{token } \text{ident} \\
\text{natural} & :: \Parser\ \text{Int} \\
\text{natural} & = \text{token } \text{nat} \\
\text{symbol} & :: \text{String} \rightarrow \Parser\ \text{String} \\
\text{symbol } xs & = \text{token } (\text{string } xs)
\end{align*}
\]
For example, a parser for a non-empty list of natural numbers that ignores spacing around tokens can be defined as follows:

\[
\begin{align*}
    p &::= \text{Parser [Int]} \\
    p &= \text{do symbol "["} \\
    &\quad \text{n} \leftarrow \text{natural} \\
    &\quad \text{ns} \leftarrow \text{many (do symbol ",")} \\
    &\quad \text{symbol "]"} \\
    &\quad \text{return (n : ns)}
\end{align*}
\]

This definition states that such a list begins with an opening square bracket and a natural number, followed by zero or more commas and natural numbers, and concludes with a closing square bracket. Note that \( p \) only succeeds if a complete list in precisely this format is consumed:

\[
\begin{align*}
    > \text{parse } p \ "[1,2,3]\]"
    > \text{parse } p \ "[1,2,]\"
\end{align*}
\]

8.8 Arithmetic expressions

We conclude this chapter with an extended example. Consider a simple form of arithmetic expressions built up from natural numbers using addition, multiplication and parentheses. We assume that addition and multiplication associate to the right, and that multiplication has higher priority than addition. For example, \( 2 + 3 + 4 \) means \( 2 + (3 + 4) \), while \( 2 \times 3 + 4 \) means \( (2 \times 3) + 4 \).

The syntactic structure of a language can be formalised using the mathematical notion of a grammar, which is a set of rules that describes how strings of the language can be constructed. For example, a grammar for our language of arithmetic expressions can be defined by the following two rules:

\[
\begin{align*}
    \text{expr} &::= \text{expr} + \text{expr} \mid \text{expr} \times \text{expr} \mid (\text{expr}) \mid \text{nat} \\
    \text{nat} &::= 0 \mid 1 \mid 2 \mid \cdots
\end{align*}
\]

The first rule states that an expression is either the addition or multiplication of two expressions, a parenthesised expression, or a natural number. In turn, the second rule states that a natural number is either zero, one, two, etc. For example, using this grammar the construction of the expression \( 2 \times 3 + 4 \) can be represented by the following parse tree, in which the tokens in the expression appear at the leaves, and the grammatical rules applied to construct
the expression give rise to the branching structure:

```
expr
 /\  
 expr + expr
 /\   /\       
 expr * expr nat
 /\   /\  /\    
 nat   nat nat 4
 /\   /\ 
 2   3 
```

The structure of this tree makes explicit that \(2 \times 3 + 4\) can be constructed from the addition of two expressions, the first given by the multiplication of two further expressions which are in turn given by the numbers two and three, and the second expression given by the number four. However, the grammar also permits another possible parse tree for this example, which corresponds to the erroneous interpretation of the expression as \(2 \times (3 + 4)\):

```
expr
 /\  
 expr * expr
 /\   /\       
 nat   expr + expr
 /\   /\  /\    
 2   nat   nat
 /\   /\ 
 3   4 
```

The problem is that our grammar for expressions does not take account of the fact that multiplication has higher priority than addition. However, this can easily be fixed by modifying the grammar to have a separate rule for each level of priority, with addition at the lowest level of priority, multiplication at the middle level, and parentheses and numbers at the highest level:

```
expr ::= expr + expr | term
term ::= term * term | factor
factor ::= (expr) | nat
nat ::= 0 | 1 | 2 | ... 
```

Using this new grammar, \(2 \times 3 + 4\) indeed has a single parse tree, which cor-
responds to the correct interpretation of the expression as $(2 \times 3) + 4$:

\[
\begin{array}{c}
expr \\
\downarrow \\
expr + expr \\
\downarrow \\
term \\
\downarrow \\
term * term \\
\downarrow \\
factor \\
\downarrow \\
nat \\
\downarrow \\
2 \\
\downarrow \\
3 \\
\end{array}
\]

We have now dealt with the issue of priority, but our grammar does not yet take account of the fact that addition and multiplication associate to the right. For example, the expression $2 + 3 + 4$ currently has two possible parse trees, corresponding to $(2 + 3) + 4$ and $2 + (3 + 4)$. However, this can also easily be fixed by modifying the grammatical rules for addition and multiplication to be recursive in their right argument only, rather than both arguments:

\[
\begin{align*}
expr &::= \term + expr \mid \term \\
\term &::= \factor * \term \mid \factor
\end{align*}
\]

Using these new rules, $2 + 3 + 4$ now has a single parse tree, which corresponds to the correct interpretation of the expression as $2 + (3 + 4)$:

\[
\begin{array}{c}
expr \\
\downarrow \\
\term + expr \\
\downarrow \\
factor + term + expr \\
\downarrow \\
nat + factor + term \\
\downarrow \\
2 + nat + factor \\
\downarrow \\
3 + nat + factor \\
\downarrow \\
2 + 3 + nat + factor \\
\downarrow \\
2 + 3 + 4
\end{array}
\]

In fact, our grammar for expressions is now *unambiguous*, in the sense that every well-formed expression has precisely one parse tree.

Our final modification to the grammar is one of simplification. For example, consider the rule $expr ::= \term + expr \mid \term$, which states that an
expression is either the addition of a term and an expression, or a term. In other words, an expression always begins with a term, which can then be followed by the addition of an expression or by nothing. Hence, the rule for expressions can be simplified to \( \text{expr} ::= \text{term} (+ \text{expr} | \epsilon) \), in which the symbol \( \epsilon \) denotes the empty string. Simplifying the rule for terms in a similar manner gives our final grammar for arithmetic expressions:

\[
\begin{align*}
\text{expr} & ::= \text{term} (+ \text{expr} | \epsilon) \\
\text{term} & ::= \text{factor} (* \text{term} | \epsilon) \\
\text{factor} & ::= (\text{expr}) | \text{nat} \\
\text{nat} & ::= 0 | 1 | 2 | \cdots
\end{align*}
\]

It is now straightforward to translate this grammar into a parser for expressions, by simply rewriting the rules using our parsing primitives. In fact, we choose to have the parser itself evaluate the expression being parsed to its integer value, rather than returning some form of tree:

\[
\begin{align*}
\text{expr} & :: \text{Parser Int} \\
\text{expr} &= \text{do } t \leftarrow \text{term} \\
& \quad \text{do symbol } "*" \\
& \quad \quad e \leftarrow \text{expr} \\
& \quad \quad \text{return} (t + e) \\
& \quad \quad +++ \text{return} t \\
\text{term} & :: \text{Parser Int} \\
\text{term} &= \text{do } f \leftarrow \text{factor} \\
& \quad \text{do symbol } "*" \\
& \quad \quad t \leftarrow \text{term} \\
& \quad \quad \text{return} (f \times t) \\
& \quad \quad +++ \text{return} f \\
\text{factor} & :: \text{Parser Int} \\
\text{factor} &= \text{do symbol } "("  \\
& \quad e \leftarrow \text{expr} \\
& \quad \text{symbol } ")" \\
& \quad \text{return} e \\
& \quad +++ \text{ natural}
\end{align*}
\]

For example, the parser \( \text{expr} \) first parses a term with value \( t \), then parses a plus symbol followed by an expression with value \( e \) and returns the value \( t + e \), or else parses nothing further and simply returns the value \( t \). The parsers \( \text{term} \) and \( \text{factor} \) can be read in a similar manner.

Finally, using \( \text{expr} \) we define a function \( \text{eval} :: \text{String} \rightarrow \text{Int} \) that evaluates an arithmetic expression to its integer value. To handle the cases of unconsumed and invalid input, we use the library function \( \text{error} :: \text{String} \rightarrow \text{a} \) that
displays an error message and then terminates the program:

\[
\begin{align*}
\text{eval} & \quad :: \quad \text{String} \rightarrow \text{Int} \\
\text{eval} \ x s & = \quad \textbf{case} \ (\text{parse expr} \ x s) \ \textbf{of} \\
& \quad \quad [\!\![n,\!\!]\!] \rightarrow n \\
& \quad \quad [\!\![\_ \ out]\!\!] \rightarrow \text{error} \ ("\text{unconsumed input}" \ \# \ \text{out}) \\
& \quad \quad [] \rightarrow \text{error} \ "\text{invalid input}"
\end{align*}
\]

For example:

\[
\begin{align*}
> \ \text{eval} \ "2*3+4" \\
& 10
\end{align*}
\]

\[
\begin{align*}
> \ \text{eval} \ "2*(3+4)" \\
& 14
\end{align*}
\]

\[
\begin{align*}
> \ \text{eval} \ "2_\_\_\_*(3_\_\_\_+4)" \\
& 14
\end{align*}
\]

\[
\begin{align*}
> \ \text{eval} \ "2*3-4" \\
& \text{Error} : \ \text{unconsumed input} - 4
\end{align*}
\]

\[
\begin{align*}
> \ \text{eval} \ "-1" \\
& \text{Error} : \ \text{invalid input}
\end{align*}
\]

### 8.9 Chapter remarks

A library file comprising the parsing primitives from this chapter is available on the web [9]. For technical reasons concerning the connection between parsers, the do notation, and the mathematical notion of a monad, a number of the basic definitions in this library are slightly different to those given here. This chapter is based upon [12, 13], which explores these and other issues in further detail. More information concerning grammars can be found in [20], and more advanced approaches to building parsers in Haskell are given in [16, 5]. The reading of the parser type as a rhyme is due to Fritz Ruehr.

### 8.10 Exercises

1. The library file also defines a parser \texttt{int} :: \textit{Parser Int} for an integer. Without looking at this definition, define \texttt{int}. Hint: an integer is either a minus symbol followed by a natural number, or a natural number.

2. Define a parser \texttt{comment} :: \textit{Parser} () for ordinary Haskell comments that begin with the symbol -- and extend to the end of the current line, which is represented by the control character ' \n'.

105
3. Using our second grammar for arithmetic expressions, draw the two possible parse trees for the expression \(2 + 3 + 4\).

4. Using our third grammar for arithmetic expressions, draw the parse trees for the expressions \(2 + 3\), \(2 \times 3 \times 4\) and \((2 + 3) + 4\).

5. Extend the parser for arithmetic expressions to support subtraction and division, based upon the following extensions to the grammar:

\[
\begin{align*}
expr &::= \ term \ (+ \ expr \ | \ - \ expr \ | \ \epsilon) \\
\term &::= \ factor \ (* \ term \ | \ / \ term \ | \ \epsilon)
\end{align*}
\]

6. Further extend the grammar and parser for arithmetic expressions to support exponentiation, which is assumed to associate to the right and have higher priority than multiplication and division, but lower priority than parentheses and numbers. For example, \(2 \uparrow 3 \times 4\) means \((2 \uparrow 3) \times 4\). Hint: the new level of priority requires a new rule in the grammar.

7. Consider expressions built up from natural numbers using a subtraction operator that is assumed to associate to the left.

(a) Define a grammar for such expressions.

(b) Translate this grammar into a parser \(expr :: Parser Int\).

(c) What is the problem with this parser?

(d) Show how it can be fixed. Hint: rewrite the parser using the repetition primitive \(many\) and the library function \(foldl\).
Chapter 9

Interactive Programs

DRAFT of February 19, 2005

In this chapter we show how Haskell can be used to write interactive programs. We start by explaining what interactive programs are, show how such programs can naturally be viewed as functions, define a number of basic interactive programs and higher-order functions for combining interactive programs, and conclude by developing a desktop calculator and the game of life.

9.1 Interaction

A batch program is one that takes all its input at the start, processes this input in some way, and then produces all its output at the end. In the early days of computing, most programs were batch programs, run in isolation from their users in order to maximise the amount of time that the computer was performing useful work. For example, a payroll program for a company may take details of all the employees as input, calculate the pay for each employee, and then produce payslips for all the employees as output.

Up to this point in the book we have considered how Haskell can be used to write batch programs. In Haskell such programs, and more generally all programs, are modelled as pure functions that take all their input as explicit arguments, and produce all their output as explicit results. For example, a payroll program may be modelled as a function of type [Employee] → [Payslip] that takes a list of employee details and produces a list of payslips.

In contrast, an interactive program is one that may take additional input from the user, and produce additional output for the user, while the program is running. In the modern era of computing, most programs are interactive programs, run as a dialogue with their users in order to provide increased flexibility. For example, a calculator program may allow the user to enter numeric expressions interactively using the keyboard, and immediately display the value of such expressions on the screen.

At first sight, modelling interactive programs as pure functions may seem infeasible, because such programs by their very nature require the side effects
of taking additional input and producing additional output while the program is running. For example, what is an appropriate type for the calculator program described above as a pure function from arguments to results?

Over the years there have been many proposed solutions to the problem of combining the notion of pure functions with that of side effects. In the remainder of this chapter we present the solution that is adopted in Haskell, which is based upon the use of a new type in conjunction with a small number of primitives. As we shall see, this approach shares much in common with the approach to parsers presented in the previous chapter.

9.2 The input/output type

In Haskell, an interactive program is viewed as a pure function that takes the current “state of the world” as its argument, and produces a modified world as its result, in which the modified world makes explicit any side effects performed by the program. Hence, given a suitable type World whose values represent the current state of the world, the notion of an interactive program can be represented by a function of type World → World, which we abbreviate as IO (short for “Input/Output”) as follows:

\[ \text{type } IO = \text{World} \to \text{World} \]

In general, however, an interactive program may return a result value in addition to performing side effects. For example, a program for reading a character from the keyboard may return the character that was read. For this reason, we generalise our input/output type to also return a result value, with the type of such values being a parameter of the input/output type:

\[ \text{type } IO a = \text{World} \to (a, \text{World}) \]

Expressions of type \( IO a \) for some \( a \) are called actions. For example, \( IO \) \( Char \) is the type of actions that return a character, while \( IO () \) is the type of actions that return the empty tuple () as a dummy result value. Actions of the latter type can be thought of as purely side effecting actions that return no result value, and are frequently used when writing interactive programs.

In addition to returning a result value, interactive programs may also take an argument value. However, there is no need to generalise the type of actions to take account of this, because this behaviour can already be achieved by exploiting currying. For example, an interactive program that takes a character and returns an integer would have type \( Char \to IO \) \( Int \), which abbreviates the curried function type \( Char \to \text{World} \to (\text{Int}, \text{World}) \).

At this point the reader may be concerned about the feasibility of passing around the entire state of the world when programming with actions. In reality, Haskell systems such as Hugs and the Glasgow Haskell Compiler implement actions in a more efficient manner than described above, but for the purposes of understanding, this conceptual view of actions will suffice.


9.3 Basic actions

We now introduce three basic actions from which all other actions will be constructed. First of all, the action getChar reads a character from the keyboard, echos it to the screen, and returns the character as its result value:

\[
\begin{align*}
\text{getChar} & : \text{IO Char} \\
\text{getChar} & = \cdots
\end{align*}
\]

The actual definition for getChar is built-in to the Haskell system, and cannot be defined within Haskell itself. If there are no characters waiting to be read from the keyboard, getChar waits until one is typed.

The dual action putChar c writes the character c to the screen, and returns no result value, represented by the empty tuple:

\[
\begin{align*}
\text{putChar} & : \text{Char} \rightarrow \text{IO } () \\
\text{putChar} & = \cdots
\end{align*}
\]

Our final basic action is return v, which simply returns the result value v without performing any interaction:

\[
\begin{align*}
\text{return} & : a \rightarrow \text{IO } a \\
\text{return } v & = \lambda\text{world} \rightarrow (v, \text{world})
\end{align*}
\]

Evaluating an action using Hugs performs its side effects, and discards the result value produced by the action. For example, evaluating getChar waits until a character is typed, displays it on the screen, and then terminates.

Note that the function return::a -> IO a provides a bridge from the world of pure expressions without side effects to that of impure actions with side effects, but there is no bridge back. Once we are impure we are impure for ever, and there is no possibility for redemption! As a result, one may suspect that impurity quickly permeates entire programs, but in practice this is usually not the case. For most Haskell programs, the vast majority of functions do not involve interaction, with this being handled by a relatively small number of interactive functions at the outermost level.

9.4 Sequencing

As with parsers, the natural way of combining two actions is to perform one after the other in sequence, with the modified world returned by the first action becoming the current world for the second, by means of a sequencing operator \(\gg\) (read as “then”) defined as follows:

\[
\begin{align*}
(\gg) & : \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b \\
f \gg g & = \lambda\text{world} \rightarrow \text{case } f \text{ world of } \\
& \quad (v, \text{world}') \rightarrow g v \text{ world}'
\end{align*}
\]
That is, we apply the action $f$ to the current world, then apply the function $g$ to the result value to give a second action, which is then applied to the modified world to give the final result. In practice, however, the do notation for sequencing parsers can also be used to sequence actions.

For example, using sequencing the primitive $getChar$ can be decomposed into its three component parts of reading a character from the keyboard, echoing it to the screen, and returning the character as the result:

$$
getChar :: IO Char
getChar = do x <- getCh
           putChar x
           return x
$$

The action $getCh$ that reads a character without echoing is not part of standard Haskell, but is provided as an extension by Hugs and can be made available in any script by including the special line $primitive getCh :: IO Char$.

### 9.5 Derived primitives

Using the three basic actions together with sequencing, we can now define a number of other useful action primitives. First of all, we define an action $getLine$ that reads a string of characters from the keyboard:

$$
getLine :: IO String
getLine = do x <- getChar
            if x == '\n' then return [] else do xs <- getLine
                                          return (x : xs)
$$

(The symbol ‘\n’ represents the newline character.) Dually, we define actions $putStr$ and $putStrLnLn$ that display a string on the screen, with the latter action also moving onto a new line afterwards:

$$
putStr :: String \rightarrow IO ()
putStr [] = return ()
putStr (x : xs) = do putChar x
                    putStr xs
putStrLnLn :: String \rightarrow IO ()
putStrLnLn xs = do putStr xs
                   putChar '\n'
$$

For example, using these primitive we can now define an action that prompts for a string to be entered from the keyboard, and then displays its length:

$$
strlen :: IO ()
strlen = do putStr "Enter a string: ", xs <- getLine
            putStrLnLn $ "The string has " ++ show (length xs)
            putStrLnLn $ "characters"
$$
For example:

\[
\text{\texttt{\textgreater{} ssstrlen}} \nn \text{\texttt{\textit{Enter a string: hello}}}} \nn \text{\texttt{\textit{The string has 5 characters}}}
\]

In addition to the library primitives defined above, it is also useful to define a number of other primitives. First of all, we define actions that sound a beep and clear the screen, by displaying the appropriate control characters:

\[
\begin{align*}
\text{\texttt{beep}} & \quad : \quad \text{\texttt{IO \{\}}}
\text{\texttt{beep}} & \quad = \quad \text{\texttt{putStr "\BEL"}}
\text{\texttt{cls}} & \quad : \quad \text{\texttt{IO \{\}}}
\text{\texttt{cls}} & \quad = \quad \text{\texttt{putStr \"\ESC[2J\"}}
\end{align*}
\]

The screen on which characters are displayed can be viewed as an \((x, y)\) coordinate system, in which \(x\) and \(y\) represent steps right and down, with position \((1, 1)\) being the top-left of the screen. The notion of such a position can be represented by the following type:

\[
\text{\texttt{type Pos = (Int, Int)}}
\]

Using the appropriate control characters, we can now define a function that moves the cursor to a given position, where the cursor is a marker on the screen that indicates where the next character displayed will appear:

\[
\begin{align*}
\text{\texttt{goto}} & \quad : \quad \text{\texttt{Pos \rightarrow IO \{\}}}
\text{\texttt{goto \,(x, y) \quad = \quad putStr \text{\texttt{\"\ESC\[n + \\text{\texttt{show \,y + \";\, + \text{\texttt{show \,x + \"H\"}}}}}}}}
\end{align*}
\]

In turn, we define a function that displays a string at a given position:

\[
\begin{align*}
\text{\texttt{writenat}} & \quad : \quad \text{\texttt{Pos \rightarrow String \rightarrow IO \{\}}}
\text{\texttt{writenat \,p \,xs \quad = \quad do \,goph \,p \n\text{\texttt{putStr \,xs}}}}
\end{align*}
\]

Finally, we define a function \texttt{seqn} that performs a list of actions in sequence, discarding their result values and returning no result:

\[
\begin{align*}
\text{\texttt{seqn \quad : \quad [IO\,a] \rightarrow IO \{\}}
\text{\texttt{seqn \,[\,] \quad = \quad return \,()}}
\text{\texttt{seqn \,(a \,:\, as) \quad = \quad do \,a \n\text{\texttt{seqn \,as}}}}
\end{align*}
\]

For example, using \texttt{seqn} and a list comprehension, the above definition for \texttt{putStr} can be rewritten more compactly as follows:

\[
\text{\texttt{putStr \,xs \quad = \quad seqn \,[\text{\texttt{putChar \,x \,| \,x \,\leftarrow \,xs}}]}}
\]
9.6 Desktop calculator

At the end of the previous chapter we developed a parser for arithmetic expressions. We now extend this example to produce a simple desktop calculator, which allows the user to enter arithmetic expressions interactively using the keyboard, and displays the value of such expressions on the screen.

We will assume that arithmetic expressions are built up from integers using addition, subtraction, multiplication, division and parentheses. As our concern in this chapter is interactive programs, we do not consider the details of the parser for expressions, and assume that we are given a suitable parser \textit{expr :: Parser Int} that parses and evaluates such expressions, as can be obtained by solving some of the exercises from the previous chapter.

We begin by considering the user interface of the calculator. First of all, we define the box of the calculator as a list of strings:

\[
\begin{align*}
\text{box} &:: [\text{String}] \\
\text{box} &= ["+------------------+", \\
&\quad "|------------------|", \\
&\quad "|uqucucudlulm|", \\
&\quad "|ululul3ululm|", \\
&\quad "|ululul5ululm|", \\
&\quad "|ululul7ululm|", \\
&\quad "|ululul9ululm|", \\
&\quad "|ululul1ululm|", \\
&\quad "+------------------+
\]
\]

The first four buttons on the calculator, q, c, d, and =, allow the user to quit, clear the display, delete a character, and evaluate an expression, while the remaining sixteen buttons allow the user to enter arithmetic expressions.

We also define the buttons on the calculator as a list of characters, comprising both the twenty standard buttons that appear on the box itself, together with a number of extra characters that will be supported for flexibility, namely Q, C, D, space, escape, backspace, delete and newline:

\[
\begin{align*}
\text{buttons} &:: [\text{Char}] \\
\text{buttons} &= \text{standard ++ extra} \\
\text{where}
\end{align*}
\]

\[
\begin{align*}
\text{standard} &= "qcd=123+456-789*0()/" \\
\text{extra} &= "QCD\backslash ESC\backslash BS\backslash DEL\n"
\end{align*}
\]

Using a list comprehension, we can define an action that displays the calculator box in the top left-hand corner of the screen:

\[
\begin{align*}
\text{showbox} &:: \text{IO ()} \\
\text{showbox} &= \text{seqn [writeat (1,y) xs | (y,xs) <- zip [1..13] box]}
\end{align*}
\]
The last part of the user interface is to define a function that shows a string in the display of the calculator. We first clear the display, which ensures that if the user deletes any characters, they will automatically be removed from the display. To avoid limiting the size of the expressions that can be entered, we only show at most the last thirteen characters of the given string. In this manner, if the user types more than thirteen characters, the display will appear to scroll to the left as additional characters are typed.

\[
\begin{align*}
\text{display} & \quad \:: \quad \text{String} \rightarrow \text{IO} () \\
\text{display} \; xs & \quad = \quad \text{do} \; \text{writeat} \; (3, 2) \; "\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\n\end{align*}
\]

The calculator itself is controlled by a function \textit{calc} that displays the current string, and then reads a character from the keyboard without echoing it. If this character is a valid button then it is processed, otherwise we sound a beep to indicate an error and continue with the same string:

\[
\begin{align*}
\text{calc} & \quad :: \quad \text{String} \rightarrow \text{IO} () \\
\text{calc} \; xs & \quad = \quad \text{do} \; \text{display} \; xs \\
& \qquad \quad \quad c \leftarrow \text{getCh} \\
& \quad \quad \quad \text{if} \; \text{elem} \; c \; \text{buttons} \; \text{then} \; \text{process} \; c \; xs \; \text{else} \; \text{do} \; \text{beep} \\
& \qquad \quad \quad \quad \quad \quad \quad \text{calc} \; xs
\end{align*}
\]

The function \textit{process} takes a valid character and the current string, and performs the appropriate action depending upon the character:

\[
\begin{align*}
\text{process} & \quad :: \quad \text{Char} \rightarrow \text{String} \rightarrow \text{IO} () \\
\text{process} \; c \; xs \\
& \quad \mid \; \text{elem} \; c \; "\text{q}\text{Q}\text{ESC}" \quad = \quad \text{quit} \\
& \quad \mid \; \text{elem} \; c \; "\text{d}\text{D}\text{BS}\text{DEL}" \quad = \quad \text{delete} \; xs \\
& \quad \mid \; \text{elem} \; c \; "\text{=}\text{n}" \quad = \quad \text{eval} \; xs \\
& \quad \mid \; \text{elem} \; c \; "\text{c}\text{C}" \quad = \quad \text{clear} \\
& \quad \mid \; \text{otherwise} \quad = \quad \text{press} \; c \; xs
\end{align*}
\]

We now consider each of the five possible actions in turn.

- Quitting moves the cursor below the box and terminates:

\[
\begin{align*}
\text{quit} & \quad :: \quad \text{IO} () \\
\text{quit} & \quad = \quad \text{goto} \; (1, 14)
\end{align*}
\]

- Deleting has no effect if the current string is empty, and otherwise removes the last character from this string:

\[
\begin{align*}
\text{delete} & \quad :: \quad \text{String} \rightarrow \text{IO} () \\
\text{delete} \; "" & \quad = \quad \text{calc} \; "" \\
\text{delete} \; xs & \quad = \quad \text{calc} \; (\text{init} \; xs)
\end{align*}
\]
• Evaluation displays the result of parsing the current string if successful, and otherwise sounds a beep and retains the original string:

\[
\begin{align*}
\text{eval} & \quad :: \quad \text{String} \rightarrow \text{IO} () \\
\text{eval} \ \text{xs} & = \quad \text{case} \ \text{parse} \ \text{expr} \ \text{xs} \quad \text{of} \\
& \quad \quad \quad [(n, "")] \rightarrow \text{calc} (\text{show} n) \\
& \quad \quad \quad \_ \rightarrow \text{do} \ \text{beep} \\
& \quad \quad \quad \quad \text{calc} \ \text{xs}
\end{align*}
\]

• Clearing the display resets the current string to empty:

\[
\begin{align*}
\text{clear} & \quad :: \quad \text{IO} () \\
\text{clear} & = \quad \text{calc} ""
\end{align*}
\]

• Any other character is appended to the end of the current string:

\[
\begin{align*}
\text{press} & \quad :: \quad \text{Char} \rightarrow \text{String} \rightarrow \text{IO} () \\
\text{press} \ \text{c} \ \text{xs} & = \quad \text{calc} (\text{xs} + [\text{c}])
\end{align*}
\]

Finally, we define a function that runs the calculator, by clearing the screen, displaying the box, and starting with an empty display:

\[
\begin{align*}
\text{run} & \quad :: \quad \text{IO} () \\
\text{run} & = \quad \text{do} \ \text{cls} \\
& \quad \quad \text{showbox} \\
& \quad \quad \text{clear}
\end{align*}
\]

### 9.7 Game of Life

The game of life is a simple example of an evolutionary system, played on a two-dimensional board. Each square on the board is either empty, or contains a single living cell, as shown in the following example:

```
```

Each internal square on the board has eight immediate neighbours:
We arrange for squares on the edge of the board to also have eight neighbours, by viewing the board as wrapping around from top-to-bottom and from left-to-right. In this manner, we can think of the board as really being a torus, a three-dimensional doughnut shaped object.

Given an initial configuration of the board, the next generation is given by simultaneously applying the following rules to each square:

- A living cell survives if it has precisely two or three neighbouring squares that contain living cells, and dies otherwise;
- An empty square gives birth to a living cell if it has precisely three neighbours that contain living cells, and remains empty otherwise.

For example, applying these rules to the above board gives:

```
        0  1  2  3  4
 0   0 0 0 0  0
 1   0 0 0 0  0
 2   0 0 0 0  0
 3   0 0 0 0  0
 4   0 0 0 0  0
```

By repeating this procedure with the new board, an infinite sequence of generations can be produced from an initial configuration. By careful design of the initial board, all sorts of interesting behaviours can be observed in the sequence of generations. For example, the pattern above is called a “glider”, and over successive generations will move diagonally down the board.

Despite its simplicity, the game of life is in fact “computationally complete”, in the sense that any computational process can be simulated within it by a suitable encoding. In the remainder of this section we show how the game of life can be implemented in Haskell.

To allow the size of the board to be easily modified, we define two integer values that determine the width and height of the board in squares:

```
width  ::  Int
width  =  5

height ::  Int
height =  5
```

We represent a board as a list of the \((x, y)\) positions at which there is a living cell, using the same coordinate convention as the screen:

```
  type Board  =  [Pos]
```

For example, the initial example board above would be represented by:

```
  glider  ::  Board
  glider =  [(4, 2), (2, 3), (4, 3), (3, 4), (4, 4)]
```
Using this representation it is easy to display the cells on the screen, and to decide for a board if a given position is alive or empty:

\[
\begin{align*}
\text{showcells} &:: \text{Board} \rightarrow \text{IO} \left(\right) \\
\text{showcells } b &::= \text{seqn } \left\{ \text{writeat } (x, y) \text{ "0" | } (x, y) \leftarrow b \right\} \\
\text{isAlive} &:: \text{Board} \rightarrow \text{Pos} \rightarrow \text{Bool} \\
\text{isAlive } b \ p &::= \text{elem } p \ b \\
\text{isEmpty} &:: \text{Board} \rightarrow \text{Pos} \rightarrow \text{Bool} \\
\text{isEmpty } b \ p &::= \neg \left(\text{isAlive } b \ p\right)
\end{align*}
\]

Next we define a function that returns the list of neighbours of a position:

\[
\begin{align*}
\text{neighbs} &:: \text{Pos} \rightarrow \left[\text{Pos}\right] \\
\text{neighbs } (x, y) &::= \text{map } \text{wrap } \left\{ (x - 1, y - 1), (x, y - 1), (x + 1, y - 1), (x - 1, y), (x + 1, y), (x - 1, y + 1), (x, y + 1), (x + 1, y + 1) \right\}
\end{align*}
\]

The function \text{wrap} used in this definition takes account of the wrapping around at the edges of the board, by subtracting one from each component of the given position, taking the remainder when divided by the width and height of the board, and adding one to each component again:

\[
\begin{align*}
\text{wrap} &:: \text{Pos} \rightarrow \text{Pos} \\
\text{wrap } (x, y) &::= \left\langle\left(\left((x - 1) \text{ mod } \text{width}\right) + 1, \left((y - 1) \text{ mod } \text{height}\right) + 1\right)\right\rangle
\end{align*}
\]

We can now define a function that calculates the number of live neighbours on a board for a given position by producing the list of neighbours, retaining those that are alive, and counting their number:

\[
\begin{align*}
\text{liveneighbs} &:: \text{Board} \rightarrow \text{Pos} \rightarrow \text{Int} \\
\text{liveneighbs } b &::= \text{length } \circ \text{filter } \left(\text{isAlive } b\right) \circ \text{neighbs}
\end{align*}
\]

Using this function, it is now straightforward to produce the list of living positions in a board that have precisely two or three living neighbours, and hence survive to the next generation:

\[
\begin{align*}
\text{survivors} &:: \text{Board} \rightarrow \left[\text{Pos}\right] \\
\text{survivors } b &::= \left[ p \mid p \leftarrow b, \text{elem } \left(\text{liveneighbs } b \ p\right) \left[2, 3\right]\right]
\end{align*}
\]

In turn, the list of empty positions in a board that have precisely three living neighbours, and hence give birth to a new cell, can be produced as follows:

\[
\begin{align*}
\text{births} &:: \text{Board} \rightarrow \left[\text{Pos}\right] \\
\text{births } b &::= \left[ (x, y) \mid x \leftarrow \left[1..\text{width}\right], \ y \leftarrow \left[1..\text{height}\right], \ \text{isEmpty } b \ (x, y), \ \text{liveneighbs } b \ (x, y) == 3\right]
\end{align*}
\]
However, this definition considers every possible position on the board. A more refined approach, which may be more efficient for larger boards, is to only consider the neighbours of living cells on the board, because only such positions can potentially give rise to new births. Using this approach, the function \textit{births} can be rewritten as follows:

\[
\begin{align*}
\text{births } b &= \{ p \mid p \leftarrow \text{rndups} (\text{concat} (\text{map} \text{neighs} b)), \\
&\quad \text{isEmpty} \ b \ p, \\
&\quad \text{liveneighs} \ b \ p == 3 \}
\end{align*}
\]

The function \textit{rndups} removes duplicate elements from a list, and is used above to ensure that each potential new cell is only considered once:

\[
\begin{align*}
\text{rndups} &:: \ E q \ a \Rightarrow [a] \rightarrow [a] \\
\text{rndups} \ [] &= [] \\
\text{rndups} \ (x : xs) &= x : \text{rndups} \ (\text{filter} \ (\not\equiv x) \ xs)
\end{align*}
\]

The next generation of a board can now be produced simply by appending the list of survivors and the list of new births:

\[
\begin{align*}
\text{nextgen} &:: \ \text{Board} \rightarrow \text{Board} \\
\text{nextgen} \ b &= \ \text{survivors} \ b \uplus \text{births} \ b
\end{align*}
\]

Finally, we define a function \textit{life} that implements the game of life itself, by clearing the screen, showing the living cells in the current board, waiting for a moment, and then continuing with the next generation:

\[
\begin{align*}
\text{life} &:: \ \text{Board} \rightarrow \text{IO} () \\
\text{life} \ b &= \ \text{do} \ \text{cls} \\
&\quad \text{showcells} \ b \\
&\quad \text{wait} \ 5000 \\
&\quad \text{life} \ (\text{nextgen} \ b)
\end{align*}
\]

The function \textit{wait} is used to slow down the game to a reasonable speed, and can be implemented by performing a given number of trivial actions:

\[
\begin{align*}
\text{wait} &:: \ \text{Int} \rightarrow \text{IO} () \\
\text{wait} \ n &= \ \text{seqn} \ [\text{return} () | i \leftarrow [1 \ldots n]]
\end{align*}
\]

For fun, you may like to try out the \textit{life} function with the \textit{glider} example, and experiment with some patterns of your own.

### 9.8 Chapter remarks

Reading and writing of files can also be performed using the \texttt{input/output} type, as discussed in further detail in the Haskell Report [18]. A formal meaning for \texttt{input/output} and various other side-effects is given in \textit{Tackling the Awkward Squad} [?]. A variety of libraries for performing graphical interaction are available from the Haskell home page, \texttt{www.haskell.org}. The game of life was invented by John Conway, and popularised by an article in the October 1970 edition of Scientific American written by Martin Gardner.
9.9 Exercises

1. Define an action `readLine :: IO String` that behaves in the same way as `getLine`, except that it also permits the delete key to be used to remove characters. Hint: the delete character is `\DEL`, and the control string for moving the cursor back one character is `\ESC[1D`.

2. Modify the calculator program to indicate the approximate position of an error rather than just sounding a beep, by using the fact that the parser returns the unconsumed part of the input string.

3. One some systems the game of life may flicker, due to the entire screen being cleared each generation. Modify the game to avoid such flicker by only displaying and clearing positions whose status changes.

4. Produce an editor that allows the user to interactively create and modify the content of the board in the game of life.

5. Produce graphical versions of the calculator and game of life programs, using one of the graphics libraries available from `www.haskell.org`.

6. Nim is a game that is played on a board comprising five numbered rows of stars, which is initially set up as follows:

   1 : * * * * *
   2 : * * * *
   3 : * * *
   4 : * *
   5 : *

   Two players take it in turn to remove one or more stars from the end of a single row. The winner is the player who removes the last star or stars from the board. Implement the game of nim in Haskell. Hint: represent the board as a list comprising the number of stars remaining on each row, with the initial board being `[5, 4, 3, 2, 1]`. 
Chapter 10

Defining Types and Classes

In preparation.
Chapter 11

The Countdown Problem

In this chapter we show how Haskell can be used to solve the countdown problem, a numbers game in which the aim is to construct numeric expressions satisfying certain constraints. We start by formalising the rules of the problem in Haskell, and then present a simple but inefficient program that solves the problem, whose efficiency is then improved in two stages.

11.1 Introduction

Countdown is a popular quiz programme that has been running on British television since 1982, and includes a numbers game that we shall refer to as the *countdown problem*. The essence of the problem is as follows:

Given a sequence of numbers and a target number, attempt to construct an expression whose value is the target, by combining one or more numbers from the sequence using addition, subtraction, multiplication, division and parentheses.

Each number in the sequence can only be used at most once in the expression, and all of the numbers involved, including intermediate values, must be integers greater than zero. In particular, the use of negative numbers, zero, and proper fractions such as $2 \div 3$, is not permitted.

For example, suppose that we are given the sequence 1, 3, 7, 10, 25, 50 and the target 765. Then one possible solution is given by the expression $(1 + 50) \times (25 - 10)$, as shown by the following simple calculation:

\[
(1 + 50) \times (25 - 10) \\
= \{ \text{applying } + \} \\
51 \times (25 - 10) \\
= \{ \text{applying } - \} \\
51 \times 15 \\
= \{ \text{applying } \ast \} \\
765
\]
In fact, for this example it can be shown that there are 780 different solutions! On the other hand, keeping the same sequence but changing the target to 831 gives an example that can be shown to have no solutions.

In the television version of the countdown problem, a number of additional rules are adopted to make the problem suitable for human players on a quiz programme. In particular, there are always six numbers selected from the sequence 1..10, 1..10, 25, 50, 75, 100, the target is always in the range 100..999, and there is a time limit of 30 seconds. It is natural to abstract from such constraints when developing computer players, so none of the programs that we develop in this chapter enforces or depends upon these extra rules. Note, however, that we do not abstract from the integers greater than zero to a richer numeric domain such as the integers or rationals, as this would fundamentally change the computational complexity of the problem.

### 11.2 Formalising the problem

We start by defining a type $Op$ of numeric operators, together with a function $valid$ that decides if the application of an operator to two integers that are greater than zero gives an integer greater than zero, and a function $apply$ that actually performs such a valid application:

```haskell
data Op = Add | Sub | Mul | Div

deriving Show

valid :: Op -> Int -> Int -> Bool
valid Add x y = True
valid Sub x y = x > y
valid Mul x y = (x `mod` y) == 0
apply :: Op -> Int -> Int -> Int
apply Add x y = x + y
apply Sub x y = x - y
apply Mul x y = x * y
apply Div x y = x `div` y
```

For example, $Sub 2 3$ is invalid because $2 - 3$ is negative, while $Div 2 3$ is invalid because $2 \div 3$ is rational. Note the use of the derived instance in the definition for $Op$, which ensures that values of this type can be converted into strings and hence displayed by Haskell system.

We now define a type $Expr$ of numeric expressions, which can either be an integer value or the application of an operator to two expressions, together with a function $values$ that returns the list of values in an expression, and a function $eval$ that returns the overall value of an expression, provided that
this value is an integer that is greater than zero:

\[
\text{data \textit{Expr} = Val \textit{Int} \mid \textit{App} \textit{Op} \textit{Expr} \textit{Expr}}
\]

\[
\text{deriving Show}
\]

\[
\begin{align*}
\text{values} & \quad :: \quad \textit{Expr} \rightarrow \textit{[Int]} \\
\text{values (Val } n \text{)} & \quad = \quad [n] \\
\text{values (App } l \text{ } r \text{)} & \quad = \quad \text{values } l \oplus \text{values } r \\
\text{eval} & \quad :: \quad \textit{Expr} \rightarrow \textit{[Int]} \\
\text{eval (Val } n \text{)} & \quad = \quad [n \mid n > 0] \\
\text{eval (App } o \text{ } l \text{ } r \text{)} & \quad = \quad [\text{apply } o \ x \ y \mid x \gets \text{eval } l, y \gets \text{eval } r, \text{valid } o \ x \ y]
\end{align*}
\]

Note that the possibility of failure within \textit{eval} is handled by returning a list of results, with the convention that a singleton list denotes success, and the empty list denotes failure. For example, for \(2 + 3\) and \(2 - 3\) we have:

\[
\begin{align*}
\text{> eval (App Add (Val 2) (Val 3))} \\
\text{[5]}
\end{align*}
\]

\[
\begin{align*}
\text{> eval (App Sub (Val 2) (Val 3))} \\
\text{[]}
\end{align*}
\]

Failure within \textit{eval} could also be handled by using the \textit{Maybe} type, but we prefer to use the list type in this case because the comprehension notation then provides a convenient way to define the \textit{eval} function.

We now define a number of useful \textit{combinatorial} functions that return all possible lists satisfying certain properties. The function \textit{subs} returns all subsequences of a list, which are given by all possible combinations of excluding or including each element, \textit{interleave} returns all possible ways of inserting a new element into a list, and \textit{perms} returns all permutations of a list, which are given by all possible reorderings of the elements:

\[
\begin{align*}
\text{subs} & \quad :: \quad [a] \rightarrow [\{a\}] \\
\text{subs \textit{[]}} & \quad = \quad [\textit{[]}] \\
\text{subs \textit{(x : xs)}} & \quad = \quad \textit{ysss} \oplus \text{map} \ (\textit{x} ::) \ \textit{ysss} \\
\text{where} & \quad \textit{ysss} = \textit{subs \ xxs} \\
\text{interleave} & \quad :: \quad a \rightarrow [a] \rightarrow [\{a\}] \\
\text{interleave \textit{x \textit{[]}}} & \quad = \quad [\textit{x}] \\
\text{interleave \textit{x \textit{(y : ys)}}} & \quad = \quad (\textit{x : y : ys}) : \text{map} \ (\textit{y} ::) \ (\text{interleave \textit{x \ ys})} \\
\text{perms} & \quad :: \quad [a] \rightarrow [\{a\}] \\
\text{perms \textit{[]}} & \quad = \quad [\textit{[]}] \\
\text{perms \textit{(x : xs)}} & \quad = \quad \text{concat} \ (\text{map} \ (\text{interleave \textit{x})} \ (\text{perms \ xxs}))
\end{align*}
\]

For example:

\[
\begin{align*}
\text{> subs \textit{[1, 2, 3]}} \\
\text{[[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]}
\end{align*}
\]
> \textit{interleave} 1 [2,3,4] \\
[[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1]]

> \textit{perms} [1,2,3] \\
[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]

In turn, a function that returns all \textit{subbags} of a list, which are given by all possible ways of selecting zero or more elements in any order, can be defined simply by considering all permutations of all subsequences:

\[
\text{subbags} :: \ [a] \to [[a]] \\
\text{subbags} \ \text{xs} \ = \ \text{concat} \ (\text{map} \ \text{perms} \ (\text{sub} \ \text{xs}))
\]

For example:

> \text{subbags} [1,2,3] \\
[[],[3],[2],[2,3],[3,2],[1],[1,3],[3,1],[1,2],[2,1],
[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1]]

Finally, we can now define a function \textit{solution} that formalises what it means to solve an instance of the countdown problem:

\[
\text{solution} :: \ \text{Expr} \to \text{Int} \to \text{Int} \to \text{Bool} \\
\text{solution} \ e \ \text{ns} \ n \ = \ \text{elem} \ (\text{values} \ e) \ (\text{subbags} \ \text{ns}) \ \land \ \text{eval} \ e \ == \ [n]
\]

That is, an expression is a solution for a given list of numbers and a target if the list of values in the expression is a subbag of the list of numbers, and the expression successfully evaluates to give the target. For example, if \( e :: \text{Expr} \) corresponds to the expression \((1 + 50) \ast (25 - 10)\), then we have:

> \text{solution} e [1,3,7,10,25,50] 765 \\
\text{True}

Note that the efficiency of \textit{solution} could be improved by using a function \textit{subbag} that decides if one list is a subbag of another directly, rather than doing so indirectly using the function \textit{subbags} that returns all possible subbags of a list. However, efficiency is not important at this stage, and \textit{subbags} itself is used to define a number of other functions in this chapter.

### 11.3 Brute force solution

Our first approach to solving the countdown problem is by brute force, using the idea of generating all possible expressions over the given list of numbers. We start by defining a function \textit{split} that returns all possible ways of splitting a list into two non-empty lists that append to give the original list:

\[
\text{split} :: \ [a] \to [[[a],[a]]] \\
\text{split} [] = [] \\
\text{split} [\_] = [] \\
\text{split} \ (x:xs) = ([x],xs) : ([x:ls,rs] \mid (ls,rs) \leftarrow \text{split} \ xs)
\]

124
For example:

\[
> \text{split} \ [1, 2, 3, 4] \\
\quad = \{(1, [2, 3, 4]), ([1, 2], [3, 4]), ([1, 2, 3], [4])\}
\]

Using \text{split} we can now define the key function \text{exprs}, which returns all possible expressions whose list of values is precisely a given list:

\[
\text{exprs} :: [\text{Int}] \to [\text{Expr}] \\
\text{exprs} \ [\] = [\text{Val} \ n] \\
\text{exprs} \ n = [\text{Val} \ n] \\
\text{exprs} \ ns = [e | (ls, rs) \leftarrow \text{split} \ ns, \\
\quad l \leftarrow \text{exprs} \ ls, \\
\quad r \leftarrow \text{exprs} \ rs, \\
\quad e \leftarrow \text{combine} \ l \ r]
\]

That is, for the empty list of numbers there are no possible expressions, while for a single number there is a single expression comprising that number. Otherwise, for a list of two or more numbers we first produce all splittings of the list, then recursively calculate all possible expressions for each of these lists, and finally combine each pair of expressions using each of the four numeric operators, using an auxiliary function defined as follows:

\[
\text{combine} :: \text{Expr} \to \text{Expr} \to [\text{Expr}] \\
\text{combine} \ l \ r = [\text{App} \ o \ l \ r | o \leftarrow \text{ops}] \\
\text{ops} :: [\text{Op}] \\
\text{ops} = [\text{Add}, \text{Sub}, \text{Mul}, \text{Div}]
\]

Finally, we can now define a function \text{solutions} that returns all possible expressions that solve an instance of the countdown problem, by first generating all expressions over each subbag of the given list of numbers, and then selecting those expressions that successfully evaluate to give the target:

\[
\text{solutions} :: [\text{Int}] \to [\text{Expr}] \\
\text{solutions} \ ns \ n = [e | ns' \leftarrow \text{subbags} \ ns, \\
\quad e \leftarrow \text{exprs} \ ns', \\
\quad e \text{eval} e == [n]]
\]

For the purposes of testing our programs in this chapter, the performance of Hugs is somewhat limited, so instead we use the Glasgow Haskell Compiler (GHC). For example, using GHC (version 5.04.1) on a 1.5GHz Pentium 4 processor, \text{solutions} \ [1, 3, 7, 10, 25, 50] 765 returns the first solution in 0.62 seconds, and all 780 solutions in 74.08 seconds, while if the target is changed to 831 then the empty list of solutions is returned in 69.52 seconds.

125
11.4 Combining generation and evaluation

The function solutions generates all possible expressions over the given numbers, but in practice many of these expressions will fail to evaluate, due to the fact that subtraction and division are not always valid for integers greater than zero. For example, it can be shown that there are 33,665,406 possible expressions over the numbers 1, 3, 7, 10, 25, 50, but only 4,672,540 of these expressions evaluate successfully, which is just under 14%.

Based upon this observation, our second approach to solving the countdown problem is to improve our brute force program by combining the generation of expressions with their evaluation, such that both tasks are performed simultaneously. In this way, expressions that fail to evaluate are rejected at an earlier stage, and more importantly, are not used to generate further such expressions. We start by defining a type Result of expressions that evaluate successfully paired with their overall values:

\[
\text{type Result} = (\text{Expr},\text{Int})
\]

Using this type, we now define a function results that returns all possible results comprising expressions whose list of values is precisely a given list:

\[
\begin{align*}
\text{results} & :: [\text{Int}] \rightarrow [\text{Result}] \\
\text{results} [\ ] & = [ ] \\
\text{results} [n] & = [(\text{Val} n, n) \mid n > 0] \\
\text{results} ns & = [\text{res} \mid (ls, rs) \leftarrow \text{split} ns, \\
& \quad lx \leftarrow \text{results} ls, \\
& \quad ry \leftarrow \text{results} rs, \\
& \quad \text{res} \leftarrow \text{combine'} lx \; ry]
\end{align*}
\]

That is, for the empty list there are no possible results, while for a single number there is a single result formed from that number, provided that the number itself is greater than zero. Otherwise, for two or more numbers we first produce all splittings of the list, then recursively calculate all possible results for each of these lists, and finally combine each pair of results using each of the four numeric operators that are valid:

\[
\begin{align*}
\text{combine'} & :: \text{Result} \rightarrow \text{Result} \rightarrow [\text{Result}] \\
\text{combine'} (l, x) (r, y) & = [(\text{App} o l r, \text{apply} o x y) \mid o \leftarrow \text{ops}, \text{valid} o x y]
\end{align*}
\]

Using results we can now define a new function solutions' that returns all possible expressions that solve an instance of the countdown problem, by first generating all results over each subbag of the given numbers, and then selecting those expressions whose value is the target:

\[
\begin{align*}
\text{solutions'} & :: [\text{Int}] \rightarrow \text{Int} \rightarrow [\text{Expr}] \\
\text{solutions'} ns n & = [e \mid ns' \leftarrow \text{subbags} ns, \\
& \quad (e, m) \leftarrow \text{results} ns', \\
& \quad m == n]
\end{align*}
\]

126
In terms of performance, solutions’ [1, 3, 7, 10, 25, 50] 765 returns the first solution in 0.06 seconds (10 times faster than solutions) and all solutions in 7.52 seconds (almost 10 times faster), while if the target is changed to 831 the empty list is returned in 5.46 seconds (almost 13 times faster).

11.5 Exploiting algebraic properties

The function solutions’ generates all possible expressions over the given numbers whose evaluation is successful, but in practice many of these expressions will be essentially the same, due to the fact that the numeric operators have algebraic properties. For example, the expressions \(2 + 3\) and \(3 + 2\) are essentially the same because the result of an addition does not depend upon the order of the two arguments, while \(2 \div 1\) and \(2\) are essentially the same because dividing any number by one has no effect on that number.

Based upon this observation, our final approach to solving the countdown problem is to improve our second program by exploiting such properties to reduce the number of generated expressions. In particular, we exploit the following five commutativity and identity properties:

\[
\begin{align*}
x + y & = y + x \\
x \times y & = y \times x \\
x \times 1 & = x \\
1 \times y & = y \\
x \div 1 & = x
\end{align*}
\]

We start by recalling the function valid that decides if the application of an operator to two integers that are greater than zero gives another such:

\[
\begin{align*}
\text{valid} & \quad :: \quad \text{Op} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool} \\
\text{valid Add } \_\_ & = \text{True} \\
\text{valid Sub } x \ y & = x \succ y \\
\text{valid Mul } \_\_ & = \text{True} \\
\text{valid Div } x \ y & = x \mod y == 0
\end{align*}
\]

This definition can be modified to exploit the commutativity of addition and multiplication simply by requiring that their arguments are in numeric order \((x \leq y)\), and the identity properties of multiplication and division simply by requiring that the appropriate arguments are non-unitary (\(\neq 1\)):

\[
\begin{align*}
\text{valid Add } x \ y & = x \leq y \\
\text{valid Sub } x \ y & = x \succ y \\
\text{valid Mul } x \ y & = x \neq 1 \land y \neq 1 \land x \leq y \\
\text{valid Div } x \ y & = y \neq 1 \land x \mod y == 0
\end{align*}
\]

For example, using this new definition Add 3 2 is now invalid because it is essentially the same as Add 2 3 using the commutativity of addition, while
Div 2 1 is now invalid because it is essentially the same as the number 2 on its own using the identity property for division.

Using the new version of valid gives a new version of our function solutions\(’\) that solves the countdown problem, which we write as solutions\(”\). Using this new function can considerably reduce the number of generated expressions and the number of solutions. For example, solutions\(”\) \([1, 3, 7, 10, 25, 50]\) 765 only generates 245,644 expressions, of which just 49 are solutions, which is just over 5% and 6% respectively of the numbers using solutions\(’\).

As regards performance, solutions\(”\) \([1, 3, 7, 10, 25, 50]\) 765 now returns the first solution in 0.03 seconds (twice as fast as solutions\(’\)) and all solutions in 0.80 seconds (9 times faster), while for the target number 831 the empty list is returned in 0.66 seconds (8 times faster). More generally, given any numbers from the television version of the countdown problem, our final program solutions\(”\) typically returns all solutions in under one second.

11.6 Chapter remarks

Countdown is based upon an original version on French television called “Des Chiffres et des Lettres”, while the countdown problem itself is related to the childrens arithmetic games called “krypto” and “four fours”. This chapter is based upon [11], which also includes proofs of correctness of the three programs produced. A library file comprising the definitions from this chapter is available on the web [9]. A number of more advanced approaches to solving the countdown problem are explored by Bird and Mu [1]. The definitions for the functions subs, interleave and perms are due to Bird and Wadler [2].

11.7 Exercises

1. Redefine the combinatorial function subbags using a list comprehension rather than the library functions concat and map.

2. Define a recursive function subbag :: Eq a ⇒ [a] → [a] → Bool that decides if one list is a subbag of another, without using the combinatorial functions perms and subs. Hint: start by defining a function that removes the first occurrence of a value from a list.

3. What effect would generalising the function split to also return pairs containing the empty list have on the behaviour of solutions?\(’\)

4. Using subbags, exprs and eval, verify that there are 33,665,406 possible expressions over the numbers 1, 3, 7, 10, 25, 50, and that only 4,672,540 of these expressions evaluate successfully.

5. Similarly, verify that the number of expressions that evaluate successfully increases to 10,839,369 if the numeric domain is generalised to arbitrary integers. Hint: modify the definition of valid.
6. Modify the final program to:

(a) allow the use of exponentiation in expressions;
(b) produce the nearest solutions if no exact solution is possible;
(c) order the solutions using a suitable measure of simplicity.
Chapter 12

Lazy Evaluation

DRAFT of February 19, 2005

In this chapter we introduce lazy evaluation, the mechanism by which expressions in Haskell are evaluated. We start by reviewing the notion of evaluation, then consider evaluation strategies and their properties, discuss infinite data structures and their influence on modular programming, and conclude by showing how evaluation order can be controlled.

12.1 Introduction

As explained in chapter 1, the basic method of computation in Haskell is the application of functions to arguments, and expressions are evaluated by successively applying functions until this is no longer possible. For example, suppose that we define a function \( \text{inc} \) that increments an integer:

\[
\begin{align*}
\text{inc} & \quad : \ Int \to Int \\
\text{inc} \ n & \quad = \ n + 1
\end{align*}
\]

Then the expression \( \text{inc} \ (2 \ast 3) \) can be evaluated as follows:

\[
\begin{align*}
\text{inc} \ (2 \ast 3) & \quad = \quad \{ \text{applying } \ast \} \\
\text{inc} \ 6 & \quad = \quad \{ \text{applying } \text{inc} \} \\
6 + 1 & \quad = \quad \{ \text{applying } + \} \\
7 & \quad \\
\end{align*}
\]

However, this is not the only possible evaluation sequence. In particular, we could start by applying \( \text{inc} \), rather than by applying \( \ast \):

\[
\begin{align*}
\text{inc} \ (2 \ast 3) & \quad = \quad \{ \text{applying } \text{inc} \} \\
\end{align*}
\]
\[(2 \times 3) + 1 \]
\[= \{ \text{applying } \ast \} \]
\[6 + 1 \]
\[= \{ \text{applying } + \} \]
\[7 \]

The final value is the same in both cases. In fact, this behaviour is not specific to simple examples, but is an important general property of function application in Haskell: any two different ways of evaluating the same expression will always produce the same final value, provided that they both terminate. We will return to the issue of termination later on in this chapter.

Note that the above property is not true for evaluation in most imperative programming languages, in which the basic method of computation is changing stored values. For example, suppose that we have a variable \(n\) that stores a number that can be changed over time using the assignment symbol :=, and which is initialised to zero. Then the expression \(n + (n := 1)\) that adds together the current value of \(n\) and the result of an assignment to \(n\) can be evaluated either by first evaluating the left-hand argument of the addition

\[
n + (n := 1) \\
= \{ \text{applying } n \} \\
0 + (n := 1) \\
= \{ \text{applying } := \} \\
0 + 1 \\
= \{ \text{applying } + \} \\
1
\]

or by first evaluating the right-hand argument:

\[
n + (n := 1) \\
= \{ \text{applying } := \} \\
n + 1 \\
= \{ \text{applying } n \} \\
1 + 1 \\
= \{ \text{applying } + \} \\
2
\]

The final value is different in each case. The general problem illustrated by this example is that the precise time at which an assignment is performed in an imperative language may affect the value that results from a computation. In contrast, the time at which a function is applied to an argument in Haskell never affects the value that results from a computation. Nonetheless, as we shall see in the remainder of this chapter, there are important practical issues concerning the order and nature of evaluation.
12.2 Innermost evaluation

An expression which has the form of a function applied to one or more arguments that can be “reduced” by performing the application is called a reducible expression, or \textit{redex} for short. As indicated by the use of quotations marks in the preceding sentence, such reductions do not necessarily decrease the size of an expression, although in practice this is often the case.

For example, suppose that we define a function \textit{mult} that takes a pair of integers and returns their product as follows:

\[
\begin{align*}
mult & \quad : \quad (\text{Int, Int}) \rightarrow \text{Int} \\
mult (x, y) & \quad = \quad x \times y
\end{align*}
\]

Now consider the expression \textit{mult} \((1 + 2, 2 + 3)\). This expression contains three redexes, namely the subexpressions \(1 + 2\) and \(2 + 3\), which have the form of the function \(+\) applied to two arguments, and the entire expression \textit{mult} \((1 + 2, 2 + 3)\) itself, which has the form of the function \textit{mult} applied to a pair of arguments. Performing the corresponding reductions gives the expressions \textit{mult} \((3, 2 + 3)\), \textit{mult} \((1 + 2, 5)\), and \((1 + 2) \times (2 + 3)\).

When evaluating an expression, in what order should reductions be performed? One common strategy is to always choose a redex that is innermost, in the sense that it contains no other redex. If there is more than innermost redex, by convention we choose that which begins at the leftmost position in the expression. This evaluation strategy is called \textit{innermost} evaluation.

For example, both \(1 + 2\) and \(2 + 3\) contain no other redexes and are hence innermost within the expression \textit{mult} \((1 + 2, 2 + 3)\), with the redex \(1 + 2\) beginning at the leftmost position. More generally, our example expression is evaluated using innermost evaluation as follows:

\[
\begin{align*}
mult (1 + 2, 2 + 3) & = \{ \text{applying the first +} \} \\
mult (3, 2 + 3) & = \{ \text{applying +} \} \\
mult (3, 5) & = \{ \text{applying mult} \} \\
3 \times 5 & = \{ \text{applying *} \} \\
15 & \quad
\end{align*}
\]

Innermost evaluation can also be characterised in terms of how arguments are passed to functions. In particular, using this strategy ensures that the argument of a function is always fully evaluated before the function itself is applied. That is, arguments are passed \textit{by value}. For example, as shown above, evaluating \textit{mult} \((1 + 2, 2 + 3)\) using innermost evaluation proceeds by first evaluating the argument expressions \(1 + 2\) and \(2 + 3\), and then applying the function \textit{mult}. The fact that we always choose the leftmost innermost redex ensures that the first argument is evaluated before the second.
12.3 Outermost evaluation

Another common strategy for evaluating an expression, dual to innermost evaluation, is to always choose a redex that is outermost, in the sense that it is contained in no other redex. If there is more than one such redex then as previously we choose that which begins at the leftmost position. Not surprisingly, this evaluation strategy is called \textit{outermost} evaluation.

For example, the expression \textit{mult} \((1 + 2, 2 + 3)\) is contained in no other redex and is hence outermost within itself. More generally, evaluating this expression using outermost evaluation proceeds as follows:

\[
\begin{align*}
mult (1 + 2, 2 + 3) & = \{ \text{applying} \ mult \} \\
(1 + 2) * (2 + 3) & = \{ \text{applying the first} + \} \\
3 * (2 + 3) & = \{ \text{applying} + \} \\
3 * 5 & = \{ \text{applying} * \} \\
15 &
\end{align*}
\]

In terms of how arguments are passed to functions, using outermost evaluation allows functions to be applied before their arguments are evaluated. For this reason, we say that arguments are passed \textit{by name}. For example, as shown above, evaluating \textit{mult} \((1 + 2, 2 + 3)\) using outermost evaluation proceeds by first applying the function \textit{mult} to the two unevaluated arguments \(1 + 2\) and \(2 + 3\), and then evaluating these two expressions in turn.

Note that many built-in functions require one or more of their arguments to be evaluated before being applied, even when using outermost evaluation. For example, as illustrated in the calculation above, arithmetic operators such as \(*\) and \(+\) cannot be applied until their two arguments have been evaluated to numbers. Functions with this property are called \textit{strict}, and will be discussed in further detail at the end of this chapter.

12.4 Lambda expressions

Let us now define a curried version of \textit{mult} that takes its arguments one at a time, using a lambda expression to make the use of currying explicit:

\[
\begin{align*}
mult & :: Int \to Int \to Int \\
mult \ x & = \lambda y \to x * y
\end{align*}
\]

Then using innermost evaluation, for example, we have:

\[
\begin{align*}
mult (1 + 2) (2 + 3) & = \{ \text{applying the first} + \} \\
mult 3 (2 + 3) &
\end{align*}
\]
\[ \begin{align*}
&= \{ \text{applying } \text{mult} \} \\
&\quad (\lambda y \to 3 \ast y)(2 + 3) \\
&= \{ \text{applying } + \} \\
&\quad (\lambda y \to 3 \ast y) \ 5 \\
&= \{ \text{applying } \lambda y \to 3 \ast y \} \\
&\quad 3 \ast 5 \\
&= \{ \text{applying } \ast \} \\
&\quad 15
\end{align*} \]

That is, the results of evaluating the two arguments are now substituted into the body of the function \text{mult} one at a time as expected, rather than at the same time as previously. This behaviour arises because \text{mult} 3 is the leftmost innermost redex in the expression \text{mult} 3 (2 + 3), as opposed to \(2 + 3\) in the expression \text{mult} (3, 2 + 3). Performing a reduction on this redex in the second step of the calculation above gives the lambda expression \(\lambda y \to 3 \ast y\) that awaits the result of evaluating the second argument.

In functional programming, the selection of redexes \textit{within} lambda expressions is normally prohibited. That is, we do not “reduce under lambdas”. For example, the lambda expression \(\lambda x \to 1 + 2\) that represents the function that takes an argument \(x\) and returns the result \(1 + 2\) is deemed to already be fully evaluated, even though its body contains the redex \(1 + 2\).

The reason for this constraint is that in functional programming, functions are viewed as “black boxes” that we are not permitted to look inside. More formally, the only operation that can be performed on a function is that of applying it to an argument. As such, reduction within the body of a function is only permitted once the function has been applied.

Using leftmost innermost and outermost evaluation but not under lambdas is normally referred to as \textit{call-by-value} and \textit{call-by-name} evaluation, respectively. In the next two sections we explore how these two evaluation strategies compare in terms of two important properties, namely their termination behaviour and the number of reductions that they require.

### 12.5 Termination

Consider the following recursive definition:

\[ \begin{align*}
\text{inf} &:: \text{Int} \\
\text{inf} &= 1 + \text{inf}
\end{align*} \]

That is, the integer \(\text{inf}\) (representing “infinity”) is defined as the successor of itself. Evaluating \(\text{inf}\) produces a larger and larger expression, and hence does not terminate, regardless of the evaluation strategy used:

\[ \begin{align*}
&= \{ \text{applying } \text{inf} \} \\
&1 + \text{inf}
\end{align*} \]
\[
\begin{align*}
&= \{ \text{applying } \text{inf} \} \\
&= 1 + (1 + \text{inf}) \\
&= \{ \text{applying } \text{inf} \} \\
&= 1 + (1 + (1 + \text{inf})) \\
&= \{ \text{applying } \text{inf} \} \\
&= 1 + (1 + (1 + (1 + \text{inf}))) \\
&= \{ \text{applying } \text{inf} \} \\
&= \ldots
\end{align*}
\]

In practice, evaluating \text{inf} using Hugs will quickly exhaust the available memory and produce an error message. Now consider the expression \text{fst} (0, \text{inf}) that contains \text{inf}, where \text{fst} is the library function that selects the first component of a pair, defined by \text{fst} (x, y) = x. Using call-by-value evaluation with this expression also results in non-termination:

\[
\begin{align*}
\text{fst} (0, \text{inf}) \\
&= \{ \text{applying } \text{inf} \} \\
\text{fst} (0, 1 + \text{inf}) \\
&= \{ \text{applying } \text{inf} \} \\
\text{fst} (0, 1 + (1 + \text{inf})) \\
&= \{ \text{applying } \text{inf} \} \\
\text{fst} (0, 1 + (1 + (1 + \text{inf}))) \\
&= \{ \text{applying } \text{inf} \} \\
&= \ldots
\end{align*}
\]

In contrast, using call-by-name evaluation results in termination in just one step, by immediately applying the definition of \text{fst} and hence avoiding the evaluation of the non-terminating expression \text{inf}:

\[
\begin{align*}
\text{fst} (0, \text{inf}) \\
&= \{ \text{applying } \text{fst} \} \\
&= 0
\end{align*}
\]

This simple example shows that call-by-name evaluation may produce a result when call-by-value evaluation fails to terminate. More generally, we have the following important property: if there exists any evaluation sequence that terminates for a given expression, then call-by-name evaluation will also terminate for this expression, and produce the same final result.

In summary, call-by-name evaluation is preferable to call-by-value for the purposes of ensuring that evaluation terminates as often as possible.

### 12.6 Number of reductions

Consider the following definition:

\[
\begin{align*}
\text{square} &:: \text{Int} \rightarrow \text{Int} \\
\text{square} \ n &= n \times n
\end{align*}
\]

Then using call-by-value evaluation, we have:
\[\text{square} \ (1 + 2) = \begin{cases} \text{applying +} \end{cases} \]
\[
\text{square} \ 3 = \begin{cases} \text{applying square} \end{cases} \\
3 * 3 = \begin{cases} \text{applying *} \end{cases} \\
9
\]

In contrast, using call-by-name evaluation with the same expression requires one extra reduction step, due to the fact that \(1 + 2\) is duplicated when the function \(\text{square}\) is applied, and hence must be evaluated twice:

\[
\text{square} \ (1 + 2) = \begin{cases} \text{applying square} \end{cases} \\
(1 + 2) * (1 + 2) = \begin{cases} \text{applying the first +} \end{cases} \\
3 * (1 + 2) = \begin{cases} \text{applying +} \end{cases} \\
3 * 3 = \begin{cases} \text{applying *} \end{cases} \\
9
\]

This example shows that call-by-name evaluation may require more steps than call-by-value evaluation, in particular when an argument is used more than once in the body of a function. More generally, we have the following property: arguments are evaluated precisely once using call-by-value evaluation, but may be evaluated many times using call-by-name.

In summary, call-by-value evaluation is preferable to call-by-name for the purposes of ensuring that evaluation does not duplicate work.

Fortunately, the above efficiency problem with call-by-name evaluation can easily be solved by using “pointers” to indicate sharing of expressions during evaluation. Rather than physically copying an argument if it is used many times in the body of a function, we simply keep one copy of the argument and make many pointers to it. In this manner, any reductions that are performed on the argument are automatically shared between each of pointers to that argument. For example, using this strategy we have:

\[
\text{square} \ (1 + 2) = \begin{cases} \text{applying square} \end{cases} \\
\quad \begin{array}{c}
\quad * \\
\quad 1 + 2
\end{array} \\
= \begin{cases} \text{applying +} \end{cases} \\
\quad \begin{array}{c}
\quad * \\
\quad 3
\end{array} \\
= \begin{cases} \text{applying *} \end{cases}
\]

137
That is, when applying the definition \( \text{square } n = n \times n \) in the first step, we keep a single copy of the argument expression \( 1 + 2 \), and make two pointers to it. In this manner, when the expression \( 1 + 2 \) is reduced in the second step, both pointers in the expression share the result.

The use of call-by-name evaluation in conjunction with sharing is called lazy evaluation. This is the evaluation strategy that is used in Haskell, as a result of which Haskell is called a lazy functional programming language. Being based upon call-by-name evaluation, lazy evaluation has the property that it ensures that evaluation terminates as often as possible. Moreover, using sharing ensures that lazy evaluation never requires more steps than call-by-value evaluation. The use of the term “lazy” will be explained shortly.

### 12.7 Infinite structures

An additional property of call-by-name evaluation, and hence lazy evaluation, is that it allows what at first sight may seem impossible: programming with infinite structures. We have already seen a simple example of this idea earlier in this chapter, in the form of the evaluation of \( \text{fst } (0, \text{inf}) \) avoiding the production of the infinite structure \( 1 + (1 + (1 + \cdots)) \) defined by \( \text{inf} \).

Things become more interesting when we consider infinite lists. For example, consider the following recursive definition:

\[
\text{ones} :: [\text{Int}]
\]
\[
\text{ones} = 1 : \text{ones}
\]

That is, the list \( \text{ones} \) is defined as a single one followed by itself. As with \( \text{inf} \), evaluation of \( \text{ones} \) using any strategy does not terminate:

\[
\text{ones}
\]
\[= \{ \text{applying } \text{ones} \}
\]
\[= 1 : \text{ones}
\]
\[= \{ \text{applying } \text{ones} \}
\]
\[= 1 : (1 : \text{ones})
\]
\[= \{ \text{applying } \text{ones} \}
\]
\[= 1 : (1 : (1 : \text{ones}))
\]
\[= \{ \text{applying } \text{ones} \}
\]
\[= \]

In practice, evaluating \( \text{ones} \) using Hugs will produce a never ending list of ones, until we eventually decide to terminate this process:

\[
> \text{ones}
\]
\[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, \cdots]
\]
That is, \textit{ones} defines an infinite list of ones. Now consider the expression \textit{head ones}, which selects the first element of this list. Using call-by-value evaluation with this expression also results in non-termination, because the next step is always to apply the definition of \textit{ones}:

\begin{verbatim}
head ones
  =  \{ applying ones \}
head (1 : ones)
  =  \{ applying ones \}
head (1 : (1 : ones))
  =  \{ applying ones \}
head (1 : (1 : (1 : ones)))
  =  \{ applying ones \}
  ;
\end{verbatim}

In contrast, using lazy evaluation (or call-by-name evaluation as sharing is not required in this example) results in termination in two steps:

\begin{verbatim}
head ones
  =  \{ applying ones \}
head (1 : ones)
  =  \{ applying head \}
1
\end{verbatim}

This behaviour arises because lazy evaluation proceeds in a “lazy” manner as its name suggests, only evaluating arguments as and when strictly necessary to produce results. For example, when selecting the first element of a list the remainder of the list is not required, and hence in \textit{head (1 : ones)} the further evaluation of \textit{ones} is avoided. More generally, we have the following property: using lazy evaluation, expressions are only evaluated as much as required by the context in which they are used.

Using this idea, we now see that under lazy evaluation \textit{ones} is not an infinite list as such, but rather a \textit{potentially} infinite list, which is only evaluated as much as required by the context. This idea is not restricted to lists, but applies equally to all other data structures in Haskell. For example, this chapter includes a number of exercises concerning infinite trees.

### 12.8 Modular programming

Lazy evaluation also encourages us to separate control from data. For example, we can produce a list of three ones by selecting the first three elements (the control) from the infinite list \textit{ones} (the data):

\begin{verbatim}
> take 3 ones
[1, 1, 1]
\end{verbatim}
Using the definition of the library function \( \text{take} \)

\[
\text{take} \ 0 \ 0 \quad = \ [\]
\text{take} \ (n + 1) \ [] \quad = \ []
\text{take} \ (n + 1) \ (x : xs) \quad = \ x : \text{take} \ n \ xs
\]

this behaviour arises using lazy evaluation as follows:

\[
\text{take} \ 3 \ \text{ones} \\
\quad = \ \{ \ \text{applying} \ \text{ones} \ \} \\
\text{take} \ 3 \ (1 : \text{ones}) \\
\quad = \ \{ \ \text{applying} \ \text{take} \ \} \\
1 : \text{take} \ 2 \ \text{ones} \\
\quad = \ \{ \ \text{applying} \ \text{ones} \ \} \\
1 : \text{take} \ 2 \ (1 : \text{ones}) \\
\quad = \ \{ \ \text{applying} \ \text{take} \ \} \\
1 : 1 : \text{take} \ 1 \ \text{ones} \\
\quad = \ \{ \ \text{applying} \ \text{ones} \ \} \\
1 : 1 : \text{take} \ 1 \ (1 : \text{ones}) \\
\quad = \ \{ \ \text{applying} \ \text{take} \ \} \\
1 : 1 : 1 : \text{take} \ 0 \ \text{ones} \\
\quad = \ \{ \ \text{applying} \ \text{take} \ \} \\
1 : 1 : 1 : []
\]

That is, the data part is only evaluated as much as required by the control part, and the two parts take it in turn to perform reductions. Without using lazy evaluation, we would need to combine the control and data parts into a single function that generates a list of ones of a given length, such as:

\[
\text{makeones} \ :\ Int \to [Int] \\
\text{makeones} \ 0 \quad = \ [] \\
\text{makeones} \ (n + 1) \quad = \ 1 : \text{makeones} \ n
\]

Being able to \textit{modularise} program by separating them into logically distinct parts is an important goal in programming, and being able to separate control from data is one of the most important benefits of lazy evaluation.

Note that some care is still required when programming with infinite lists to avoid non-termination. For example, the expression

\[
\text{filter} \ (\leq 5) \ [1..]
\]

(where \([n..]\) produces the infinite list of integers beginning with \(n\)) will produce the integers 1,2,3,4,5 and then loop forever, because the function \text{filter} keeps testing elements of the infinite list \([1..]\) in a vain attempt to find another that less than or equal to five. In contrast, the expression

\[
\text{takeWhile} \ (\leq 5) \ [1..]
\]
will produce the five integers and then terminate, because takeWhile stops as soon as it finds an element of the list that is greater than five.

We conclude this section with an example concerning prime numbers. In chapter 5 we wrote a program to generate prime numbers up to a given limit. Here is a a simple procedure for generating the infinite sequence of all prime numbers, as opposed to some finite prefix of this sequence:

- Write down the infinite sequence 2, 3, 4, 5, 6, · · ·;
- Mark the first number, p, in the sequence as prime;
- Delete all multiples of p from the sequence;
- Return to the second step.

Note that the first and third steps require an infinite amount of work, and hence in practice the steps must be interleaved. The first few iterations of this procedure can be illustrated as follows:

\[
\begin{aligned}
&\underline{2} 3 4 5 6 7 8 9 10 11 12 13 14 15 \cdots \\
&\underline{3} 5 \underline{7} 9 \underline{11} \underline{13} 15 \cdots \\
&\underline{5} 7 \underline{11} \underline{13} \underline{15} \cdots \\
&7 11 13 \underline{15} \cdots \\
&\underline{11} 13 \underline{15} \cdots \\
&\underline{13} \cdots 
\end{aligned}
\]

Each line corresponds to one iteration of the procedure, with the first number in each line being written in bold to indicate that it is prime (step two), and all multiples of this number being underlined to indicate that they are deleted (step three) prior to the next iteration. In this manner, we can imagine the first line of numbers “falling down the page”, with certain numbers being sieved out at each stage by the underlining, and the bold numbers 2, 3, 5, 7, 11, 13, · · · forming the infinite sequence of prime numbers.

The above procedure for generating prime numbers is known as the sieve of Eratosthenes, after the Greek mathematician who first described it. This procedure can be translated directly into Haskell:

\[
\begin{align*}
\text{primes} & \quad :: \quad [\text{Int}] \\
\text{primes} & \quad = \quad \text{sieve} \ [2\ldots] \\
\text{sieve} & \quad :: \quad [\text{Int}] \rightarrow [\text{Int}] \\
\text{sieve} \ (p : \text{xs}) & \quad = \quad p : \text{sieve} \ [x \mid x \leftarrow \text{xs}, x \mod p \neq 0]
\end{align*}
\]

That is, starting with the infinite list \([2\ldots]\) (step one), we apply the function \text{sieve} that retains the first number \(p\) as being prime (step two), and then calls itself recursively with a new list obtained by filtering all multiples of \(p\) from this list (steps three and four). Lazy evaluation ensures that this program does indeed produce the infinite list of all prime numbers:
> primes
\[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, \cdots \]

By freeing the generation of prime numbers from the constraint of finiteness, we have obtained a modular program on which different control parts can be used in different situations. For example, the first ten prime numbers, and the prime numbers less than ten, can be produced as follows:

> take 10 primes
\[2, 3, 5, 7, 11, 13, 17, 19, 23, 29\]

> takeWhile (<10) primes
\[2, 3, 5, 7\]

12.9 Strict application

Haskell uses lazy evaluation by default, but also provides a special version of function application, written as $!$ and called strict application. An expression of the form \( f \, x \) has the same meaning as the normal application \( f \, x \), except that the “top level” of evaluation of the argument expression \( x \) is forced before the function \( f \) is applied. More formally, an expression \( f \, x \) is only a redex once evaluation of the argument \( x \), using lazy evaluation as normal, has reached the point where it is clear that the result is not an undefined value, at which point the expression can be reduced to the plain application \( f \, x \).

For example, using the definition \( \text{square} \, n = n \times n \), evaluation of \( \text{square} \, (1 + 2) \) proceeds by first evaluating the argument expression \( 1 + 2 \) to give the value 3, and then applying the function \( \text{square} \):

\[
\text{square} \, (1 + 2) \\
= \{ \text{applying } + \} \\
\text{square} \, 3 \\
= \{ \text{applying } \text{square} \} \\
3 \times 3 \\
= \{ \text{applying } \ast \} \\
9
\]

In this example, strict application forces complete evaluation of the argument expression, but this is not always the case. For example, using the definition \( \text{ones} = 1 \, : \, \text{ones} \), evaluation of \( \text{head} \, \text{ones} \) proceeds as with lazy evaluation by only performing one level of evaluation of the argument expression \( \text{ones} \):

\[
\text{head} \, \text{ones} \\
= \{ \text{applying } \text{ones} \} \\
\text{head} \, (1 \, : \, \text{ones})
\]
\[
\begin{align*}
&\text{= } \{ \text{applying } \$! \} \\
&\text{head} \ (1 : \text{ones}) \\
&\text{= } \{ \text{applying } \text{head} \} \\
&1 \\
\end{align*}
\]

When used with a curried function with multiple arguments, strict application can be used to force top-level evaluation of any combination of arguments. For example, if \( f \) is a curried function with two arguments, an application of the form \( f \; x \; y \) can be modified to have three different behaviours:

\[
\begin{align*}
(f \; \$! \; x) \; y &\quad \text{forces top-level evaluation of } x \\
(f \; x) \; \$! \; y &\quad \text{forces top-level evaluation of } y \\
(f \; \$! \; x) \; \$! \; y &\quad \text{forces top-level evaluation of } x \text{ and } y \\
\end{align*}
\]

In Haskell, strict application is mainly used to improve the space performance of programs. For example, consider a function \textit{sumwith} that calculates the sum of a list of integers using an accumulator value:

\[
\begin{align*}
\text{sumwith} &:: \text{Int} \rightarrow [\text{Int}] \rightarrow \text{Int} \\
\text{sumwith} \; v \; [] &\quad = \quad v \\
\text{sumwith} \; v \; (x : xs) &\quad = \quad \text{sumwith} \; (v + x) \; xs
\end{align*}
\]

This function could also be as \( \text{sumwith} = \text{foldl} \ (\_\_\_\_\)\), but the above definition is preferable here. Then using lazy evaluation, we have:

\[
\begin{align*}
\text{sumwith} \; 0 \; [1,2,3] &\quad = \quad \{ \text{applying } \text{sumwith} \} \\
\text{sumwith} \; (0 + 1) \; [2,3] &\quad = \quad \{ \text{applying } \text{sumwith} \} \\
\text{sumwith} \; ((0 + 1) + 2) \; [3] &\quad = \quad \{ \text{applying } \text{sumwith} \} \\
\text{sumwith} \; (((0 + 1) + 2) + 3) \; [] &\quad = \quad \{ \text{applying } \text{sumwith} \} \\
\quad &\quad = (0 + 1) + 2 + 3 \\
\quad &\quad = \quad \{ \text{applying } + \} \\
\quad &\quad = (1 + 2) + 3 \\
\quad &\quad = \quad \{ \text{applying } + \} \\
\quad &\quad = 3 + 3 \\
\quad &\quad = \quad \{ \text{applying } + \} \\
\quad &\quad = 6
\end{align*}
\]

Note that the entire summation \((0 + 1) + 2 + 3\) is constructed before any additions are actually performed. More generally, \textit{sumwith} will construct an expression containing the same number of additions as there are integers in the original list, before a single addition is performed. For example, evaluating \textit{sumwith} \; 0 \; [1 \ldots 10000] \) using Hugs will quickly exhaust the available memory and produce an error message. In practice, it would be preferable to perform
each addition as soon as it is introduced, in order to avoid constructing a large expression during evaluation.

This behaviour can be achieved by redefining \texttt{sumwith} using strict application to force evaluation of its accumulator argument:

\[
\begin{align*}
\texttt{sumwith'} &:: \text{Int} \to \text{[Int]} \to \text{Int} \\
\texttt{sumwith'} \ v \ [] & = v \\
\texttt{sumwith'} \ v \ (x : xs) & = (\texttt{sumwith'} \ $! (v + x)) \ xs
\end{align*}
\]

For example, we now have:

\[
\begin{align*}
\texttt{sumwith'} \ 0 \ [1, 2, 3] & = \{ \text{applying} \ \texttt{sumwith'} \} \\
\texttt{sumwith'} \ $! (0 + 1) \ [2, 3] & = \{ \text{applying} + \} \\
\texttt{sumwith'} \ $! 1 \ [2, 3] & = \{ \text{applying} $! \} \\
\texttt{sumwith'} \ 1 \ [2, 3] & = \{ \text{applying} \ \texttt{sumwith'} \} \\
\texttt{sumwith'} \ $! (1 + 2) \ [3] & = \{ \text{applying} + \} \\
\texttt{sumwith'} \ $! 3 \ [3] & = \{ \text{applying} $! \} \\
\texttt{sumwith'} \ 3 \ [3] & = \{ \text{applying} \ \texttt{sumwith'} \} \\
\texttt{sumwith'} \ $! (3 + 3) \ [] & = \{ \text{applying} + \} \\
\texttt{sumwith'} \ $! 6 \ [] & = \{ \text{applying} $! \} \\
\texttt{sumwith'} \ 6 \ [] & = \{ \text{applying} \ \texttt{sumwith'} \} \\
6 &
\end{align*}
\]

This evaluation requires more steps than previously, due to the additional overhead of using strict application, but now performs each addition as soon as it is introduced rather than constructing a large summation. For example, evaluating \texttt{sumwith'} 0 [1..10000] using Hugs gives the correct result, as opposed to producing an error as previously.

Generalising from the above example, one may wish to define a strict version of the higher-order library function \texttt{foldl} that forces evaluation of its accumulator prior to processing the tail of the list:

\[
\begin{align*}
\texttt{foldl'} &:: (a \to b \to a) \to a \to \text{[b]} \to a \\
\texttt{foldl'} \ f \ v \ [] & = v \\
\texttt{foldl'} \ f \ v \ (x : xs) & = ((\texttt{foldl'} \ f) \ $! (f \ v x)) \ xs
\end{align*}
\]

For example, we can now define \texttt{sumwith'} = \texttt{foldl'} (+).
It should be noted, however, that strict application is not a “silver bullet” that automatically improves the space behaviour of Haskell programs. Even for relatively simple examples, the use of strict application is a specialist topic that requires very careful consideration of the behaviour of lazy evaluation.

12.10 Chapter remarks

Further details about evaluation orders and their properties can be found in [21], and further examples of the use of lazy evaluation for modular programming in the classic Why Functional Programming Matters [8]. A formal meaning for lazy evaluation is given in [15], and a comprehensive tutorial on the efficient implementation of lazy evaluation in [19].

12.11 Exercises

In preparation.
Chapter 13

Reasoning About Programs

DRAFT of February 19, 2005

Introduction in preparation.

13.1 Equational reasoning

At school we learn basic algebraic properties of numbers, such as the fact that
dAddition is commutative and associative, and that multiplication distributes
over addition on both the left and right sides:

\[
\begin{align*}
  x + y &= y + x \\
  x + (y + z) &= (x + y) + z \\
  x * (y + z) &= x * y + x * z \\
  (x + y) * z &= x * z + y * z
\end{align*}
\]

Such properties can be used directly to reason about numbers, and to establish
other useful properties. For example, using the above equations we can show
that the multiplication \((x + a) * (x + b)\) can be rewritten as an addition \(x * x + (a + b) * x + a * b\), by the following calculation:

\[
\begin{align*}
  (x + a) * (x + b) &= \{ \text{left distributivity} \} \\
  (x + a) * x + (x + a) * b &= \{ \text{right distributivity} \} \\
  x * x + a * x + x * b + a * b &= \{ \text{commutativity of +} \} \\
  x * x + a * x + b * x + a * b &= \{ \text{right distributivity} \} \\
  x * x + (a + b) * x + a * b
\end{align*}
\]

Note that in this calculation we follow the common practice of implicitly ex-
ploting associativity properties, in this case the associativity of addition by
omitting are parentheses when more than one addition is used in sequence.
As well as being interesting in their own right, algebraic properties can also have a computational significance. For example, the expression \( x \times (y + z) \) requires two operations (one multiplication and one addition), whereas the equivalent expression \( x \times y + x \times z \) requires three (two multiplications and one addition). Hence even though these two expressions are algebraically equal, from the point of view of efficiency, the former is preferable to the latter.

Such equational reasoning can also be used to reason about Haskell. For example, in this context an equation such as \( x + y = y + x \) means that for any expressions \( x \) and \( y \) of the same numeric type, evaluation of \( x + y \) and \( y + x \) will always produce the same numeric value. In contrast with the normal practice of mathematics, however, in programming languages the issue of termination is of central importance. For example, what does the above equation mean if evaluation of either side does not terminate?

In Haskell, non-termination is viewed as a special value in its own right, written as \( \bot \). Every type in Haskell implicitly includes this value, which is called \textit{bottom}, because it plays the role of the smallest, or bottom, value of every type. In this manner, the equation \( x + y = y + x \) states that evaluation of the expressions \( x + y \) and \( y + x \) either both terminate with the same numeric value, or do not terminate (both produce the value \( \bot \)).

At this point, some readers may be wondering why the commutativity of addition was not stated as \( x + y \equiv y + x \), using the equality operator \( \equiv \) provided by Haskell? The answer is that our goal is to use mathematics as a language to reason about expressions in Haskell, rather than using Haskell as a language to reason about itself, which would be somewhat circular!

When reasoning about Haskell programs, we don’t just use properties of built-in operations of the language such as addition and multiplication, but also use the equations from which user-defined functions are constructed. For example, consider the function that doubles an integer, defined as follows:

\[
\begin{align*}
double & :: \; \text{Int} \rightarrow \text{Int} \\
double \; x & = \; x + x
\end{align*}
\]

As well as being viewed as the \textit{definition} of a function, this equation can also be viewed as a \textit{property} that can be used when reasoning about this function. In particular, as a logical property the above equation states that for any integer expression \( x \), the expression \( \text{double} \; x \) can freely be replaced by \( x + x \), and conversely, that the expression \( x + x \) can freely be replaced by \( \text{double} \; x \).

In this manner, when reasoning about programs, function definitions can be both \textit{applied} from left-to-right and \textit{unapplied} from right-to-left.

However, some care is required when viewing definitions with multiple equations as logical properties. For example, consider a function that decides if an integer is zero, defined as follows:

\[
\begin{align*}
\text{isZero} & :: \; \text{Int} \rightarrow \text{Bool} \\
\text{isZero} \; 0 & = \; \text{True} \\
\text{isZero} \; n & = \; \text{False}
\end{align*}
\]
The first equation, $\text{isZero } 0 = \text{True}$, can freely be viewed as a logical property that can be applied in both directions. However, this is not the case for the second equation, $\text{isZero } n = \text{False}$. In particular, because the order in which the equations are written matters, an expression of the form $\text{isZero } n$ can only be replaced by $\text{False}$ provided that $n \neq 0$, as in the case when $n = 0$ the first equation applies. Dually, it only valid to unapply the equation $\text{isZero } n = \text{False}$ and replace $\text{False}$ by an expression of the form $\text{isZero } n$ in the case when $n \neq 0$, for the same reason.

In summary, when a function is defined using multiple equations, the equations cannot be viewed as logical properties in isolation from one another, but need to be interpreted in light of order in which patterns are matched within the equations. For this reason, it is preferable to define functions in a manner that does not rely on the order in which their equations are written. For example, if we write the above definition using a guard

$$
\text{isZero } 0 \quad = \quad \text{True} \\
\text{isZero } n \mid n \neq 0 \quad = \quad \text{False}
$$

then it is now explicitly clear that $\text{isZero } n$ can only be replaced by $\text{False}$, and conversely that $\text{False}$ can only be replaced by $\text{isZero } n$, when the side condition that $n \neq 0$ is satisfied. Patterns that do not rely on the order in which they are matched are called disjoint or non-overlapping. In order to simplify the process of reasoning about programs, it is good practice to use non-overlapping patterns whenever possible. For example, most functions in the standard library given in Appendix B are defined in this manner.

We conclude this section by noting that lazy evaluation is also relevant when viewing function definitions as logical properties. For example, consider the library function that selects the first component from a pair:

$$
\begin{align*}
\text{fst} & \quad :: \quad (a, b) \rightarrow a \\
\text{fst} \ (x, y) & \quad = \quad x
\end{align*}
$$

Under lazy evaluation, where arguments are passed by name, the equation $\text{fst} \ (x, y) = x$ can freely be used as a logical property for all expressions $x$ and $y$, including non-terminating expressions, because evaluation of the argument expressions $x$ and $y$ is not forced by the application of $\text{fst}$.

In contrast, in a language where arguments are passed by value, the equation $\text{fst} \ (x, y) = x$ is invalid as a logical property when $y$ does not terminate. For example, if $x = 1$ and $y = \bot$ then the equation $\text{fst} \ (x, y) = x$ reduces to the invalid equation $\bot = 1$, because non-termination of the argument $y$ results in non-termination of $\text{fst} \ (x, y)$. Hence, in order to view $\text{fst} \ (x, y) = x$ as a logical property in a call-by-value language we would require the side-condition that evaluation of $y$ terminates, despite the fact that $y$ plays no role in the result. Using lazy evaluation avoids many such termination side-conditions, and hence simplifies the process of reasoning about program.
13.2 Simple examples

As a simple example of equational reasoning in Haskell, recall the following
definition of the reverse function on lists:

\[
\begin{align*}
\text{reverse} & :: [a] \to [a] \\
\text{reverse} \; [] & = [] \\
\text{reverse} \; (x : xs) & = \text{reverse} \; xs :: [x]
\end{align*}
\]

Using this definition we can show that \text{reverse} has no effect on singleton lists,
in the sense that \text{reverse} \; [x] = [x] for any \; x:

\[
\begin{align*}
\text{reverse} \; [x] \\
& = \{ \text{list notation} \} \\
& = \{ \text{applying } \text{reverse} \} \\
& = \{ \text{applying } \text{reverse} \} \\
& = \{ \text{applying } \_ \_ \} \\
& = [x]
\end{align*}
\]

Hence any expression of the form \text{reverse} \; [x] in a program can freely be
replaced by \; [x] without change in meaning, but with a change in efficiency by
avoiding the need to use the \text{reverse} function.

As another example, suppose that we define a function that decides if a
non-negative integer is even as follows:

\[
\begin{align*}
\text{even} & :: \text{Int} \to \text{Bool} \\
\text{even} \; 0 & = \text{True} \\
\text{even} \; (n + 1) & = \neg (\text{even} \; n)
\end{align*}
\]

(Of course this is not an efficient way to decide this, but this is not the point
here.) Then we can show that \text{even} \; (n + 2) = \text{even} \; n for all non-negative
integers \; n by the following simple calculation:

\[
\begin{align*}
\text{even} \; (n + 2) \\
& = \{ \text{applying } \text{even} \} \\
& = \neg (\text{even} \; (n + 1)) \\
& = \{ \text{applying } \text{even} \} \\
& = \neg (\neg (\text{even} \; n)) \\
& = \{ \text{property of } \neg \} \\
& = \text{even} \; n
\end{align*}
\]

This calculation exploits the fact that \neg (\neg b) = b for all logical values \; b,
which means that \neg is its own inverse. It is very common when reasoning
about programs to make use of other properties that remain to be verified. Such properties are called \textit{lemmas}. Recall the definition of the function \(\neg\):

\[
\neg \quad :: \quad \text{Bool} \rightarrow \text{Bool} \\
\neg \text{False} \quad = \quad \text{True} \\
\neg \text{True} \quad = \quad \text{False}
\]

Because this function is defined by pattern matching, properties of \(\neg\) are normally proved by case analysis. For example, in the case when \(b = \text{False}\), the above lemma for \(\neg\) is verified as follows:

\[
\neg (\neg \text{False}) \\
= \quad \{ \text{applying } \neg \} \\
\neg \text{True} \\
= \quad \{ \text{applying } \neg \} \\
\text{False}
\]

The case for \(b = \text{True}\) can be verified similarly.

### 13.3 Structural induction

Most interesting functional programs involve recursion. Reasoning about such programs normally proceeds using the simple but powerful technique of \textit{structural induction}. Let us begin by recalling the simplest example of a recursive datatype from chapter 10, namely the type of natural numbers:

\[
\textbf{data Nat} \quad = \quad \text{Zero} \mid \text{Succ Nat}
\]

This type definition states that \text{Zero} is a value of type \text{Nat} (the base case), and that if \(n\) is a value of type \text{Nat} then so is \text{Succ} \(n\) (the recursive case). Implicit in the definition is the fact that \text{Zero} and \text{Succ} are the only constructors for the type \text{Nat}. Hence, the values of type \text{Nat} can be enumerated as follows:

\[
\begin{align*}
\text{Zero} \\
\text{Succ} \text{ Zero} \\
\text{Succ} (\text{Succ} \text{ Zero}) \\
\text{Succ} (\text{Succ} (\text{Succ} \text{ Zero})) \\
\vdots
\end{align*}
\]

For simplicity, we only consider the finite natural numbers, obtained by starting from \text{Zero} and applying \text{Succ} a finite number of times. In particular, we don’t consider infinity, which can be defined by \text{infinity} = \text{Succ} \text{ infinity} and is a perfectly well defined value in Haskell because of lazy evaluation.

Now suppose we want to prove that some property, \(p\) say, is true for all (finite) natural numbers. Then the principle of structural induction states that it is sufficient to show that \(p\) is true for \text{Zero}, called the \textit{base case}, and that \(p\) is preserved by \text{Succ}, called the \textit{inductive case}. More precisely, in the
inductive case one is required to show that if the property \( p \) is true for any natural number \( n \), which assumption is called the *induction hypothesis*, then it is also true for the natural number \( \text{Succ} \ n \).

Why are the two cases in structural induction sufficient to show that \( p \) is true for all natural numbers? For example, how does it then follow that \( p \) is true for \( \text{Succ} \ (\text{Succ} \ \text{Zero}) \)? Starting from the fact that we have shown that \( p \) is true for \( \text{Zero} \) (the base case), we can then apply the inductive case once to conclude that \( p \) is true for \( \text{Succ} \ \text{Zero} \), by taking \( n = \text{Zero} \), and then apply the induction case a second time to conclude that \( p \) is true for \( \text{Succ} \ (\text{Succ} \ \text{Zero}) \), by taking \( n = \text{Succ} \ \text{Zero} \). In a similar manner, the validity of \( p \) can be established for any finite natural number \( n \).

Thinking of the “domino effect” may also be useful. Suppose you have a line of dominoes standing on end and you can be sure that the first domino will fall, and that whenever a domino falls then its next neighbour will also fall. Then it is clear that all the dominoes will fall by applying the first fact to get the process started and repeatedly applying the second to keep it going. The same pattern of reasoning occurs with structural induction: we first verify the property for \( \text{Zero} \) (the first domino falls), then that the property is preserved by \( \text{Succ} \) (if any domino falls then so will its neighbour), and can then conclude that the property is true for all natural numbers (all dominos fall).

As a concrete example, consider the definition of a recursive function that takes two natural numbers and adds them together:

\[
\begin{align*}
\text{add} & : \ Nat \rightarrow Nat \rightarrow Nat \\
\text{add} \ \text{Zero} \ m & = m \\
\text{add} \ (\text{Succ} \ n) \ m & = \text{Succ} \ (\text{add} \ n \ m)
\end{align*}
\]

From the first equation it is immediate that \( \text{add} \ \text{Zero} \ m = m \) is true for any natural number \( m \). Now let us show that the property \( \text{add} \ n \ \text{Zero} = n \), which we abbreviate by \( p \ n \), is also true for all natural numbers \( n \). We proceed by structural induction on \( n \). The base case, showing that \( p \ \text{Zero} \) is true, amounts to showing that \( \text{add} \ \text{Zero} \ \text{Zero} = \text{Zero} \), which is trivially true:

\[
\begin{align*}
\text{add} \ \text{Zero} \ \text{Zero} \\
= & \quad \{ \text{definition of add} \} \\
\text{Zero}
\end{align*}
\]

For the inductive case, we must show that if \( p \ n \) is true for any natural number \( n \), then \( p \ (\text{Succ} \ n) \) is also true. That is, using the induction hypothesis \( \text{add} \ n \ \text{Zero} = n \) as an assumption, we must show that \( \text{add} \ (\text{Succ} \ n) \ \text{Zero} = \text{Succ} \ n \), which can be verified by the following calculation:

\[
\begin{align*}
\text{add} \ (\text{Succ} \ n) \ \text{Zero} \\
= & \quad \{ \text{applying add} \} \\
\text{Succ} \ (\text{add} \ n \ \text{Zero}) \\
= & \quad \{ \text{induction hypothesis} \} \\
\text{Succ} \ n
\end{align*}
\]
As another example, let us now show that addition of natural numbers is associative: \( add \ x \ (add \ y \ z) = add \ (add \ x \ y) \ z \) for all natural numbers \( x, y \) and \( z \). This time there are three variables to choose from, so which should structural induction be performed over? Note that the \( add \) function is defined by pattern matching on its first argument, so it is natural to carry this over to the equation and try induction on \( x \), which appears twice as the first argument to \( add \), whereas \( y \) only appears once as such and \( z \) never. Here is the proof of the associativity equation by structural induction on \( x \).

Base case:

\[
\begin{align*}
add \ & \text{Zero} \ (add \ y \ z) \\
= & \quad \{ \text{applying the outer } add \} \\
& \quad add \ y \ z \\
= & \quad \{ \text{unapplying } add \} \\
& \quad add \ (add \ \text{Zero} \ y) \ z
\end{align*}
\]

Inductive case:

\[
\begin{align*}
add \ & \text{Succ} \ n \ (add \ y \ z) \\
= & \quad \{ \text{applying the outer } add \} \\
& \quad \text{Succ} \ (add \ n \ (add \ y \ z)) \\
= & \quad \{ \text{induction hypothesis} \} \\
& \quad \text{Succ} \ (add \ (add \ n \ y) \ z) \\
= & \quad \{ \text{unapplying the outer } add \} \\
& \quad add \ (\text{Succ} \ (add \ n \ y) \ z) \\
= & \quad \{ \text{unapplying the inner } add \} \\
& \quad add \ (add \ (\text{Succ} \ n) \ y) \ z
\end{align*}
\]

Note that both calculations start by applying definitions, and conclude by unapplying definitions. This pattern is typical in proofs by structural induction, but the latter part may seem somewhat mysterious at first sight. In particular, knowing which definitions to unapply seems to require a degree of foresight. In practice, however, if one becomes stuck at a certain point during such a calculation, progress can often be made by focussing on the desired end result and trying to work backwards to the point where one became stuck.

For example, after applying the induction hypothesis in the latter calculation above to obtain \( \text{Succ} \ (add \ (add \ n \ y) \ z) \), it may not be clear how to proceed, as there are no more definitions that can be applied. However, if we then focus on the expression that we are aiming towards, \( add \ (add \ (\text{Succ} \ n) \ y) \ z \), we can simply apply the inner \( add \) and then the outer \( add \) to produce the expression at which we became stuck, which process can then be reversed (turning applying into unapplying) to complete the calculation.

The remainder of this chapter is in preparation.
Appendix A

Symbol Table

In this appendix we present a table that summarises the meaning of each of the special Haskell symbols that are used in this book, and shows how each of the symbols is typed using a normal keyboard.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>→</td>
<td>maps to</td>
<td>-&gt;</td>
</tr>
<tr>
<td>⇒</td>
<td>constrains</td>
<td>=&gt;</td>
</tr>
<tr>
<td>≥</td>
<td>at least</td>
<td>&gt;=</td>
</tr>
<tr>
<td>≤</td>
<td>at most</td>
<td>&lt;=</td>
</tr>
<tr>
<td>≠</td>
<td>inequality</td>
<td>/=</td>
</tr>
<tr>
<td>∧</td>
<td>conjunction</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>∨</td>
<td>disjunction</td>
<td></td>
</tr>
<tr>
<td>¬</td>
<td>negation</td>
<td>not</td>
</tr>
<tr>
<td>↑</td>
<td>exponentiation</td>
<td>^</td>
</tr>
<tr>
<td>⨿</td>
<td>composition</td>
<td>.</td>
</tr>
<tr>
<td>λ</td>
<td>abstraction</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>append</td>
<td>++</td>
</tr>
<tr>
<td>←</td>
<td>drawn from</td>
<td>&lt;-</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>then</td>
<td>&gt;&gt;=</td>
</tr>
<tr>
<td>+++</td>
<td>choice</td>
<td>+++</td>
</tr>
</tbody>
</table>
Appendix B

Haskell Standard Prelude

In this appendix we present some of the most commonly used definitions from the standard prelude. For clarity, a number of the definitions have been simplified or modified from those given in the Haskell Report [18]. Other standard libraries are described in the Haskell Library Report [17].

B.1 Classes

Equality types:

```haskell
class Eq a where
  (==), (≠) :: a → a → Bool
  x ≠ y = ¬ (x == y)
```

Ordered types:

```haskell
class Eq a ⇒ Ord a where
  (<), (≤), (>), (≥) :: a → a → Bool
  min, max :: a → a → a
  min x y | x ≤ y = x
          | otherwise = y
  max x y | x ≤ y = y
          | otherwise = x
```

Showable types:

```haskell
class Show a where
  show :: a → String
```

Readable types:

```haskell
class Read a where
  read :: String → a
```

157
**Numeric types:**

class (Eq a, Show a) ⇒ Num a where
  (+), (−), (∗) :: a → a → a
  negate, abs, signum :: a → a

**Integral types:**

class Num a ⇒ Integral a where
  div, mod :: a → a → a

**Fractional types:**

class Num a ⇒ Fractional a where
  (/) :: a → a → a
  recip :: a → a
  recip n = 1 / n

### B.2 Booleans

Type declaration:

data Bool = False | True
  deriving (Eq, Ord, Show, Read)

Logical conjunction:

  (∧) :: Bool → Bool → Bool
  False ∧ _ = False
  True ∧ b = b

Logical disjunction:

  (∨) :: Bool → Bool → Bool
  False ∨ b = b
  True ∨ _ = True

Logical negation:

  ¬ :: Bool → Bool
  ¬ False = True
  ¬ True = False

Guard that always succeeds:

  otherwise :: Bool
  otherwise = True
B.3 Characters and strings

Type declarations:

```haskell
data Char = ... deriving (Eq, Ord, Show, Read)

type String = [Char]
```

Decide if a character is a lower-case letter:

```haskell
isLower :: Char → Bool
isLower c = c ≥ 'a' ∧ c ≤ 'z'
```

Decide if a character is an upper-case letter:

```haskell
isUpper :: Char → Bool
isUpper c = c ≥ 'A' ∧ c ≤ 'Z'
```

Decide if a character is alphabetic:

```haskell
isAlpha :: Char → Bool
isAlpha c = isLower c ∨ isUpper c
```

Decide if a character is a digit:

```haskell
isDigit :: Char → Bool
isDigit c = c ≥ '0' ∧ c ≤ '9'
```

Decide if a character alpha-numeric:

```haskell
isAlphaNum :: Char → Bool
isAlphaNum c = isAlpha c ∨ isDigit c
```

Decide if a character is a spacing character:

```haskell
isSpace :: Char → Bool
isSpace c = elem c "\t\n"
```

Convert a character to a Unicode number:

```haskell
ord :: Char → Int
ord = ... 
```

Convert a Unicode number to a character:

```haskell
chr :: Int → Char
chr = ...
```

Convert a digit to an integer:

```haskell
digitToInt :: Char → Int
digitToInt c | isDigit c = ord c − ord '0'
```

159
Convert an integer to a digit:

\[
\text{intToDigit} :: \text{Int} \rightarrow \text{Char} \\
\text{intToDigit } n \quad | \quad n \geq 0 \land n \leq 9 = \text{chr } (\text{ord } '0' + n)
\]

Convert a letter to lower-case:

\[
\text{toLowerCase} :: \text{Char} \rightarrow \text{Char} \\
\text{toLowerCase } c \mid \text{isUpper } c = \text{chr } (\text{ord } c - \text{ord } 'A' + \text{ord } 'a') \\
| \text{otherwise } = c
\]

Convert a letter to upper-case:

\[
\text{toUpperCase} :: \text{Char} \rightarrow \text{Char} \\
\text{toUpperCase } c \mid \text{isLower } c = \text{chr } (\text{ord } c - \text{ord } 'a' + \text{ord } 'A') \\
| \text{otherwise } = c
\]

B.4 Numbers

Type declarations:

\[
\textbf{data} \ \text{Int} = \ldots \\
\quad \text{deriving } (\text{Eq, Ord, Show, Read,} \\
\quad \text{Num, Integral})
\]

\[
\textbf{data} \ \text{Integer} = \ldots \\
\quad \text{deriving } (\text{Eq, Ord, Show, Read,} \\
\quad \text{Num, Integral})
\]

\[
\textbf{data} \ \text{Float} = \ldots \\
\quad \text{deriving } (\text{Eq, Ord, Show, Read,} \\
\quad \text{Num, Fractional})
\]

Decide if an integer is even:

\[
\text{even} :: \text{Integral } a \Rightarrow a \rightarrow \text{Bool} \\
\text{even } n = n \text{ 'mod' } 2 == 0
\]

Decide if an integer is odd:

\[
\text{odd} :: \text{Integral } a \Rightarrow a \rightarrow \text{Bool} \\
\text{odd } = \neg \circ \text{even}
\]

Exponentiation:

\[
(\uparrow) :: (\text{Num } a, \text{Integral } b) \Rightarrow a \rightarrow b \rightarrow a \\
\quad a \uparrow 0 = 1 \\
\quad x \uparrow (n + 1) = x \ast (x \uparrow n)
\]

160
B.5 Tuples

Type declarations:

\[
\text{data } () = \ldots
\]
\[
\text{deriving } (\text{Eq}, \text{Ord}, \text{Show}, \text{Read})
\]
\[
\text{data } (a, b) = \ldots
\]
\[
\text{deriving } (\text{Eq}, \text{Ord}, \text{Show}, \text{Read})
\]
\[
\text{data } (a, b, c) = \ldots
\]
\[
\text{deriving } (\text{Eq}, \text{Ord}, \text{Show}, \text{Read})
\]

Select the first component of a pair:

\[
fst \quad :: \quad (a, b) \to a
\]
\[
fst (x, -) = x
\]

Select the second component of a pair:

\[
snd \quad :: \quad (a, b) \to b
\]
\[
snd (- y) = y
\]

B.6 Lists

Type declaration:

\[
\text{data } [a] = [] \mid a : [a]
\]
\[
\text{deriving } (\text{Eq}, \text{Ord}, \text{Show}, \text{Read})
\]

Decide if a list is empty:

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

Decide if a value is an element of a list:

\[
elem \quad :: \quad \text{Eq } a \Rightarrow a \to [a] \to \text{Bool}
\]
\[
elem x xs = \text{any } (== x) \; xs
\]

Decide if all logical values in a list are True:

\[
\text{and} \quad :: \quad [\text{Bool}] \to \text{Bool}
\]
\[
\text{and} = \text{foldr } (\land) \; \text{True}
\]

Decide if any logical value in a list is False:

\[
or \quad :: \quad [\text{Bool}] \to \text{Bool}
\]
\[
or = \text{foldr } (\lor) \; \text{False}
\]

Decide if all elements of a list satisfy a predicate:
all :: (a → Bool) → [a] → Bool
all p = and ◦ map p

Decide if any element of a list satisfies a predicate:

any :: (a → Bool) → [a] → Bool
any p = or ◦ map p

Select the first element of a non-empty list:

head :: [a] → a
head (x : _) = x

Select the last element of a non-empty list:

last :: [a] → a
last [x] = x
last ((_:xs)) = last xs

Select the nth element of a list:

(!!)
(x : _)!! 0 = x
(_ : xs)!! (n + 1) = xs !! n

Select the first n elements of a list:

take :: Int → [a] → [a]
take 0 _ = []
take (n + 1) [] = []
take (n + 1) (x : xs) = x : take n xs

Select all elements of a list that satisfy a predicate:

filter :: (a → Bool) → [a] → [a]
filter p xs = [x | x ← xs, p x]

Select elements of a list while they satisfy a predicate:

takeWhile :: (a → Bool) → [a] → [a]
takeWhile _ [] = []
takeWhile p (x : xs)
| p x = x : takeWhile p xs
| otherwise = []

Remove the first element from a non-empty list:

tail :: [a] → [a]
tail (_ : xs) = xs

Remove the last element from a non-empty list:
\texttt{init} :: \([a] \rightarrow [a]\\n\texttt{init [\_]} = []\\n\texttt{init (x : xs)} = x : \texttt{init xs}

Remove the first \(n\) elements from a list:
\[
\begin{align*}
\texttt{drop} & :: \text{Int} \rightarrow [a] \rightarrow [a] \\
\texttt{drop 0 xs} & = xs \\
\texttt{drop (n + 1) []} & = [] \\
\texttt{drop (n + 1) (\_ : xs)} & = \texttt{drop n xs}
\end{align*}
\]

Remove elements from a list while they satisfy a predicate:
\[
\begin{align*}
\texttt{dropWhile} & :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \\
\texttt{dropWhile []} & = [] \\
\texttt{dropWhile p (x : xs)} & \\
| \texttt{p x} & = \texttt{dropWhile p xs} \\
| \texttt{otherwise} & = x : \texttt{xs}
\end{align*}
\]

Split a list at the \(n\)th element:
\[
\begin{align*}
\texttt{splitAt} & :: \text{Int} \rightarrow [a] \rightarrow ([a], [a]) \\
\texttt{splitAt n xs} & = (\text{take n xs}, \text{drop n xs})
\end{align*}
\]

Split a list using a predicate:
\[
\begin{align*}
\texttt{span} & :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow ([a], [a]) \\
\texttt{span p xs} & = (\text{takeWhile p xs}, \text{dropWhile p xs})
\end{align*}
\]

Process a list using a right-bracketing function:
\[
\begin{align*}
\texttt{foldr} & :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\
\texttt{foldr v []} & = v \\
\texttt{foldr f v (x : xs)} & = f x (\texttt{foldr f v xs})
\end{align*}
\]

Process a non-empty list using a right-bracketing function:
\[
\begin{align*}
\texttt{foldr1} & :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a \\
\texttt{foldr1 [x]} & = x \\
\texttt{foldr1 f (x : xs)} & = f x (\texttt{foldr1 f xs})
\end{align*}
\]

Process a list using a left-bracketing function:
\[
\begin{align*}
\texttt{foldl} & :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \\
\texttt{foldl v []} & = v \\
\texttt{foldl f v (x : xs)} & = \texttt{foldl f (f v x) xs}
\end{align*}
\]

Process a non-empty list using a left-bracketing function:
\[
\begin{align*}
\texttt{foldl1} & :: (a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a \\
\texttt{foldl1 f (x : xs)} & = \texttt{foldl f x xs}
\end{align*}
\]

Produce an infinite list of identical elements:
repeat :: \( a \rightarrow [a] \)
repeat \( x \) = \( xs \) where \( xs = x : xs \)

Produce a list with \( n \) identical elements:

replicate :: \( \text{Int} \rightarrow a \rightarrow [a] \)
replicate \( n \) = take \( n \circ \text{repeat} \)

Produce an infinite list by iterating a function over a value:

iterate :: \((a \rightarrow a) \rightarrow a \rightarrow [a]\)
iterate \( f \) \( x \) = \( x : \text{iterate} \ f \ (f \ x) \)

Produce a list of pairs from a pair of lists:

zip :: \([a] \rightarrow [b] \rightarrow [(a, b)]\)
zip \([\_\_]\) = \([\_]\)
zip \((x : xs) (y : ys)\) = \((x, y) : \text{zip} \ xs \ ys\)

Calculate the length of a list:

length :: \([a] \rightarrow \text{Int} \)
length = \( \text{foldl} \ (\lambda n \_ \rightarrow n + 1) \ 0 \)

Calculate the sum of a list of numbers:

sum :: \(\text{Num} \ a \Rightarrow [a] \rightarrow a \)
sum = \( \text{foldl} \ (+) \ 0 \)

Calculate the product of a list of numbers:

product :: \(\text{Num} \ a \Rightarrow [a] \rightarrow a \)
product = \( \text{foldl} \ (*) \ 1 \)

Calculate the minimum of a non-empty list:

minimum :: \(\text{Ord} \ a \Rightarrow [a] \rightarrow a \)
minimum = \( \text{foldl1} \ \text{min} \)

Calculate the maximum of a non-empty list:

maximum :: \(\text{Ord} \ a \Rightarrow [a] \rightarrow a \)
maximum = \( \text{foldl1} \ \text{max} \)

Append two lists:

\((++\) :: \([a] \rightarrow [a] \rightarrow [a] \)
\([\_] ++ ys\) = \( ys \)
\((x : xs) ++ ys\) = \( x : (xs ++ ys) \)

Concatenate a list of lists:
concat :: [] → [a]
concat = foldr (++) []

Reverse a list:
reverse :: [a] → [a]
reverse = foldl (λxs x → x : xs) []

Apply a function to all elements of a list:
map :: (a → b) → [a] → [b]
map f xs = [f x | x ← xs]

B.7 Functions

Type declaration:
data a → b = · · ·

Identity function:
id :: a → a
id = λx → x

Function composition:
(∘) :: (b → c) → (a → b) → (a → c)
f ∘ g = λx → f (g x)

Constant functions:
const :: a → (b → a)
const x = λ_ → x

Strict application:
($) :: (a → b) → a → b
f $! x = · · ·

Convert a function on pairs to a curried function:
curry :: ((a, b) → c) → (a → b → c)
curry f = λx y → f (x, y)

Convert a curried function to a function on pairs:
uncurry :: (a → b → c) → ((a, b) → c)
uncurry f = λ(x, y) → f x y
B.8 Actions

Type declaration:
\[
data IO a = \cdots
\]

Read a character from the keyboard:
\[
\begin{align*}
getChar & \quad :: \quad IO \ Char \\
getChar & \quad = \quad \cdots
\end{align*}
\]

Read a string from the keyboard:
\[
\begin{align*}
\text{getLine} & \quad :: \quad IO \ String \\
\text{getLine} & \quad = \quad \text{do} \ x \gets \text{getChar} \\
& \quad \quad \quad \text{if} \ x == '\n' \ \text{then} \\
& \quad \quad \quad \quad \text{return} \ "" \\
& \quad \quad \quad \text{else} \\
& \quad \quad \quad \quad \text{do} \ xs \gets \text{getLine} \\
& \quad \quad \quad \quad \text{return} \ (\text{read} \ xs)
\end{align*}
\]

Read a value from the keyboard:
\[
\begin{align*}
\text{readLn} & \quad :: \quad \text{Read} \ a \Rightarrow IO \ a \\
\text{readLn} & \quad = \quad \text{do} \ xs \gets \text{getLine} \\
& \quad \quad \quad \text{return} \ (\text{read} \ xs)
\end{align*}
\]

Write a character to the screen:
\[
\begin{align*}
\text{putChar} & \quad :: \quad Char \rightarrow IO \ () \\
\text{putChar} & \quad = \quad \cdots
\end{align*}
\]

Write a string to the screen:
\[
\begin{align*}
\text{putStr} & \quad :: \quad String \rightarrow IO \ () \\
\text{putStr} \ "" & \quad = \quad \text{return} \ () \\
\text{putStr} \ (x : xs) & \quad = \quad \text{do} \ \text{putChar} \ x \\
& \quad \quad \quad \text{putStr} \ xs
\end{align*}
\]

Write a string to the screen and move to a new line:
\[
\begin{align*}
\text{putStrLn} & \quad :: \quad String \rightarrow IO \ () \\
\text{putStrLn} \ xs & \quad = \quad \text{do} \ \text{putStr} \ xs \\
& \quad \quad \quad \text{putChar} \ '\n'
\end{align*}
\]

Write a value to the screen:
\[
\begin{align*}
\text{print} & \quad :: \quad Show \ a \Rightarrow a \rightarrow IO \ () \\
\text{print} & \quad = \quad \text{putStrLn} \circ \text{show}
\end{align*}
\]

Display an error message and terminate the program:
\[
\begin{align*}
\text{error} & \quad :: \quad String \rightarrow a \\
\text{error} & \quad = \quad \cdots
\end{align*}
\]
Bibliography


167

of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Pro-
gramming Languages, pages 144–154, Charleston, South Carolina, January 1993.


[19] Simon Peyton Jones and David Lester. Implementing Functional
research.microsoft.com/~simonpj.


[21] John C. Reynolds. Theories of Programming Languages. Cambridge Uni-