An Introduction to Computing
with Haskell

Manuel M. T. Chakravarty
Gabriele C. Keller
Preface

This text is meant as an introduction to computing for the absolute beginner. It teaches fundamental concepts of programming and simple forms of reasoning about programs. The aim is not to learn a single programming language, but to understand elementary concepts that illustrate the nature of computing. The language Haskell has been chosen as a vehicle to explain these concepts for a variety of reasons. It is a clean language that is light on syntax, which makes it easy to learn and simplifies the presentation of important concepts. It is a high-level language that leads to concise programs, which takes the text more quickly to non-trivial examples, and thus benefits the development and presentation of programs in the classroom. It is a member of the family of functional languages, which simplify reasoning about programs, due to their clean semantics. Last, but not least, the language tends to serve as an equaliser in introductory classes, as few students tend to have prior knowledge, which puts everybody on more equal footing.

In addition to programming concepts and reasoning about programs, the text includes an introduction to Unix-like programming environments. This furthers the understanding of the context in which programs are executed and, in particular, reinforces essential concepts of input/output programming.

In summary, this book strives to achieve a careful balance between the conceptual and practical aspects of programming. In addition, it takes first careful steps towards a theoretical understanding of programs, and thus towards computing as a scientific discipline.

Software

Programming requires practice, which in turn requires an implementation of the language Haskell used in this book. The text attempts to stay independent of the details of an Haskell implementation; however, in some places, we need to commit to a specific system. We will use the Glasgow Haskell Compiler, which is freely available from

http://haskell.org/ghc/

Similar comments apply to examples involving Unix commands. We suggest the use of the free GNU/Linux system, which is widely available.

Acknowledgements

We thank our colleagues Richard Buckland, Tim Lambert, and John Plaice, who have been involved with teaching first-year computing at the University of New South Wales, for their suggestions and feedback on the lecture notes that became the present book. Moreover, we thank the student tutors who helped us teaching COMP1011 for their valuable comments on the course material.
Contents

Preface iii

1 Introduction 1
  1.1 Programming Languages ............................................. 1
  1.2 Values, Functions and Types ....................................... 2
    1.2.1 Values Versus Types ........................................... 2
    1.2.2 Functions: Canned Computations ............................... 3
    1.2.3 Type signatures .................................................. 4
    1.2.4 Multiple Arguments .............................................. 4
    1.2.5 A First Glance at Overloading .................................. 5
  1.3 Exercises ............................................................. 6

2 Programming Environment 7
  2.1 The Unix Family of Operating Systems ............................. 7
    2.1.1 Multi-user Operating Systems .................................. 7
    2.1.2 Multi-tasking Operating Systems ................................ 8
    2.1.3 Files and Directories ............................................ 8
  2.2 Executing Haskell Programs ....................................... 8
  2.3 Exercises ............................................................. 9

3 Basic Control Structures and Types 11
  3.1 Programs are Composed from Modules ............................. 11
  3.2 Branches in the Control Flow ..................................... 12
    3.2.1 Choices are Implemented Through Conditionals ................ 12
    3.2.2 Guards .......................................................... 12
  3.3 Binders ............................................................... 13
    3.3.1 How to Associate Names with Values ........................... 13
  3.4 Tuples: Combining Different Data Items .......................... 14
    3.4.1 Example: Points ................................................ 14
    3.4.2 Example: Colour Points ....................................... 15
    3.4.3 Important Symmetries in Haskell .............................. 15
    3.4.4 Special Names for Some Tuples ................................ 16
  3.5 Lists: Many Values of a Single Type ............................. 16
    3.5.1 Useful List Functions .......................................... 16
    3.5.2 Lists Versus Tuples ............................................. 17
    3.5.3 Strings as Lists ............................................... 18
  3.6 Layout .................................................................. 18
  3.7 Exercises ............................................................. 19
## 7 Input and Output

7.1 Interpreter Versus Compiler Revisited ................................. 57
7.2 Input/Output Actions .................................................. 58
7.2.1 Combining Actions .................................................. 60
7.2.2 Defining Interaction Functions ................................... 61
7.2.3 Showing and Reading Values ..................................... 62
7.2.4 Returning a Result From Within an Action ...................... 62
7.2.5 Conditionals and Recursion in I/O Actions ................. 63
7.2.6 Adding Up Numbers ................................................. 64
7.3 File I/O .................................................................... 64
7.4 Examples .................................................................. 65
7.4.1 Showing and Reading Values Revisited ...................... 65
7.4.2 Supermarket ............................................................ 66
7.5 Exercises ................................................................. 67

## 8 User-defined Data Types

8.1 Parametric Type Synonyms ............................................ 69
8.2 Enumeration Types ....................................................... 70
8.3 Parametrised Alternatives ............................................. 71
8.4 Recursive Data Type Definitions ................................... 73
8.4.1 Example: Defining Lists of Int ............................... 73
8.4.2 Polymorphic Lists ................................................... 74
8.5 Intermezzo: Seeing the Meaning for All the Syntax .......... 74
8.5.1 Looking Behind the Scenes ..................................... 75
8.5.2 Syntax Versus Semantics ......................................... 75
8.6 Type Classes for User-Defined Data Types .................. 76
8.6.1 Eq and Ord ............................................................. 76
8.6.2 Enum ................................................................. 76
8.6.3 Show and Read ...................................................... 77
8.7 Exercises ................................................................. 77

## 9 Formal Reasoning

9.1 Calculating with Programs .......................................... 79
9.1.1 Case Distinction .................................................... 81
9.1.2 Induction ............................................................... 82
9.1.3 Example ............................................................... 83
9.1.4 Structural Induction .............................................. 85
9.2 Working with Lemmata and Generalisation ................... 86
9.3 Program Derivation ...................................................... 89
9.4 Exercises ................................................................. 91

## 10 Modularisation & Program Decomposition

10.1 Modular Design .......................................................... 93
10.1.1 Interface Design ................................................... 95
10.1.2 Exporting and Importing in Haskell ...................... 95
10.1.3 The Supermarket Example Revisited ..................... 96
10.1.4 The Main Module .................................................. 97
10.2 Scopes and Visibility of Names .................................. 97
10.2.1 Categories of Names ............................................ 97
10.2.2 Definition Versus Use ......................................... 98
10.2.3 The Scope of Definitions ..................................... 98
10.2.4 Visibility and Modules ........................................ 100
Chapter 1

Introduction

“If thought corrupts language, language can also corrupt thought.”
—George Orwell, Politics and the English Language

The history of computing is tightly interwoven with the history of programming languages. In fact, it may be argued that the invention of the computer was a consequence of inventing programming languages. It is the programmability—the ability to alter behaviour by software expressed by means of a programming language—that distinguishes a computer from any other device. Thus, it comes hardly as a surprise that an introduction to computing begins with a discussion of programming languages.

1.1 Programming Languages

Initially, computers were programmed by strings of numbers. Indeed, it may be argued that this is still the case, as machine code—that is, the language that a computer natively understands—is but a string of numbers. As a computer program in the form of a string of numbers is rather unwieldy for a human programmer, considerable effort went into finding a more suitable representation. Clearly, a language based on the letters of the alphabet plus some common symbols is more appropriate.

However, language is not only a matter of notation, and hence, syntax; it is also a matter of meaning, or semantics. Generally, the semantics of a programming language may be close to the primitive operations of the executing machine or it may be close to the concepts of a given problem domain. This leads to the distinction of machine-oriented versus problem-oriented programming languages. The latter, which are also termed high-level languages, were initially based on the idea of sequences of commands that are a step-by-step description of the intended changes to the configuration of a computer. Such languages are called imperative languages. In contrast, languages that focus on what is to be computed rather than on the details of how this is achieved are known as declarative languages. The latter lead to more concise programs and are usually more rigorously defined. In particular, they encourage us to regard programs as mathematical objects, which is the foundation for computing as a scientific discipline; that is, the basis for formal reasoning about programs and computational processes.

From what has been said so far, it might appear that the most high-level and problem-oriented language is unconditionally the best for any application. However, there is a price to the convenience of a high-level language. As indicated before, there is only one language that a computer understands natively, namely its low-level machine language. Thus, any high-level program needs to be converted into such a low-level form before it can be executed by the machine. This conversion carries a penalty in terms of processing speed and memory consumption, which, for
some applications, may be prohibitively high. Nevertheless, whenever the desired efficiency can be achieved, a high-level language is clearly to be favoured.

When we learn to program, the gory details of executing high-level languages will obviously not be our first concern. Instead, the elementary concepts of programming need to be understood first. Consequently, a declarative high-level language that cleanly displays the foundations of computing by virtue of precisely defined semantics is to be the vehicle of choice. In the following exposition, we use the language Haskell. It is a so-called functional language, that is, a language that is based on the idea of regarding computation as a mapping from input to output values—exactly as we already know it from functions in mathematics. Moreover, the notion of types as categories, or domains of input and output values, plays a central role in modern functional languages and might indeed be regarded as one of the most fundamental concepts in the design, semantics, and implementation of programming languages. We will discuss functions and types in the remainder of this chapter and broaden these concepts in Chapter 3 with the inclusion of elementary data structures. In Chapter 4, we complete the coverage of elementary concepts by discussing the pivotal role that repetition plays in computing. Chapter 5 reinforces and generalises the material discussed so far.

Interleaved with the elementary concepts of programming, we discuss the environment in which programs are developed and executed in Chapter 2 and 6. Slightly more advanced concepts, such as input/output, user-defined data structures, and modularisation are discussed in Chapter 7, 8, and 10, respectively. We will formally reason about programs in Chapter 9, 11, and 12, where the latter two are concerned with the efficiency of programs. In particular, Chapter 11 will introduce the reasoning about essential resources, such as the processing time consumed during the execution of a program. However, instead of the machine-specific details of program execution, we will discuss fundamental properties of resource usage. Finally, Chapter 13 wraps the presentation up with a larger programming example that will convey an idea of how functional programs are executed by a computer.

### 1.2 Values, Functions and Types

Values are terms, such as 5 (an integer number), "Hello World!" (a character string), and 3.141 (a floating point number). Values are processed by functions. For example, addition + takes two numbers and produces a new number, namely the sum of the two input values; ++ takes two strings and produces a new string by concatenating the two input strings; length takes a string and produces a number, namely the length of the input string. In other words, functions, such as +, ++, and length, are mappings from input values to output values.

We can combine multiple values and functions, by using the result of a function application as input value for another function, as in

```haskell
length ("Hello " ++ "World!"")
```

The application of ++ to "Hello " and "World!" results in the string "Hello World!", which is the input value for the function length. Such a composition of values and functions is called an expression or term.

#### 1.2.1 Values Versus Types

What is the difference between 42 and "Hello World"? What does 1 + "abc" mean? Group the following values into pairs:

```text
3 True "abc" "cool" 7 False
```

Quite naturally certain values belong together, because they belong to the same category of values, such as integer numbers or strings. We call such categories of values types:
1.2 Values, Functions and Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>..., -3, -2, -1, 0, 1, 2, 3, ...</td>
</tr>
<tr>
<td>Float</td>
<td>1.0, 3.141, ...</td>
</tr>
<tr>
<td>String</td>
<td>&quot;Hello&quot;, &quot;World!&quot;, ...</td>
</tr>
<tr>
<td>Char</td>
<td>'a', 'A', 'b', 'B', ...</td>
</tr>
<tr>
<td>Bool</td>
<td>True, False</td>
</tr>
</tbody>
</table>

We write 1 :: Int or "Hello" :: String to indicate that the values 1 and "Hello" have the type Int and String, respectively. Hence, 1 :: Int can be read as “1 has type Int”.

Types essentially describe sets of values with similar properties and help us to distinguish correct from erroneous programs. For example, the expression 1 + "abc" contains a type error, because the value "abc" does not match the type of argument expected by +. We call an expression without type errors well typed. Programming languages that enforce a rigorous type discipline are often called strongly typed languages. Type errors should generally be regarded as a hint by the programming system, which tells us that our program does not make sense. Hence, they are one of the means by which the programming system helps us to write better programs.

1.2.2 Functions: Canned Computations

We have seen that, by applying functions to values, we can compute new values; but, how can we define new functions? Let us start with a simple example and write a function that increments a number by the value 1; let us call this function inc. So, the application inc 10 should produce 11, inc 11 should produce 12, and so forth—in other words, for any number x, the expression inc x, should yield x + 1. This general rule is formalised by the following function definition:

\[
\text{inc } x = x + 1
\]

A function definition comprises a head and a body separated by an equals sign. The head consists of the name of the function as well as names for the arguments to the function. In our example, there is only one argument denoted by the argument variable x. When inc is applied to an argument value, the result of the application is computed by replacing all occurrences of the variable x in the function body by the argument value:

\[
\text{inc } 2 \Rightarrow 2 + 1 \Rightarrow 3
\]

The arrow “⇒” represents a step in progressing from an expression to the value denoted by that expression. This process is called expression evaluation and corresponds to the execution of a program.

As before, we can nest expressions:

\[
\text{inc } (\text{inc } 5) \Rightarrow \text{inc } (5 + 1) \Rightarrow \text{inc } 6 \Rightarrow 6 + 1 \Rightarrow 7
\]

In our example, the choice of the name for the variable x and the function inc was arbitrary. There are, however, some syntactic restrictions for variable and function names in Haskell: the name of a function or variable

- has to start with a lower case letter or _ (underscore) and
- may only contain letters, digits, _ (underscore), or ' (apostrophe).

Moreover, when defining new functions, we have to be careful not to use a function name that does already carry a meaning, such as length. In programming languages, the names of objects, such as variables, are often called identifier.
1.2.3 Type signatures

Functions map input values to output values, for example, inc maps integers to integers, or pictorially:

\[
\begin{align*}
5 &: \text{Int} \\
\downarrow \quad \text{inc} \\
6 &: \text{Int}
\end{align*}
\]

Thus, we denote the type of inc as Int \(\rightarrow\) Int. Overall, a complete function definition appears as follows:

\[
\text{inc} :: \text{Int} \rightarrow \text{Int} \quad \text{-- type signature} \\
\text{inc} \; x \; = \; x \; + \; 1 \quad \text{-- function equation}
\]

Function signatures provide documentation for other programmers and help the Haskell system to spot type errors. In the above example, note also how Haskell allows us to annotate function definitions with comments in plain English by introducing these annotations with \(--\) (a sequence of two minus signs). Such comments are disregarded by the computer, but may help other humans reading our program to understand its purpose.

In general, two functions, such as inc and the function double,

\[
\text{double} :: \text{Int} \rightarrow \text{Int} \\
\text{double} \; x \; = \; 2 \times x
\]

may have the same type, but perform different operations. Nevertheless, like values of the same type, functions of the same type have something in common—they accept and produce values of the same kind.

An example of a function with a type different from inc and double consider

\[
\text{exclaim} :: \text{String} \rightarrow \text{String} \\
\text{exclaim} \; \text{sentence} \; = \; \text{sentence} \; \text{++} \; \text{"!"}
\]

The functions inc and double expect integers as arguments, whereas exclaim expects a string. Consequently, the expression inc "abc" is nonsensical and leads to a type error.

1.2.4 Multiple Arguments

We can compute the average value of the two floating point values 3.0 and 4.0 as follows:

\[
(3.0 + 4.0) / 2.0
\]

If we generalise this to computing the average of two numbers a and b, we get

\[
(a + b) / 2.0
\]

which we can turn into a function with two arguments as follows:

\[
\text{average} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \\
\text{average} \; \text{a} \; \text{b} \; = \; (a \; + \; b) \; / \; 2.0
\]

So, we have

\[
\text{average} \; 3.0 \; 4.0 \; \Rightarrow \; (3.0 \; + \; 4.0) \; / \; 2.0 \; \Rightarrow \; 3.5
\]

The type of a function with more than one argument separates the arguments with an arrow \((\rightarrow)\). This symmetry in notation between argument types and the result type may be somewhat surprising at first. We shall see later that there is a good reason for it—for the moment, however, let us stay with a visual analogy. We can depict a function with two arguments, such as average, as follows:
1.2 Values, Functions and Types

The function requires that two `Float` argument values be filled into the argument positions represented by the two white holes on the left. As soon as these are provided, a result value is produced:

Conceptually, we can regard the two arrows following the two `Float` arguments in the type

\[ \text{average} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \]

as representing the two arrows leading into the two argument boxes above.

### 1.2.5 A First Glance at Overloading

All serious programming languages provide some functions of which the argument types are not fixed to a single type, but instead a whole family of types is accepted. For example, both `1 + 2` (where the arguments are of type `Int`) as well as `1.5 + 1.2` (where the arguments are of type `Float`) make sense. Consequently, the function `+` simultaneously has the type

\[ (+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \]

as well as the type

\[ (+) :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \]

We call functions, such as `+`, *overloaded* functions; the name of an overloaded function carries more than just one meaning as witnessed by the multiplicity of type signatures. The motivation for permitting overloaded functions, such as `+`, is that it would be awkward to enforce the use of two different symbols—that is, two different function names—for the two cases of adding integers or adding floating-point numbers.

Unfortunately, all of this means that, given our current knowledge, we cannot denote the type of `+` in a single type signature of the form `(+) :: type`; instead, we have to resort to a family of type signatures (one for each possible type of `+`). To improve on this, we need to consider additional notation, where we exploit the fact that both argument types and the result type in one particular use of `+` are always identical. In other words, we might say that `+` has type `a -> a -> a` where `a` is either `Int` or `Float`. In fact, Haskell does not restrict `a` to only `Int` or `Float`, but instead allows any *numeric type* (most of which we have not encountered yet). We denote the set of numeric types by `Num` and generally call such sets of types *type classes*.

Using the type class `Num`, we can specify the type of `+` to be `a -> a -> a`, where `a ∈ Num`. Haskell abbreviates `a ∈ Num` to `Num a` and places it in front of the function type separated by a double arrow `=>`. Hence, the closed form of the type signature for `+` is

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

Other binary arithmetic operations, such as `-` and `*`, have the same type. Note how types, such as `Int` and `Float`, as well as type classes, such as `Num`, have names starting with an upper case letter, whereas place holders, such as `a`, have names starting with a lower case letter. This convention simplifies reading type signatures and is enforced in Haskell. We call place holders in
types, such as a above, type variables. They are important in programming languages that have
a sophisticated type system.

In addition to Num, another important type class is Eq. It contains all those types for which
the function == is defined, which checks whether its two arguments are equal. All types that we
have encountered so far, except function types, are part of Eq. So, all of the following makes
sense:

\[
\begin{align*}
2 & \equiv 2 \quad \Rightarrow \quad \text{True} \\
5.0 & \equiv 6.0 \quad \Rightarrow \quad \text{False} \\
("Hello " \text{++} "World!") & \equiv "Hello World!" \quad \Rightarrow \quad \text{True}
\end{align*}
\]

The type of == is

\[(\equiv) : \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool}\]

where \text{Bool} is the type of Boolean values \text{False} and \text{True}.

1.3 Exercises

1. What is the difference between the type Char and the type String? Do the two expressions "a" and 'a' represent the same value?

2. Given the function definition

\[
\text{square} : \text{Int} \rightarrow \text{Int} \\
\text{square } x = x * x
\]

and the previous definitions of inc and double. What is the value of

(a) inc (square 5) \\
(b) square (inc 5) \\
(c) average (inc 3) (inc 5)

3. Which of the following identifiers can be function or variable names?

- \text{square.1}
- 1square
- Square
- \text{square!}
- \text{square'}

4. The predefined function show, given a number, converts the number into a string:

\[
\text{show } 133 \quad \Rightarrow \quad "133"
\]

Define a new function, let us call it showNumber, that for example, given the number 123,
produces a string as follows:

\[
\text{showNumber } 123 \quad \Rightarrow \quad "The result is 123"
\]

Use show in the definition of the new function.
Chapter 2

Programming Environment

“The only way to learn a new programming language is by writing programs in it.”
—Brian Kernighan

Programs do not execute in a void. Instead, there is always an execution context that affects the program and may, in turn, be affected by the program. In particular, virtually any computer runs an operating system that manages resources as well as the execution of programs. As programmers, we are affected by the operating system in two ways: (1) it determines the environment in which we develop programs and (2) it determines the environment in which our programs execute. In the following, we will discuss the most elementary concepts of an operating system and then proceed to explore a Haskell programming environment. We will come back to the topic of operating systems in Chapter 6, where we shall discuss it in more detail.

2.1 The Unix Family of Operating Systems

Below, when we say Unix, we mean a family of operating systems that have a set of basic features in common, first proposed and implemented by K. Thompson and D. Ritchie. Concrete examples of Unix-like operating systems are GNU/Linux, FreeBSD, and Solaris. The fundamental features of Unix systems, some of which are discussed here, represent general mechanisms and have subsequently been replicated in other operating systems (for example, the Windows family). Unix systems are so-called multi-user and multi-tasking operating systems.

2.1.1 Multi-user Operating Systems

A multi-user operating system distinguishes users by virtue of user identities and allows multiple users to simultaneously use the same computer; some of these users may be connected to such a computer via a computer network, instead of sitting physically in front of a terminal of that machine. In a multi-user system, each user has a user name or a user id (UID), which is simply a handle by which the machine identifies that user. However, to make use of a user name, users have to prove that they have authority to operate under that name. The process of proving permission to operate under a given user name is called authentication; it usually happens during the processes of logging into a computer and typically involves entering the user name and a secret password. The access rights and storage facilities associated with a user name are called an account.

Every computer provides a distinguished account, called superuser or root, for the purpose of maintaining the system. Regular users cannot modify system data or the data of other users (without receiving specific permission to do so). In contrast, the superuser can access and manipulate all data. This distinction between regular users and the superuser serves as a mechanism
that protects against both the accidental and malicious corruption of data (for example, it protects against computer viruses).

2.1.2 Multi-tasking Operating Systems

A multi-tasking operating system can perform multiple tasks concurrently; in other words, it can simultaneously execute multiple programs. Any multi-user system naturally has to be a multi-tasking system, as different users usually execute different programs at any point in time. But even a single user often wants to execute multiple programs at the same time—for example, one might like to use an audio player at the same time as a program to develop new software. It might seem that any modern computer provides this facility, but, for example, small handheld computers might have a simpler operating system.

Generally, we call a program that is currently being executed a process. A process, however, is determined not only by the program that it is executing, but also by the data that it is processing. Multiple runs of a program may operate on different data and, indeed, two processes that execute the same program, but on different data may execute concurrently. Each process in a system is associated with a user (usually the one who directly or indirectly invoked the program)—we say, a process runs under a particular user name or UID. The data access permissions of a process correspond to those of its associated user.

2.1.3 Files and Directories

Files contain data like text, graphics, or programs and sets of files are collected in so-called directories. In addition to files, directories can also contain other directories; that is, they may be nested. Every Unix system contains a special directory called the root directory that directly or indirectly (that is, by way of nested directories) contains every file in the system. The root directory is denoted by “/”. Hence a file called foo located in the root directory is identified by “/foo”. If we assume that the root directory contains a (nested) directory bin, which in turn contains a file ping, then we can uniquely identify this file by “/bin/ping”. Such a file name, which traces the location of a file from the root directory through all directories in which it is contained is called an absolute file name. The component after the last / (slash) is called the base name (that would be ping in the previous example) and the component preceding the base name is called the path (that would be /bin/). As directories may be deeply nested, absolute file names can get quite long. Consider, for example,

```
/home/chak/lectures/cs1011/00s2/notes/week01-2.tex
```

Here, the root directory contains a directory home, that has a subdirectory called chak, which in turn contains a subdirectory lectures, and so on. Finally, the directory notes contains the file week01-2.tex. Due to the hierarchical structure of nested directories, such a file system is called a hierarchical file system.

2.2 Executing Haskell Programs

In the following, we will have a look at executing Haskell programs with the interactive shell of the Glasgow Haskell Compiler (GHC), which is called GHCi. After being invoked, GHCi displays a prompt Prelude> at which we can type commands. Upon entering an expression, GHCi executes the reduction steps needed to compute the value of the expression, and then, prints that value. For example, we have

```
Prelude> (3.0 + 4.0) / 2.0
3.5
```
2.3 Exercises

We can also load a set of function definitions into the system and evaluate expressions containing these functions. Let us assume that the file `Simpl.hs` contains the definitions for `inc`, `double`, `exclaim`, and `average` from the previous chapter. Then, we might see the following session:

```
Prelude> :l Simple.hs
Compiling simple           ( Simple.hs, interpreted )
Ok, modules loaded: Simple.
Simple> inc 2
3
Simple> exclaim "Hello"
"Hello!"
Simple> average 3.0 4.0
3.5
```

When loading function definitions and before executing any expression, the system checks the consistency of the types used in definitions and expressions. As a helpful side effect, this means that the system can also print this type information.

```
Test> :t inc
Int -> Int
Test> :t inc 2
Int
```

Even more importantly, the system complains if it finds that types are being used inconsistently.

```
Main> inc "abc"
<interactive>:0:
  Couldn't match `Int' against `[Char]'
  Expected type: Int
  Inferred type: [Char]
  In the first argument of `inc', namely "abc",
  in the definition of function `it': inc "abc"
Main>
```

Type errors are sometimes regarded as a nuisance, while in fact they are very helpful. If the system were quiet and would just go on and execute `inc "abc"`, the system would terminate abnormally or we would get some bogus result. In large programs, it is much easier to find a bug when told the precise position of a type error, instead of tediously having to track down the cause of a wrong result.

### 2.3 Exercises

1. Consult your local system manual to find out how to invoke a command shell and how to use the elementary file and directory operations.

2. Consult your local system manual to find out how to create and edit a text file. Create a file called `Simple.hs`, enter `module Simple where` in the first line, followed by the definitions of the functions `inc`, `double`, `exclaim`, and `average` from the previous chapter.

3. Load `Simple.hs` into GHCi and use the functions as described above.
Chapter 3

Basic Control Structures and Types

“A complex system that works is invariably found to have evolved from a simple system that works.”
—Unknown

The functions that we defined so far have been restricted to elementary operations, such as incrementing a given number. In this chapter, we will discuss some slightly more advanced functions and survey elementary operations on lists.

3.1 Programs are Composed from Modules

Usually, a program consists of a large number of function and type definitions. Obviously, putting them all into one single file is a bad idea. Modern programming languages therefore provide some means to structure the program by allowing to group related definitions into logical units which are stored in separate files. In Haskell, these units are called modules.

Let us have a look at the following definition of a Haskell module, called Simple:

```haskell
-- Example module
-- Manuel M. T. Chakravarty, July 2000
--
-- This is a simple example for a module definition

module Sample
where

-- yield the square of a given number
--
square :: Int -> Int
square x = x * x

-- checks whether all three given numbers are equal
--
threeEqual :: Int -> Int -> Int -> Bool
threeEqual a b c = (a == b) && (b == c)
```
The module starts with the header: a comment that contains a one line description of the module, the
author and date of creation, and briefly describes the purpose of the module.¹

The first line of code starts with the keyword module, followed by the name of the module,
the keyword where, and the definitions that belong to the module. Note that a module name, in
contrast to function or variable names has to start with an uppercase letter.

In Haskell, there is a special module called Prelude.hs that GHCi automatically loads on
startup. The module Prelude contains all the functions that are pre-defined in Haskell, such as
+, length, and so on.

For now, as we are starting with simple, short programs, we will write all the function def-
initions of one program into a single module. Later we will learn how we can structure more
complex programs using modules. In fact, this is a central topic in software development, and
modules play an essential role in structuring large software systems; a topic to which we will
return later.

3.2 Branches in the Control Flow

So far, all our programs have unconditionally performed the same computation.

3.2.1 Choices are Implemented Through Conditionals

What should we do if we want to implement a function

\[
\text{max} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]

which returns the greater of its two arguments, i.e., the expressions max 5 2 and max 2 5 both
evaluate to 5, max 1 7 to 7, and so on. For two arbitrary numbers x and y, we want max x y to
return x if x \(\geq\) y, otherwise it should return y.

This can be expressed in Haskell by a so-called conditional or if-then-else expression:

\[
\text{if} \langle \text{condition} \rangle \text{ then } \langle \text{value if true} \rangle \text{ else } \langle \text{value if false} \rangle
\]

Now, we can implement max:

\[
\text{max} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}
\]

\[
\text{max} \ x \ y = \text{if} \ x \geq y \text{ then } x \text{ else } y
\]

Let’s look at the evaluation of max 5 2:

\[
\text{max} \ 5 \ 2 \Rightarrow \text{if} \ 5 \geq 2 \text{ then } 5 \text{ else } 2 \Rightarrow \text{if} \ \text{True} \text{ then } 5 \text{ else } 2 \Rightarrow 5
\]

Conditionals are an essential component of programming, because they allow us to choose to
perform different computations depending on the values of the inputs. Here is another example:

\[
\text{signum} :: \text{Int} \rightarrow \text{Int}
\]

\[
\text{signum} \ x = \text{if} \ x < 0 \text{ then } -1 \text{ else if } x = 0 \text{ then } 0 \text{ else } 1
\]

3.2.2 Guards

Cascading conditional expressions—as in the previous definition of signum—are difficult to
read; therefore, some programming languages provide guarded expressions as an alternate syn-
tax:

¹The header is optional in that the compiler will not raise an error if it is missing, but your tutor will!
3.3 Binders

A binder binds a value to a name. The value can subsequently be referred to by that name. For example,

\[
\text{pi} = 3.141592653589793
\]

allows us to just write `pi` instead of spelling out `3.141592653589793`.

3.3.1 How to Associate Names with Values

We may use a newly introduced name in another function definition:

\[
\begin{align*}
\text{pi} & : \text{Float} \\
\text{pi} & = 3.141592653589793
\end{align*}
\]

\[
\begin{align*}
\text{circleArea} & : \text{Float} \to \text{Float} \\
\text{circleArea} \text{ radius} & = \text{pi} \ast \text{radius} \ast \text{radius}
\end{align*}
\]

Sometimes, we need to introduce a new name, which will only be used within a function. In that case, we should use a local binding. For example,

\[
\begin{align*}
\text{pi} & : \text{Float} \\
\text{pi} & = 3.141592653589793
\end{align*}
\]

\[
\begin{align*}
\text{circleArea'} & : \text{Float} \to \text{Float} \\
\text{circleArea'} \text{ diameter} & = \text{pi} \ast \text{radius} \ast \text{radius} \\
& \quad \text{where} \\
& \quad \text{radius} = \text{diameter} / 2.0
\end{align*}
\]

The evaluation of this function proceeds as follows:

\[
\begin{align*}
\text{circleArea'} \ 6.0 & \Rightarrow \text{pi} \ast \text{radius} \ast \text{radius} \text{ where radius} = 6.0 / 2.0 \\
& \Rightarrow \text{pi} \ast \text{radius} \ast \text{radius} \text{ where radius} = 3.0 \\
& \Rightarrow \text{pi} \ast 3.0 \ast 3.0 \\
& \Rightarrow \text{pi} \ast 9.0 \\
& \Rightarrow 3.141592653589793 \ast 9.0 \\
& \Rightarrow 28.2743
\end{align*}
\]
3.4 Tuples: Combining Different Data Items

So far, we have seen how to pass multiple values to a function, but not how a function can return more than one result value. We can achieve this by using *tuples*:

\[
\text{addMul} :: \text{Int} \rightarrow \text{Int} \rightarrow (\text{Int}, \text{Int}) \\
\text{addMul} \ x \ y = (x + y, x \times y)
\]

A tuple combines multiple components (two integer values, in the above example) into one. The compound value can be manipulated as a single entity and, in particular, be returned as a value from a function.

However, the *construction* of a compound value is only half of the story. We also need a method for *decomposing* such values. We achieve this by using a notation dual to that of tuple construction:

\[
\text{fstFromIntPair} :: (\text{Int}, \text{Int}) \rightarrow \text{Int} \\
\text{fstFromIntPair} \ (x, y) = x \\
\text{sndFromIntPair} :: (\text{Int}, \text{Int}) \rightarrow \text{Int} \\
\text{sndFromIntPair} \ (x, y) = y
\]

In the argument of \text{fstFromIntPair}, we do not use a variable to refer to the compound argument as a whole. Instead, we decompose the pair into its components \(x\) and \(y\). The combined use of \text{addMul} and \text{fstFromIntPair} behaves as follows:

\[
\text{fstFromIntPair} \ \text{addMul} \ 5 \ 6 \\
\Rightarrow \text{fstFromIntPair} \ (5 + 6, 5 \times 6) \\
\Rightarrow \text{fstFromIntPair} \ (11, 30) \\
\Rightarrow 11
\]

3.4.1 Example: Points

Tuples are not just useful to return multiple results, but also for the representation of data items that cannot be modeled by one primitive value alone. A useful example is given by the points in a 2-dimensional Cartesian coordinate system, which can be represented by a pair of integer values. To avoid having to write the less informative \((\text{Int}, \text{Int})\) whenever we denote the type of a point, we can introduce a new type name—similar to the introduction of names for repeatedly used values, which we discussed earlier:

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

With this definition, we define some simple operations on points:

\[
\text{-- origin of the co-ordinate system} \\
\text{--} \\
\text{origin :: Point} \\
\text{origin} = (0, 0)
\]

\[
\text{-- move a given point to the right} \\
\text{--} \\
\text{moveRight :: Point} \rightarrow \text{Int} \rightarrow \text{Point} \\
\text{moveRight} \ (x, y) \ \text{distance} = (x + \text{distance}, y)
\]

\[
\text{-- move a given point to upwards} \\
\text{--} \\
\text{moveUp :: Point} \rightarrow \text{Int} \rightarrow \text{Point} \\
\text{moveUp} \ (x, y) \ \text{distance} = (x, y + \text{distance})
\]
3.4.2 Example: Colour Points

When we extend points to include a colour, another important property of tuples becomes obvious: tuple components may be of different types. Hence, if we denote colour values with a textual (string) representation, we have

```haskell
-- we represent colours by strings
--
type Colour = String

-- new name for the type of colour points
--
type ColourPoint = (Int, Int, Colour)
```

which allows the following operations on colour points:

```haskell
-- origin of the co-ordinate system in a given colour
--
origin :: Colour -> ColourPoint
origin colour = (0, 0, colour)

-- move a colour point vertically and horizontally
--
move :: ColourPoint -> Int -> Int -> ColourPoint
move (x, y, colour) xDistance yDistance =
  (x + xDistance, y + yDistance, colour)

-- compute the distance between two colour points
--
distance :: ColourPoint -> ColourPoint -> Float
distance (x1, y1, colour1) (x2, y2, colour2) =
  sqrt (intToFloat (dx * dx + dy * dy))
  where
    dx = x2 - x1
    dy = y2 - y1
```

Note how we use a `where` clause in the last definition to avoid repeating the expressions `x2 - x1` and `y2 - y1`. The auxiliary function `intToFloat` is defined as

```haskell
intToFloat :: Int -> Float
intToFloat x = fromInteger (toInteger x)
```

The two functions `fromInteger` and `toInteger` are pre-defined Prelude functions that facilitate conversions between different encodings of numbers.

3.4.3 Important Symmetries in Haskell

If we compare the syntax of values and types of tuples, we see that they correspond. For example, consider

```haskell
(10, 15, "green") :: (Int, Int, String)
```

If we replace the values 10, 15, and "green" with their respective types `Int`, `Int`, and `String`, we obtain the type of the tuple. Moreover, we have a correspondence between *term construction* and *term decomposition* (also called ). Consider,
Basic Control Structures and Types

startPoint = (0, 0, "black")

colourOfPoint (x, y, colour) = colour

If we replace the components in the tuple construction (0, 0, and "black") by variable names (in this case x, y, colour), we arrive at the pattern that can be used to decompose the corresponding tuple.

3.4.4 Special Names for Some Tuples

The following table lists a number of tuple types and their names:

<table>
<thead>
<tr>
<th>#</th>
<th>Expression</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>()</td>
<td>Unit</td>
</tr>
<tr>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>2</td>
<td>(x₁, x₂)</td>
<td>Pair</td>
</tr>
<tr>
<td>3</td>
<td>(x₁, x₂, x₃)</td>
<td>Triple</td>
</tr>
<tr>
<td>4</td>
<td>(x₁, x₂, x₃, x₄)</td>
<td>Quadruple</td>
</tr>
<tr>
<td>5</td>
<td>(x₁, x₂, x₃, x₄, x₅)</td>
<td>Quintuple</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>(x₁, ..., xₙ)</td>
<td>n-tuple</td>
</tr>
</tbody>
</table>

3.5 Lists: Many Values of a Single Type

Tuples provide the ability to bring together a fixed number of values of varying type. However, many applications, in addition, require the ability to manipulate compound types, which may contain a varying number of elements of a single type. This is what lists are for.

oddNumbers :: Int -> [Int]
oddNumbers maxNumber = [1, 3..maxNumber]

The number of values returned by this function depends on the argument, unlike with the earlier addMul, where only the value, but not the number of values, depended on the input. For example, we have

oddNumbers 10 ⇒ [1, 3, 5, 7, 9]
oddNumbers 15 ⇒ [1, 3, 5, 7, 9, 11, 13, 15]

The difference between tuples and lists can be seen by comparing their types, as in

(1, 2, "green") :: (Int, Int, String)

and

[1, 2, 3, 4] :: [Int]

The number of components is explicit in the type of a tuple, but not in the type of a list. As a consequence, the elements of tuples may be heterogeneous, whereas those of lists must be homogeneous.

3.5.1 Useful List Functions

- Values enclosed in square brackets:

  [4, 2, 6, 7, 2]
  ["red", "green", "blue"]
  [x, y, z]
3.5 Lists: Many Values of a Single Type

- Given a list, we can add another element in front:

  "yellow" : ["red", "green", "blue"]
  ⇒ ["yellow", "red", "green", "blue"]

  The operator (:) (pronounced cons) can only add elements at the front. So,

  ["red", "green", "blue"] : "yellow"
  ⇒ Error!

- Or we can join two lists together using (++):

  [4, 2, 3] ++ [3, 1, 2, 7]
  ⇒ [4, 2, 3, 3, 1, 2, 7]

- Get any element out of a list: !!

  [0, 1, 2, 3] !! 2 ⇒ 2

  (Index count starts at 0!)

- Split a list into its first element and the rest: head & tail

  head [0, 1, 2, 3] ⇒ 0
  tail [0, 1, 2, 3] ⇒ [1, 2, 3]

- Length of a list: length

  length [0, 1, 2, 3] ⇒ 4

- Add up or multiply the elements of a list: sum & product

  sum [0, 1, 2, 3] ⇒ 6
  product [1, 2, 3, 4] ⇒ 24

Given these functions, how can we add "yellow" at the end of ["red", "green", "blue"]? We can’t use : (cons). We need to use ++ as follows:

  ["red", "green", "blue"] ++ ["yellow"]

Note the list constructing brackets around "yellow". In effect, we wrap "yellow" into a singleton list, which we then append to the list ["red", "green", "blue"].

3.5.2 Lists Versus Tuples

As lists and tuples are often confused, let us summarise the differences between them. Tuples have the following properties:

- Fixed size, i.e., fixed number of components:
  (1, 2) :: (Int, Int) and (1, 2, 3) :: (Int, Int, Int) have different types.

- Components may be of different type:
  (5, "Hello") makes perfect sense.

In contrast, the following are the properties of lists:

- Variable size, i.e., number of components may vary:
  [1, 2] :: [Int] and [1, 2, 3] :: [Int] have the same types.

- Components must be of the same type:
  [5, "Hello"] is illegal!
3.5.3 Strings as Lists

Strings are in fact a particular form of lists in Haskell, which are defined as

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

This means, in particular, that list operations work on strings.

"Hello" !! 1 ⇒ 'e'

In fact, in Haskell, "Hello" is considered to be the same as ['H', 'e', 'l', 'l', 'o']. This is very convenient, as we will see that there are many powerful list processing operations and these are directly available for string manipulation.

3.6 Layout

Unlike in many other programming languages, formatting matters to the interpreter or compiler in Haskell. In other words, indentation and the use of new lines impose a number of constraints. This allows the language to do away with some of the noise that is introduced by some other languages to disambiguate the input.

Compare

\[
\text{foo } x = a + b \\
\text{where} \\
a = 1 + x \\
b = 2
\]

to

\[
\text{foo } x = a + b \\
\text{where} \\
a = 1 + x \\
b = 2
\]

Both are legal programs. However, in the first one, the definition of \( b \) is local to foo whereas in the second program, the use of \( b \) is not restricted to foo.

An example of proper layout is the function distance that we discussed earlier:

\[
\text{distance :: ColourPoint } \rightarrow \text{ ColourPoint } \rightarrow \text{ Float} \\
\text{distance } (x1, y1, \text{ colour1}) \ (x2, y2, \text{ colour2}) = \\
\text{sqrt} \ (\text{fromInt} \ (dx * dx + dy * dy)) \\
\text{where} \\
dx = x2 - x1 \\
dy = y2 - y1
\]

There are three layout rules that you you have to follow to get syntactically correct programs:

1. All program code that belongs to a function definition has to be further right than the first character of that definition (i.e., the first character of the function name). In the case of distance, all code has to be further right than the column in which the character \( d \) of the function name distance is.

2. Similarly, all code of a local definition in a where clause must be further right than the first character of the name of the defined variable.

3. All definitions within a where clause must be aligned—e.g., above the definitions of \( dx \) and \( dy \) start in the same column.
3.7 Exercises

1. Write a function `sort2 :: Int -> Int -> (Int, Int)` which accepts two Int values as arguments and returns them as sorted pair, so that `sort2 5 3` is equal to `(3,5)`. How can you define the function using a conditional, how can you solve it using guards?

2. Consider a function `almostEqual :: (Int, Int) -> (Int, Int) -> Bool` which compares the values of two pairs of Int. It returns True if both pairs contain the same values, regardless of the order. For example, `almostEqual (3,4) (4,3)` is True, but `almostEqual (3,4) (3,5)` is False. Which of the following definitions return the correct value? Which of the definitions would you consider good style? Why? Add comments to the correct definitions to make it easier to improve readability. `(&&) :: Bool -> Bool -> Bool` is logical "and", `(||) :: Bool -> Bool -> Bool` is logical 'or', and `(==)` tests if two values are equal).

```haskell
almostEqual (x1, y1) (x2, y2)
| (x1 == x2) && (y1 == y2) = True
| (x1 == y2) && (y1 == x2) = True
| otherwise = False

almostEqual (x1, y1) (x2, y2)
| (x1 == x2) = (y1 == y2)
| (x1 == y2) = (y1 == x2)
| otherwise = False

almostEqual pair1 pair2 =
  (pair1 == pair2) || (swap pair1 == pair2)
  where swap (x,y) = (y,x)

almostEqual pair1 pair2 =
  (pair1 == pair2) || (swap pair1 == swap pair2)
  where swap (x,y) = (y,x)

almostEqual (x1, y1) (x2, y2) =
  if (x1 == x2) then
    if (y1 == y2)
      then True
      else False
    else False
  else if (x1 == y2)
    then if (x2 == y1)
      then True
      else False
    else False
  else False
```
Chapter 4

Recursion

“To iterate is human, to recurse, divine.”
—Robert Heller

Repetition is a crucial ingredient to any non-trivial program.

4.1 Recursion over Numbers

The functions that we considered so far needed only a fixed number of operations. Even
\[
\text{distance} \ (x_1, y_1, \text{colour1}) \ (x_2, y_2, \text{colour2}) = \\
\sqrt{\text{intToFloat} \ (dx \ * \ dx + dy \ * \ dy)}
\]
where
\[
dx = x_2 - x_1 \\
dy = y_2 - y_1
\]
needs exactly one addition, two subtractions, two multiplications, the \text{intToFloat}, and one square root—which makes seven operations. If conditional expressions or guards are used, the number of operations may vary, but we could still place an upper limit on number of operations (independently of the input given to the function).

This is, however, in general not possible. Some functions can have an arbitrarily high number of operations, depending on the input. Consider the two functions \text{natSum} and \text{product}, which compute the sum of the natural numbers up to a limit and the product of all of the elements of an integer list, respectively:
\[
\text{natSum} \ n = 0 + 1 + \cdots + n \\
\text{product} \ \ [x_1, x_2, \ldots, x_n] = 1 \ * \ x_1 \ * \ x_2 \ * \ldots \ * \ x_n
\]

The above are not actual function definitions, since the notation “\cdots” is not valid Haskell.\(^1\) However, they illustrate the meaning of the two functions in reasonably formal terms. From this specification of the meaning, we see that both functions require \(n\) operations to compute their result. Thus, we can make two important observations:

1. The number of operations depends on the input.
2. A certain computation (or more generally, a set of operations) is repeatedly used.

It is this input-dependent repetition, which we will implement by a programming technique known as recursion.

\(^1\)Do not confuse this with the notation \([x \ldots y]\) to construct lists with a fixed increment. It looks somewhat similar, but is very limited in its functionality.
4.1.1 Computing $0 + 1 + \cdots + n$

Let us start with the simplest case: recursion over the natural numbers. How can we define the function \texttt{natSum :: Int \rightarrow Int}, which sums up all natural numbers from zero up to a given number \texttt{n}? It should behave as

$$\texttt{natSum n = 0 + 1 + \cdots + n}$$

but how can we substitute the ellipsis by working program code?

To get an idea of what we would like to happen, consider the following rules describing the computations to be performed by \texttt{natSum} for the input value 5:

\begin{align*}
\texttt{natSum 0} & = 0 \\
\texttt{natSum 1} & = 0 + 1 \\
\texttt{natSum 2} & = 0 + 1 + 2 \\
\texttt{natSum 3} & = 0 + 1 + 2 + 3 \\
\texttt{natSum 4} & = 0 + 1 + 2 + 3 + 4 \\
\texttt{natSum 5} & = 0 + 1 + 2 + 3 + 4 + 5 \\
& \cdots
\end{align*}

The above are legal Haskell definitions, but obviously, we would need an unbounded number of definitions to define \texttt{natSum} for every possible input. There is, however, an observation that comes to our rescue: Each of the above equations contains computations that already appear in the previous equations. For example, for \texttt{natSum 5}, we have to evaluate $0 + 1 + 2 + 3 + 4 + 5$, but the subcomputation $0 + 1 + 2 + 3 + 4$ already appears in \texttt{natSum 4}. This seems like good news, as it allows us to \textit{reuse} earlier equations for later ones. In other words, we could redefine \texttt{natSum 5} as

$$\texttt{natSum 5 = natSum 4 + 5}$$

In fact, except for the first equation, we can systematically reuse the immediately preceding equation:

\begin{align*}
\texttt{natSum 0} & = 0 \\
\texttt{natSum 1} & = \texttt{natSum 0} + 1 \\
\texttt{natSum 2} & = \texttt{natSum 1} + 2 \\
\texttt{natSum 3} & = \texttt{natSum 2} + 3 \\
\texttt{natSum 4} & = \texttt{natSum 3} + 4 \\
\texttt{natSum 5} & = \texttt{natSum 4} + 5 \\
& \cdots
\end{align*}

Interestingly, all equations—except the first one—now look exactly the same; they just use different values. This seems like an ideal situation to once more apply the trick that we already used when we introduced function definitions, i.e., we use \textit{abstraction} to replace concrete values by variables and, in this way, we distill a repeating pattern out of the above equations. The repeating pattern is

$$\texttt{natSum n = natSum m + n}$$

where we know that \texttt{m} always equals \texttt{n - 1}. In other words, given the two rules

\begin{align*}
\texttt{natSum 0} & = 0 \\
\texttt{natSum n} & = \texttt{natSum (n - 1)} + n
\end{align*}

we seem to have captured the essence of \texttt{natSum}. In natural language, we could describe this essence as follows:
The sum of the natural numbers from 0 to 0 is 0 (first case). The sum of the natural numbers from 0 to \( n \) can be obtained by computing the sum of the natural numbers from 0 to \( n - 1 \), and then, adding \( n \) (second case).

This sounds very sensible, but is it enough for a computer to actually compute the result of an application of \( \text{natSum} \)?

To verify this, let us perform the stepwise evaluation of an application of \( \text{natSum} \):

\[
\begin{align*}
\text{natSum} \ 5 & \Rightarrow \text{natSum} \ (5 - 1) + 5 \\
& \Rightarrow (\text{natSum} \ 4) + 5 \\
& \Rightarrow (\text{natSum} \ (4 - 1) + 4) + 5 \\
& \Rightarrow (((\text{natSum} \ 3) + 4) + 5 \\
& \Rightarrow (((\text{natSum} \ (3 - 1) + 3) + 4) + 5 \\
& \Rightarrow (((\text{natSum} \ 2) + 3) + 4) + 5 \\
& \Rightarrow (((\text{natSum} \ (2 - 1) + 2) + 3) + 4) + 5 \\
& \Rightarrow (((\text{natSum} \ (1 - 1) + 1) + 2) + 3) + 4) + 5 \\
& \Rightarrow (((\text{natSum} \ 0) + 1) + 2) + 3) + 4) + 5 \\
& \Rightarrow (((0 + 1) + 2) + 3) + 4) + 5 \\
& \Rightarrow 15
\end{align*}
\]

This obviously works the way we intended it to work. The above definition of \( \text{natSum} \) is called recursive, because \( \text{natSum} \) itself is used in the definition of \( \text{natSum} \)—i.e., a recursive function is a function that makes use of itself in its definition.

Recursive definitions have at least have two cases: the base case and the stepping case (or recursive case). The base case specifies what to do in the simplest case of input (where the computation stops), whereas the stepping case includes the recursive definition:

\[
\begin{align*}
\text{natSum} & :: \text{Int} \to \text{Int} \\
\text{natSum} \ 0 & = 0 \quad \text{-- base case} \\
\text{natSum} \ n & = \text{natSum} \ (n - 1) + n \quad \text{-- stepping case}
\end{align*}
\]

Later we will encounter recursive functions with more than one base case and/or more than one stepping case—but there will always be at least one of each kind.

An alternative way of writing the recursive definition of \( \text{natSum} \) would be

\[
\begin{align*}
\text{natSum} & :: \text{Int} \to \text{Int} \\
\text{natSum} \ n & = \text{if} \ n = 0 \\
& \quad \text{then} \ 0 \\
& \quad \text{else} \ \text{natSum} \ (n - 1) + n
\end{align*}
\]

It contains only one equation and makes the case distinction explicit.

### 4.1.2 Repeating Characters

Another example of a recursive function is the function \( \text{repeatChar} \), which produces a string that contains a given character \( n \) times, i.e., has type

\[
\text{repeatChar} :: \text{Int} \to \text{Char} \to \text{String}
\]

To find a suitable definition for the function, we first consider what an appropriate base case might look like. Let us assume that we want the function to work for zero repetitions. Then, the
expression \(\text{repeatChar } 0 \ 'x'\) would have to return an empty string, which fully specifies the base case.

Next, we have to find a suitable stepping case. If we already had a definition that works for \(\text{repeatChar } (n - 1) \ 'x'\), how would we extend the result of \(\text{repeatChar } (n - 1) \ 'x'\) such that it gives us the result for \(\text{repeatChar } n \ 'x'\)? Another way to ask the same question is, which operation is it that should be repeated in the recursive definition? The answer is using the colon operator to add another copy of the replicated character to the result string. So, overall, we get

\[
\text{repeatChar} :: \text{Int} \to \text{Char} \to \text{String}
\]
\[
\text{repeatChar } 0 \ \text{char} = ""
\]
\[
\text{repeatChar } n \ \text{char} = \text{char} : \text{repeatChar } (n - 1) \ \text{char}
\]

Instead of the colon operator, we could also use ++.

### 4.2 Partial Functions

A question that might come up during the definition of `natSum` and `repeatChar` is, what happens if we call these functions with arguments that make the stepping case to move away from, rather than towards the base case. For example, what does `natSum (-1)` result in? As the stepping case is applicable, the argument will be reduced to \(-2\), \(-3\), and so on, which means that we enter an infinite computation. In other words, `natSum` is not defined for arguments smaller than 0.

Generally, functions that are only defined on a subset of the possible argument values are called *partial functions*, while functions that are defined on all possible input values are called *total functions*. Another example of a partial function is `head`, which extracts the first element of a list. When called with an empty list, no meaningful result can be returned.

The comment preceding the definition of a partial function should always include a statement specifying the range of admissible input values (and what happens if the function is wrongly called with an argument that is outside of this range). Moreover, it is considered good style if partial functions raise a suitable error message when called with an input value on which they are undefined. To this end, Haskell provides the function `error`, which takes a string and raises an runtime error displaying the given string as an error message. We could modify `natSum` as follows to raise a proper error message in case of a wrong argument:

\[
natSum :: \text{Int} \to \text{Int}
natSum 0 = 0
natSum n | n > 1 = natSum (n - 1) + n
| otherwise = error "natSum: Input value too small!"
\]

Note how the name of the function is included into the error message. Another sign of good style.

### 4.3 Recursion and Lists

#### 4.3.1 List Construction

Given the material that we covered so far, it should be relatively straightforward to write a function that, when given a string, produces a list containing all of the possible suffixes of that string. For example, we would have

\[
suffixes "Hello" \Rightarrow ["Hello", "ello", "llo", "lo", "o"]
\]

The base case is when we have an empty string; then, we have no suffix, so we return the empty list.
4.3 Recursion and Lists

suffixes "" = []

On the other hand, given

suffixes "ello" ⇒ ["ello", "llo", "lo", "o"]

we only need to add the string "Hello" at the front of the result to get the value of suffixes "Hello". Moreover, as tail "Hello" = "ello", we arrive at the following definition:

suffixes :: String -> [String]
suffixes "" = []
suffixes str = str : suffixes (tail str)

In other words, after adding the current string str, we only need the suffixes of tail str.

Note that we can build lists recursively using only the empty list [] and the list forming operator (:). As these two suffice to build any list, they are regarded as the basic list forming constructors. In fact, they both—historically—carry the names nil and cons, respectively.\(^2\)

4.3.2 Lists as Recursive Structures

From now on, we will suppose all lists to be constructed from nil and cons, where the following equality illustrates the correspondence between the square bracket and the nil/cons representation.

\[[x_1, x_2, x_3, \ldots, x_n]\] = \(x_1 : (x_2 : (x_3 : \cdots : (x_n : [ ]))\))

Due to the repeated occurrence of cons, the right hand side exposes the recursive structure of lists. For each element \(x_i\) in a list, we have one cons operator including this element into the list. Finally, each list is terminated by nil. This representation not only makes the recursive nature of lists explicit, but it is, in fact, the original representation of lists. The closed \([x_1, x_2, x_2, \ldots, x_n]\) notation is only a convenient shorthand.

4.3.3 Pattern Matching

The nil and cons operators are so elementary that we use them not only to construct lists, but also to decompose them. Earlier, we used pattern matching to decompose tuples:

\(\text{fst} :: (a, b) \rightarrow a\)
\(\text{fst} \ (x, y) = x\)

In a similar way, we can use pattern matching to decompose lists into their first element and the rest, i.e., into the two components joined together by cons. In fact, this is exactly what the two functions head and tail do to extract the first element and the remaining elements from a list:

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

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

In other words, they yield the two values that are used to compose a list using the cons operator. Thus, for every non-empty list \(xs\), we have the equality:

\(xs = \text{head} \ xs : \text{tail} \ xs\)

\(^2\)The word "nil" actually stands for "Not In List" and "cons" is an abbreviation for "(list) constructor."
Therefore, the first component passed to cons is often called the head of the new list and the second component the tail. In the above implementation, the partial nature of the two functions is obvious. There is no case that covers an input of the empty list (nil).

If we want to define a total function over lists with pattern matching, we have to specify two cases, one for the case where the input is an empty list and a second for the case where it is not empty, i.e., can be regarded as being constructed by a cons operator. The following function, which checks whether a given list is empty, covers both cases:

```haskell
null :: [a] -> Bool
null [] = True
null (x:xs) = False
```

### 4.3.4 List Traversal

Now—finally—we are in the position to return to the initial question of defining a function `product` that multiplies all of the elements in a given list of integer values.

\[
product \ [x_1, x_2, \ldots, x_n] = 1 \times x_1 \times x_2 \times \ldots \times x_n
\]

Using our knowledge about the recursive nature of lists, we can rephrase the tasks as defining a function `product` computing

\[
product \ (x_1 : (x_2 : \ldots : (x_n : [ ]))) = 1 \times x_1 \times x_2 \times \ldots \times x_n
\]

By fixing a suitable order for the calculation of the products on the right hand side and adding the neutral multiplication of a 1, we can get the right hand side exactly into the same format as the recursive list representation:

\[
product \ (x_1 : (x_2 : \ldots : (x_n : [ ]))) = 1 \times (x_1 \times (x_2 \times \ldots \times (x_n-1 \times x_n)))
\]

Each cons operator corresponds to a multiplication and the final nil corresponds to the 1.

Thus, we can use a recursive function definition following the recursion of the data structure. As the base case, we have the case where `product` is passed an empty list, i.e., nil. From the above specification of `product`, it is clear that we have to return 1 in this case:

\[
product \ [ ] = 1
\]

The recursive case is that of a non-empty list, where cons allows us to decompose the list into a head and a tail. The head is a list element that we multiply by the result of recursively applying `product` to the tail. So, overall, we have

```haskell
product :: [Int] -> Int
product [] = 1
product (x:xs) = x * product xs
```

This definition clearly does what we said before. It replaces nil by 1 and cons by multiplication. Functions like `product` that take a list and reduce it to an elementary value (like an integer) are called reductions.

### 4.3.5 Inside of the List Traversal

To see how a list traversal proceeds, let us look at the stepwise evaluation of `product \ [3, 5, 6]`, which we write as `product \ (3 : (5 : (6 : [ ])))` to make the recursive structure of the list
explicit:

\[
\text{product } (3:(5:(6:[]))) \Rightarrow 3 \times \text{product } (5:(6:[])) \\
\Rightarrow 3 \times 5 \times \text{product } (6[]): \\
\Rightarrow 3 \times 5 \times 6 \times \text{product } [] \\
\Rightarrow 3 \times 5 \times 6 \times 1 \\
\Rightarrow 90
\]

4.3.6 Concatenating Lists

Lists can be nested, i.e., we can have lists of lists of integer numbers, such as

\[
[[5, 6, 2], [], [4, 2]]
\]

The function `concat` can be used to string all the sublists into one bigger list, i.e.,

\[
\text{concat } [[5, 6, 2], [], [4, 2]] \Rightarrow [5, 6, 2, 4, 2]
\]

As we can join two lists with the `++` operator, the obvious attempt is to compute

\[
\text{concat } [xs_1, xs_2, \ldots, xs_n] = xs_1 \mathbin{++} xs_2 \mathbin{++} \cdots \mathbin{++} xs_n
\]

From this specification, it looks like a recursive definition of `concat` would have a structure similar to that of `product`. In fact, we can define it as

\[
\text{concat} :: [[a]] \rightarrow [a] \\
\text{concat } [] = [] \\
\text{concat } (xs:xss) = xs \mathbin{++} \text{concat } xss
\]

Using recursion, we can now also define `++` in terms of pattern matching and recursion. There are, however, two points that make this definition different from the list traversal that we have considered so far: (1) the operator `++` is an infix function and (2) `++` does not just consume a list, but simultaneously produces one.

Concerning the first point, infix operators are written between their two arguments. In the type signature (where there are no arguments present), the operator must be enclosed in parentheses.

\[
(\mathbin{++}) :: [a] \rightarrow [a] \rightarrow [a]
\]

To understand the second point—that `++` also produces a list—let us have a look at an example application: \([5, 6, 2] \mathbin{++} [4, 2]\). To see what must happen, let us rewrite the example in cons-nil form:

\[
(5:(2:[])) \mathbin{++} (4:(2:[])) \Rightarrow 5:(6:(2:(4:(2:[]))))
\]

In this form, it becomes apparent that what `++` essentially has to do, is to replace the nil (\([\ ]\)) of the first list with the whole second list. As we cannot just pick the end of a list and replace it with some other value (the cons operator matches only the front of the list), we must traverse the list from start to end. In this process, we just copy all of the elements of the first list into the resulting list.

We do pattern matching on the first argument. In the base case, where the left argument is the empty list, we can just return the other argument. As usual, the tricky case is the stepping case. As previously, it helps to consider the result of applying the same function to an input list that is one element smaller:

\[
(6:(2:[])) \mathbin{++} (4:(2:[])) \Rightarrow 6:(2:(4:(2:[])))
\]
Obviously, this can be extended to the result of computing \((5:(6:(2:[ ])))\) ++ \((4:(2:[ ])))\) by just adding the element 5 in front of the result. In other words, the stepping case of ++ has to add the front element of its first argument to the front of the result (this is, \(x\) in the following definition).

\[
(\text{++}) \quad : \quad [a] \rightarrow [a] \rightarrow [a] \\
[ ] \quad \text{++} \quad y\!s \quad = \quad y\!s \\
(x:xs) \quad \text{++} \quad y\!s \quad = \quad x : (xs \text{++} y\!s)
\]

Given this definition, let us see how it executes:

\[
(5:(6:(2:[ ]))) \quad \text{++} \quad (4:(2:[ ])) \Rightarrow 5:((6:(2:[ ])) \quad \text{++} \quad (4:(2:[ ])))
\]

\[
\Rightarrow 5:(6:((2:[ ]) \quad \text{++} \quad (4:(2:[ ]))))
\]

\[
\Rightarrow 5:(6:(2:(2:[ ])) \quad \text{++} \quad (4:(2:[ ]))))
\]

\[
\Rightarrow 5:(6:(2:(4:(2:[ ]))))
\]

### 4.4 Polymorphism and Overloading

Although we have only used functions such as head, tail, null, concat and ++ on lists containing integers, there is in fact nothing within the implementation of these functions that actually restricts them to lists over integers. We can equally well apply them to lists of floats, strings, and Boolean values:

\[
\text{head} \quad ["a \text{ string}", \ "another string"] \Rightarrow "a \text{ string}"
\]

\[
\text{null} \quad [2.3, \ 4.1, \ 9.52] \Rightarrow \text{False}
\]

\[
[\text{False}, \ \text{True}, \ \text{True}] \quad \text{++} \quad [\text{False}] \Rightarrow [\text{False}, \ \text{True}, \ \text{True}, \ \text{False}]
\]

These functions never inspect the contents of a list—all they do is alter the structure of the list without touching the actual elements. As a result, they are sufficiently generic to work on lists of all kinds. We call such operations *polymorphic functions*. They are very convenient, because they allow us to write list manipulation operations only once without having to repeat the definition for each type of list that we happen to use in our programs.

In fact, when we look at the types that we have given to these operations

\[
\text{head} \quad :: \quad [a] \rightarrow a \\
\text{tail} \quad :: \quad [a] \rightarrow [a] \\
\text{null} \quad :: \quad [a] \rightarrow \text{Bool} \\
\text{concat} \quad :: \quad [[a]] \rightarrow [a] \\
(\text{++}) \quad :: \quad [a] \rightarrow [a] \rightarrow [a]
\]

these types do not fix the element type of the input and output lists. The types \([a]\) do not fix a concrete element type. Instead, \(a\) is a *type variable* (as all variables, it starts with a lower-case letter). A type variable is a placeholder for a concrete type in the same way as a variable in a function definition is a placeholder for a value.

Whenever, we apply a polymorphic function, we have to commit to a *concrete* type for each of the type variables in that function’s signature. We call this process *instantiation* of the type signature. It is important that we instantiate all occurrences of a given type variables in a signature with the same type. In other words, we can use ++, for example, in the following ways:

\[
(\text{++}) \quad :: \quad [\text{Int}] \rightarrow [\text{Int}] \rightarrow [\text{Int}] \\
(\text{++}) \quad :: \quad [\text{Float}] \rightarrow [\text{Float}] \rightarrow [\text{Float}] \\
(\text{++}) \quad :: \quad [\text{Bool}] \rightarrow [\text{Bool}] \rightarrow [\text{Bool}]
\]
However, we cannot use it as \([\text{Int}] \rightarrow [\text{Bool}] \rightarrow [\text{Int}]\), because here \(a\) would be replaced by \(\text{Int}\) in one occurrence and by \(\text{Bool}\) in another occurrence.

Generally, the names of type variables (like function names and argument variables) must start with a lowercase letter. This makes it easy to distinguish them from concrete types (such as, \(\text{Int}\) and \(\text{Float}\)), which always start with an uppercase letter.

The polymorphism that we discussed so far is more precisely called *parametric polymorphism*. Its distinguishing feature can be illustrated at a function, such as

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

The implementation of the function (i.e., the equation defining the function) is independent of the instances of the type parameters \(a\) and \(b\) of its signature. There is a second form of polymorphism, which we call *ad-hoc polymorphism*, where the implementation of the function varies with the instances of its type variables. This form of polymorphism is often simply called *function overloading*. As ad-hoc polymorphism requires varying implementations for varying types, overloaded functions are usually only available for a subset of all possible type instances. In other words, in contrast to functions that are parametric polymorphic, we cannot use an overloaded function at any possible type instance. The set of types for which an overloaded function can be used is called a *type class* in Haskell.

For example, arithmetic operations like \(+\) and \(\ast\) are defined on all types contained in the class \(\text{Num}\) of numerals (which includes \(\text{Int}\) and \(\text{Float}\) among other numeric types). For example, the type of \(+\) is

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

The polymorphism here is constrained, by the *type constraint* \(\text{Num a}\). It enforces that types used to instantiate \(a\) in an application of \(+\) or only draw from elements of the type class (i.e., set of types) called \(\text{Num}\). In other words, we can read \(\text{Num a} \Rightarrow [a] \rightarrow a\) as follows:

For all types \(a\) that are a member of the type class \(\text{Num}\), \(+\) has type \(a \rightarrow a \rightarrow a\).

So, as \(\text{Int}\) is a member of \(\text{Num}\), we can use \(+\), for example, as \(\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}\).

Why is it important to have a constraint on the instances of \(a\) in the type of \(+\)? We simply cannot give a reasonable definition for \(+\) on many types; for example, \(\text{True} + \text{False}\) does not make any sense. So, to exclude the instantiation \(\text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}\) from the type of \(+\), we need the constraint.

With this knowledge, we can now also give a polymorphic type to the definition of \(\text{product}\):

\[
\begin{align*}
\text{product} & : : \text{Num a} \Rightarrow [a] \rightarrow a \\
\text{product} [ ] &= 1 \\
\text{product} (x:xs) &= x \ast \text{product} \, xs
\end{align*}
\]

It is also constrained to type arguments contained in \(\text{Num}\). This constraint arises from the use of the numeral \(1\) and the function \(\ast\) in its definition.

Apart from \(\text{Num}\), two other types classes are used very frequently: they are \(\text{Eq}\) and \(\text{Ord}\). For all types that are within these classes, we have comparison functions available. More specifically, equality is defined on all members of \(\text{Eq}\) and an ordering relation on all members of \(\text{Ord}\):

\[
\begin{align*}
(==) & : : \text{Eq a} \Rightarrow a \rightarrow a \rightarrow \text{Bool} \\
(/=) & : : \text{Eq a} \Rightarrow a \rightarrow a \rightarrow \text{Bool} \\
(>) & : : \text{Ord a} \Rightarrow a \rightarrow a \rightarrow \text{Bool} \\
(\geq) & : : \text{Ord a} \Rightarrow a \rightarrow a \rightarrow \text{Bool} \\
(<) & : : \text{Ord a} \Rightarrow a \rightarrow a \rightarrow \text{Bool} \\
(\leq) & : : \text{Ord a} \Rightarrow a \rightarrow a \rightarrow \text{Bool}
\end{align*}
\]
4.5 Exercises

1. What are the values of the following expressions and what is wrong with the ones that give errors?

   1: [2,3,4]
   1:2:3:4:[]
   [1,2,3]:[4..7]
   [1,2,3] ++ [4..7]
   1: ['a','b']
   "abc"++"cd"
   "a":"bCc"
   "a" ++ "bCc"
   'a':'b'
   'a':"b"
   [1,4,7] ++ 4:[]
   [True,True:[]]
   True:[True,False]

2. Write a recursive function `fact` to compute the factorial of a given positive number (ignore the case of 0 for this exercise).

   ```
   fact n = 1 * 2 * ... * n
   ```

   Why is the function `fact` a partial function? Add an appropriate error case to the function definition.

3. Write a recursive function `enumFromTo` that computes a list of integer numbers in a given range:

   ```
   enumFromTo m n = m, m + 1,...n
   ```

4. Write a recursive function `sum` to compute the sum of a list of numerals:

   ```
   sum [x1,x2,...,xn] = x1 + x2 + ... + xn
   ```

   Justify the use of the following type signature: `sum :: Num a => [a] -> a`

5. Here is a list of common mistakes made when defining `sum`. Discuss what is wrong and why.

   • `sum x:xs = ...`
   • `sum [x:xs] = ...`
   • `sum [xs] = ...`
   • forgotten base case

6. Write a recursive function `removeOdd` that, given a list of integers, removes all odd numbers from the list, e.g.,

   ```
   removeOdd [1, 4, 5, 7, 10] = [4,10]
   ```

   Hint: You can use the function `odd :: Int -> Bool`, which tests whether a number is odd.
Chapter 5

Lists & Higher-order Functions

“A list is only as strong as its weakest link.”
—Don Knuth

Most list processing functions are well suited to illustrate the concept of recursion; hence, we will look next at list processing with the goal of reinforcing the concept of recursion as well as to learn more about this versatile data structure. Lists are a convenient data structure for maintaining collections of data items, as long as no efficient random access to individual elements is required. For the moment, we are not concerned with efficiency, but we will revisit this issue in Chapter 12.

In the second part of the present chapter, we will see that we can regard functions as data items, and thus, define functions that manipulate other functions. However, as this feature in its full flexibility is a more advanced topic, we will only discuss the basic idea as well as some basic applications of the features.

5.1 Extracting Information from a List

We will begin our discussion about list processing functions, with functions that extract information from lists.

5.1.1 Searching Through Lists

Given a list of movies available in a video store, we might want to check whether a particular movie is available. To do so, we need a function `elem` that searches for a given element in a list. The most general type for `elem` would be

```
  elem :: Eq a => a -> [a] -> Bool
```

All that we require of the type of the list elements is that it is a member of the type class Eq, which guarantees that equality—and in particular, the function `(==)`—be defined over the list elements. See the lecture notes of the previous week for more information about type classes.

So far, the recursive functions that we have encountered have had exactly one base case and one stepping case. For `elem`, the base case is clear: in case of an empty list, the searched for value is clearly not contained in the list. So, we have

```
  elem e [] = False
```

The situation is, however, more complicated for the stepping case. If the list is not empty, i.e., we have a cons operator `x:xs`, we can distinguish two cases: (1) The head of the list `x` may just be the element that we are looking for or (2) it may be a different element. In the first case, we are finished and the search was successful. In the second case, we have to check the rest of the list, `xs`. So, overall, this gives us
elem :: Eq a => a -> [a] -> Bool

elem e [] = False
elem e (x:xs) | e == x = True
| otherwise = elem e xs

With this definition, we can now check lists of movies as follows:

```haskell
elem "Strange Days" ["The Matrix", "Strange Days", "Blade Runner"]
⇒ True

elem "Trainspotting" ["The Matrix", "Strange Days", "Blade Runner"]
⇒ False
```

Note that we can use functions with names consisting of alphanumeric characters as infix functions, by enclosing the function name in backquotes, e.g., 5 `elem` [2, 0, 5, 3].

It gets a bit more complicated when we do not simply want to check for a single element in a list, but, rather, to pick out all elements that fulfil a certain criterion, for example, all strings whose first letter matches a given letter. Such a function would have type

```
filterFirstLetter :: Char -> [String] -> [String]
```

Like `elem`, the function `filterFirstLetter` needs to distinguish two stepping cases: the string at the head of the string either starts with the given first letter or not. But, unlike for `elem`, `filterFirstLetter` cannot finish as soon as it has spotted the required element; instead, it must add this element to the result, then recurse over the rest of the list:

```haskell
filterFirstLetter letter [] = []
filterFirstLetter letter (str:strs)
  | letter == head str = str : filterFirstLetter letter strs
  | otherwise = filterFirstLetter letter strs
```

### 5.1.2 Association Lists

Often, we want to store more than one data item per list element—for example, a video rental shop might wish to keep track, for each movie title, whether it is currently rented by a customer. Informally, we can capture this information in a table:

<table>
<thead>
<tr>
<th>Title</th>
<th>Rented</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Matrix</td>
<td>No</td>
</tr>
<tr>
<td>Strange Days</td>
<td>Yes</td>
</tr>
<tr>
<td>Blade Runner</td>
<td>No</td>
</tr>
</tbody>
</table>

Here a rental status is associated with each movie title. We can represent the information in the above table by a list of pairs:

```
type MovieList = [(String, Bool)] -- title, rented

movies = ["The Matrix", False],
        ["Strange Days", True ],
        ["Blade Runner", False]]
```

Lists such as these are called association lists, since they associate with a key value—in this example the movie title—one or more property values, here the rental status. In the general case, we use an \( n \)-tuple to associate \( n - 1 \) property values with a key.

A common operation over association lists is, given a key, to look up a property value. The following definition of `lookup` is generic, in that it works on association lists (built from pairs) of various types. The only constraint is that the key type allows comparisons, which is expressed by the use of the type class `Eq` in the type signature:
lookup :: Eq a => a -> [(a, b)] -> b
lookup key [] = error "lookup: Element not found!"
lookup key ((key', val) : kvs)
  | key == key'     = val
  | otherwise       = lookup key kvs

The structure of `lookup` is similar to that of `elem`. However, in the case of an empty list (i.e.,
when we cannot find the key that we are looking for), an error is raised.¹

### 5.1.3 A Slightly Larger Example

As another example, consider the following situation in a supermarket, where there is a price list
containing all offered products, along with their prices:

<table>
<thead>
<tr>
<th>Product</th>
<th>Cents (excl. GST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mango</td>
<td>249</td>
</tr>
<tr>
<td>Water, 2l</td>
<td>199</td>
</tr>
<tr>
<td>Cheese</td>
<td>399</td>
</tr>
<tr>
<td>Bread</td>
<td>150</td>
</tr>
</tbody>
</table>

The items bought by a given customer can also be viewed as an association list that relates
product with quantity purchased:

<table>
<thead>
<tr>
<th>Item</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bread</td>
<td>1</td>
</tr>
<tr>
<td>Mango</td>
<td>3</td>
</tr>
<tr>
<td>Cheese</td>
<td>2</td>
</tr>
</tbody>
</table>

We model this situation in Haskell as follows:

```
typetype Cents = Int
type PriceList = [(String, Cents)]
type ShoppingList = [(String, Int)]
```

Given the supermarket’s price list and a shopping list, we now want to compute the cost incurred
by that shopping list, i.e., a function:

```
cost :: PriceList -> ShoppingList -> Cents
```

The function must add up the costs of the individual items and, to top it off, add GST.²

The function `cost` is obviously more complex than the functions that we have implemented
so far. Therefore, we will follow good software engineering practice and decompose the problem
into smaller subproblems rather than write a single monster function. Overall, there are three
different tasks that are needed to solve the problem of computing the total cost:

1. For each item on the shopping list, we must look up its price.

2. For each item on that list, we must multiply the price of the item by the quantity in the
   shopping list, and then we must sum up the results.

3. We must add GST.

Along the lines of this problem decomposition, we can implement `cost` by delegating the solution
of the first and second tasks to two auxiliary functions `collectPrices` and `sumPrices`:

¹For an alternate definition of `lookup` that handles failure differently, refer to the Prelude definition of `lookup`.
²For the sake of simplicity, let us assume that all items attract GST.
Lists & Higher-order Functions

cost :: PriceList -> ShoppingList -> Cents
cost prices slist =
  let individualCosts = collectPrices prices slist
  subtotal = sumPrices individualCosts
  in
  subtotal + subtotal `div` 10 -- add GST

We use some new syntax here. The let-in expression serves—very much like a where clause—the introduction of local bindings. However, the order of the bindings and the main expression where these bindings are used is exchanged.

**PriceList** = [(String, Cents)]

<table>
<thead>
<tr>
<th>Product</th>
<th>Cents (excl. GST)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mango</td>
<td>249</td>
</tr>
<tr>
<td>Water, 2l</td>
<td>199</td>
</tr>
<tr>
<td>Cheese</td>
<td>399</td>
</tr>
<tr>
<td>Bread</td>
<td>150</td>
</tr>
</tbody>
</table>

**ShoppingList** = [(String, Int)]

<table>
<thead>
<tr>
<th>Item</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bread</td>
<td>1</td>
</tr>
<tr>
<td>Mango</td>
<td>3</td>
</tr>
<tr>
<td>Cheese</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 5.1: Structure of computing the cost from a shopping list

Figure 5.1 provides an overview of the dataflow required in the implementation of cost. The data structures as intermediate values also determine the types of the two auxiliary functions.

The function collectPrices extracts the price of each item on the shopping list and pairs it with its quantity. It has type

\[
\text{collectPrices} :: \text{PriceList} \rightarrow \text{ShoppingList} \rightarrow \{(\text{Cents}, \text{Int})\}
\]

and on our above example inputs, it produces

<table>
<thead>
<tr>
<th>Price</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>150</td>
<td>1</td>
</tr>
<tr>
<td>249</td>
<td>3</td>
</tr>
<tr>
<td>399</td>
<td>2</td>
</tr>
</tbody>
</table>

Moreover, the function

\[
\text{sumPrices} :: \{(\text{Cents}, \text{Int})\} \rightarrow \text{Cents}
\]

takes the result of collectPrices and adds up the individual costs, which in the case of our example would result in 1695. To this subtotal, the above definition of cost adds 10% GST, with an overall total of 1864.

Now, we are left with finding an implementation for collectPrices and sumPrices. Both functions can be implemented by simple recursion, provided that we reuse the previous generic definition of lookup to implement collectPrices.
5.2 Updating List Elements

collectPrices :: PriceList -> ShoppingList -> [(Cents, Int)]
collectPrices prices [] = []
collectPrices prices ((item, quantity) : items) =
  (lookup item prices, quantity) : collectPrices prices items

This now demonstrates the usefulness of polymorphic definitions. If we had implemented
lookup above specifically for the movie list, with type

lookup :: String -> MovieList -> Bool

then we would have had to redefine it for use in collectPrices. However, the polymorphic
definition

lookup :: Eq a => a -> [(a, b)] -> b

now pays off and saves us some programming work. This is precisely why good programmers
are happy to accept the more complex type signatures of polymorphic functions: they save work
later on.

Finally, we implement sumPrices as follows:

sumPrices :: [(Cents, Int)] -> Cents
sumPrices [] = 0
sumPrices ((price, quantity) : rest) =
  price * quantity + sumPrices rest

This example is now also big enough to demonstrate the power of very high-level programming
languages such as Haskell. If we were to implement the same program with the same decomposi-
tion in a less advanced language, say Java, we would end up with a significantly larger program.

5.2 Updating List Elements

So far, we have been given a list and extracted information from that list, but we have not modified
the list. In the previous example of the shopping centre, however, we might wish to update the
price list as needed. Adding new items is easy, we just extend the price list with : or ++. More
interesting is the task of updating the pricing information for an already existing product. We do
this with a recursive function.

For example, we might want to update the price of a product in the price list from the
previous example:

[("Mango" , 249),
 ("Water", 21", 199),
 ("Cheese", , 399),
 ("Bread", 150)]

update pricelist "Mango" 199

["Mango", 199),
 ("Water", 21", 199),
 ("Cheese", , 399),
 ("Bread", 150)]

We can implement this functionality by a recursive definition that is similar to lookup. However,
it must copy the whole input list into the output instead of extracting a single element.
update :: PriceList -> String -> Cents -> PriceList
update []       updProd newPrice =
    error "update: unknown product"
update ((prod, price):pricelist) updProd newPrice
    | prod == updProd= (prod, newPrice): pricelist
    | otherwise = (prod, price): update pricelist updProd newPrice

5.3 A Brief Look at Sorting

A important operation on lists is to sort the elements. We examine here a basic sorting algorithm
called insertion sort. It is based on a routine that inserts an element into an already sorted list,
such that the ordering in the list is preserved. For example,

    insertSorted 7 [2, 4, 6, 9] \Rightarrow [2, 4, 6, 7, 9]

We can easily implement this function recursively by traversing the list up to the first element
that is greater than the new element that we want to add to the list. Then, we add the new element
immediately before this greater one.

    insertSorted :: Ord a => a -> [a] -> [a]
    insertSorted e [] = [e]
    insertSorted e (x:xs) | e < x = e : x : xs
        | otherwise = x : insertSorted e xs

The use of the type class Ord guarantees that we can compare values of type a and determine
whether one is bigger than another.

Based on insertSorted, we define the sorting algorithm as follows:

1. Given an unsorted list,

2. start with an empty result list, and

3. successively, use insertSorted to insert each element of the unsorted list into the result
   list.

The result is guaranteed to be sorted due to the property of insertSorted, which preserves the
ordering of the list into which it inserts elements. In Haskell, we can express this as follows:

    isort :: Ord a => [a] -> [a]
    isort [] = []
    isort (x:xs) = insertSorted x (isort xs)

There are many very sophisticated sorting algorithms, which reduce the costs of sorting in
various ways. You will study some of these in later courses.

5.4 Higher-order Functions

A recurring theme in learning about programming is that of abstraction. Functions abstract over
values by replacing some concrete values by variables. Types abstract over values, separating
them into classes that form the admissible input or output values of functions. Polymorphic
definitions abstract over concrete types by allowing us to define functions that, defined once,
work for a whole range of types. The list goes on and on, and the ability to handle abstractions
and to rapidly switch between different levels of abstraction might well be the most important
skill of an expert programmer.
5.4 Higher-order Functions

5.4.1 Mapping

Below, we will have a brief look at another very useful form of abstraction, which can save a lot of programming work if used correctly. But let us start with an example. The following two functions incList and doubleList respectively increment and double the elements in a list:

\[
\begin{align*}
\text{incList} & \quad : \quad \text{[Int]} \rightarrow \text{[Int]} \\
\text{incList} & \quad [] = [] \\
\text{incList} & \quad (x:xs) = (x + 1) : \text{incList} \; xs
\end{align*}
\]

\[
\begin{align*}
\text{doubleList} & \quad : \quad \text{[Int]} \rightarrow \text{[Int]} \\
\text{doubleList} & \quad [] = [] \\
\text{doubleList} & \quad (x:xs) = (2 \cdot x) : \text{doubleList} \; xs
\end{align*}
\]

As is to be expected, these two functions look very similar. In particular, their recursive structure is identical; it is just the operation applied to the head of list that is different. To emphasise this similarity, let us rewrite the functions a little, so that the actual operation on the head of the lists is moved into an auxiliary function:

\[
\begin{align*}
\text{inc} & \quad : \quad \text{Int} \rightarrow \text{Int} \\
\text{inc} & \quad x = x + 1
\end{align*}
\]

\[
\begin{align*}
\text{double} & \quad : \quad \text{Int} \rightarrow \text{Int} \\
\text{double} & \quad x = 2 \cdot x
\end{align*}
\]

\[
\begin{align*}
\text{incList} & \quad : \quad \text{[Int]} \rightarrow \text{[Int]} \\
\text{incList} & \quad [] = [] \\
\text{incList} & \quad (x:xs) = \text{inc} \; x : \text{incList} \; xs
\end{align*}
\]

\[
\begin{align*}
\text{doubleList} & \quad : \quad \text{[Int]} \rightarrow \text{[Int]} \\
\text{doubleList} & \quad [] = [] \\
\text{doubleList} & \quad (x:xs) = \text{double} \; x : \text{doubleList} \; xs
\end{align*}
\]

The two recursive functions are now exactly the same, except that they apply different functions to \(x\) in the recursive case. When we originally discussed functions, at the beginning of the lecture, this was the point where we said, let’s generalise and just abstract out the difference between the two expressions by introducing a function argument. So, why shouldn’t we do the same here?

We can just add another argument to the definition of incList (or doubleList), which is the function applied to the head of the list, \(x\). We rename the function map:

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

The function map provides us with a generic definition of all possible functions traversing a list for which some function is to be applied to the head of the list upon each recursive step. Hence, we can use it to define incList and doubleList:

\[
\begin{align*}
\text{inc} & \quad : \quad \text{Int} \rightarrow \text{Int} \\
\text{inc} & \quad x = x + 1
\end{align*}
\]

\[
\begin{align*}
\text{double} & \quad : \quad \text{Int} \rightarrow \text{Int} \\
\text{double} & \quad x = 2 \cdot x
\end{align*}
\]

\[
\begin{align*}
\text{incList} & \quad : \quad \text{[Int]} \rightarrow \text{[Int]} \\
\text{incList} \; \text{list} & \quad = \quad \text{map} \; \text{inc} \; \text{list}
\end{align*}
\]
doubleList :: [Int] -> [Int]
doubleList list = map double list

The function map receives as first argument another function. We call a function a higher-order function if it either receives another function as argument or if it returns a function as result. Functions that do not have functions as inputs and outputs (i.e., all functions that we discussed so far) are called first-order functions. Of note is the type of map. In our example, it would be

map :: (Int -> Int) -> [Int] -> [Int]

Or, in words, map receives a function mapping integers to integers (such as inc or double), together with a list of integers, and then produces another list of integers. However, if we have a close look at the definition of map, we see that there is nothing that restricts the function to lists of integers. In fact, it is the first argument passed to map that determines what type of values get manipulated. As a result, we can give the following polymorphic type to map:

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

Given a function that maps values of some type a to values of type b, map can transform a list of as to a list of bs. This implies that we can also use it, say, on a list of strings, as in

reverseList :: [String] -> [String]
reverseList list = map reverse reverse list

The function reverseList reverses every string in a given input list.

Generally, we can say that map behaves as follows:

map foo [x1,...,xn] = [foo x1,...,foo xn]

Given a function foo, map applies foo to each element x_i of its input list.

5.4.2 Filtering

Another frequently used operation where a higher-order function is helpful is that of extracting out of a list those elements that meet a given criterion. We specify the extraction criterion as a function from list elements a to Bool, i.e., a -> Bool. We generally call such functions predicates. As an example of the use of the filtering function consider

filter odd [1, 2, 3, 4, 5, 6] ⇒ [1, 3, 5]

It uses the built-in function odd :: Int -> Bool to extract all odd numbers from a list.

We can define filter by a recursive traversal of the input list, where in each recursive step the predicate is used to determine whether the current head of list should be included into the result or not.

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
| otherwise = filter p xs

Two functions that are related to filter are takeWhile and dropWhile. They do not consider all elements in a list, but just a prefix of the list that fulfils a certain criterion. For example, we have

dropWhile smaller5 [1, 2, 3, 4, 5, 6, 3, 2, 1] ⇒ [5, 6, 3, 2, 1]
takeWhile smaller5 [1, 2, 3, 4, 5, 6, 3, 2, 1] ⇒ [1, 2, 3, 4]
where
smaller5 x = x < 5
Note how `dropWhile` and `takeWhile` completely disregard any elements past the first one that does not satisfy the given predicate. We can define these two functions (which are part of the Haskell Prelude) as follows:

```haskell
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile p [] = []
dropWhile p (x:xs) | p x = dropWhile p xs
                 | otherwise = x:xs

takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []
takeWhile p (x:xs) | p x = x : takeWhile p xs
                   | otherwise = []
```

### 5.4.3 Updating of Lists Revisited

On page 35, we discussed the updating of lists, and in particular, a function `update` that could update the price tag of a product in a supermarket price list. We used the following explicit recursive definition to implement this functionality:

```haskell
update :: PriceList -> String -> Cents -> PriceList
update []         updProd newPrice = error "update: unknown product"
update ((prod, price):pricelist) updProd newPrice
       | prod == updProd = (prod, newPrice) : pricelist
       | otherwise       = (prod, price) : update pricelist updProd newPrice
```

Equipped with our new knowledge about higher-order functions, we can see that—modulo flagging the error condition—we can achieve exactly the same behaviour with `map`. The function `map` preserves the structure of its input list, which is exactly what `update` is supposed to do. Although `map` can potentially change every element of a list, in `update`, we only want to alter elements with the matching key. This means that we must make a case distinction—using a conditional expression or guards—in the function that we pass as argument to `map`. We test whether the key of the element passed to the function matches the key that we want to update, and if so, alter the property:

```haskell
update :: PriceList -> String -> Cents -> PriceList
update prices newProd newPrice = map updateElement prices
  where
    updateElement (prod, price) | prod == newProd = (prod, newPrice)
                               | otherwise = (prod, price)
```

### 5.5 Exercises

1. Write a recursive function `delete` which, given a string and a character, returns the string with all occurrences of the character removed. For example:

   ```haskell
delete 'x' "x-files" = -files"
```

2. In a similar style as `delete`, write a function substituting a given character with another character (instead of deleting it):

   ```haskell
substitute 'e' 'i' "eigenvalue" = "iiginvalui"
```

3. Write a function that returns both the shortest and the longest string in a list:
shortestAndLongest ["abc"] = ("abc","abc")
shortestAndLongest ["This", "sentence", "is", "ridiculous"] =
("is","ridiculous")

What do you think should happen when an empty list is passed to shortestAndLongest?

4. Write a function that yields a list of all the properties associated with all occurrences of a key in an association list:

\[
\text{lookupAll } "Plates" \left[\left("Forks", 10\right), \left("Plates", 5\right),
\left("Cups", 0\right), \left("Plates", 1\right)\right] = [5,1] \\
\text{lookupAll } "Knives" \left[\left("Forks", 10\right), \left("Plates", 5\right),
\left("Cups", 0\right), \left("Plates", 1\right)\right] = []
\]

It is sufficient if the function works only for association lists of type [(String, Int)].
Chapter 6

The Unix Programming Environment

“Those who do not understand UNIX are condemned to reinvent it, poorly.”
—Henry Spencer

6.1 The Unix Spirit

6.1.1 Operating Systems

An operating system (OS) is system-level software that acts as an intermediary between the computer hardware and the application software operated by the computer user. The OS provides a uniform interface to peripherals (such as keyboard, monitor, disks, printer, etc.) but also manages internal computing resources, such as memory and processor time. Moreover, the OS is responsible for the organisation of data on external storage (like hard disks) and for loading and executing programs. Application programs (like word processors, Web browsers, email clients, and so on) run on top of the OS, which provides basic library routines and hides variations in hardware from the application program. Figure 6.1 outlines this intermediary role of an OS.

<table>
<thead>
<tr>
<th>Application Programs</th>
<th>Operating System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>System Programs</td>
</tr>
<tr>
<td>Computer Hardware</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.1: Purpose of an operating system

Generally, a complete operating system consists of a kernel and a set of system programs. The kernel is the heart of the OS, which is started when the computer is switched on (i.e., when the system boots), and continues to run until it is switched off. It contains the code that directly accesses the hardware, handles the most critical system resources, and implements basic security policies. The kernel runs in privileged mode, which gives it unlimited access and control over all computer hardware. All other programs run in user mode, which means that they can access system hardware only with the help of the kernel and that the kernel can control their use of system resources. For example, a user-mode program can get only as much memory as the kernel allows it to use. The kernel also protects different user-mode programs from each other, so that one cannot corrupt another.

The system programs included in an OS handle administrative tasks, such as formatting disks, managing user accounts, setting up network access, as well as several daemons. Daemons are
programs that constantly run in the background and perform functions such as logging system
messages, handling incoming email, and allowing users to log into the system. Moreover, the
system programs also include user-visible tools for file manipulation, interactive shells, and usu-
ally a graphical windowing system. Some systems, furthermore, include software development
tools, such as program editors and compilers.

Some less advanced OSes integrate some of the system programs with the kernel, in an ill-
advised attempt to simplify the design. However, this is not optimal from a software design point
of view and usually reduces the flexibility and often the stability of the overall system as well.
System programs should be a comprehensive collection of small tools. Unix makes it very easy
to combine these tools to implement more complex functionality. In fact, the use of scripting
languages to combine the basic tools into complex applications was certainly one of the most
important contributions of Unix.

6.1.2 Unix: Portable, Simple, and Compositional

Unix was the one of the first OSes written in a machine-independent programming language,
namely C. This together with its simple, modular design and the fact that Bell Labs distributed
it at low cost, including the source code, led to a revolution in the OS area. The system was
licensed to many hardware vendors (IBM, Sun, HP, SGI, etc.) who ported and extended it to
ship with their hardware. The portability of the system was one key feature to its wide adoption.
Moreover, portability was nicely complemented by the (compared to other OSes) simple design.
It made it easier to arrive at a portable implementation in the first place, and in addition, it greatly
reduced the time needed by the engineers in the companies adopting Unix to understand and
subsequently improve the system. One of the key design aspects of Unix is its emphasis on a
modular and compositional system.

6.1.3 Free Software & Open Source Software

The fact that the initial Unix distributions included source code significantly contributed to Unix’
success. It led to a plethora of improvements and allowed everybody to study the system. This
was not only beneficial for universities, but also contributed to what is often called the Unix
culture. Computer engineers and scientists who used and studied the system intensively often
developed a deep appreciation for the functionality and simplicity of the Unix model, which
inspired them to apply it in their own designs.

At University of New South Wales, Prof. John Lions used the Unix source code to teach
operating systems in the mid-70s. He wrote a commentary on the code so that his students found
it easier to understand the system. Unfortunately, AT&T, who owned Unix at that time, eventually
stopped the publication of the commentary, as it contained all the core source code. As a result,
it may be the most photocopied book in computing history and when the commentary finally
was published in book form in 1996, it was called “the most famous suppressed manuscript in
computer history” on the back cover.1

As companies like AT&T got more paranoid about source code, they not only complicated
the teaching by John Lions and others, but also ceased to distribute the source code of the various
flavours of Unix. Even worse, they ‘extended’ the OS with various proprietary features in an
effort to get a marketing advantage over their competition—as a result, it became increasingly
hard to write software that worked on all flavours of Unix.

There were, however, three forces that after having been pushed to the fringe changed the
landscape again. The first is in direct lineage from AT&T’s Unix. The systems group at the
University of California Berkeley was a major player in the Unix arena and, in fact, they spawned
a whole family of Unix systems, which are known as BSD Unix; they were especially influential

1Salon Magazine has an article about John Lions at http://www.salonmag.com/tech/feature/1999/11/30/lions/
in networking. Moreover, a version of the BSD system, called FreeBSD was eventually made freely available, as was its source code.

The second force was the Free Software Foundation (FSF)\(^2\) founded by Richard M. Stallman (RMS), dedicated to implementing a free\(^3\) Unix system called GNU (GNU’s Not Unix). The formation of the FSF was a direct reaction to the increasingly closed nature of proprietary software systems. One of the principal contributions of RMS and the FSF was the GNU Public License (GPL), which essentially states that you can use the software for whatever purpose you like, and that if you distribute any corrections or improvements, you must also provide the source code and give others the same rights. Again, Unix’ modular design proved to be a key feature. The FSF did not have to write a whole OS in one go, but they could concentrate on writing the system programs first—one after another—and use them on a proprietary kernel. Finally, so the plan went, they would write a kernel to complete the system. However, as it turned out, somebody else outpaced them in that last step.

The third, and today probably most well known, force was Linus Torvalds, who, with the help of a loosely connected group of programmers and testers all over the planet—connected just by the Internet—wrote Linux, a Unix-like OS kernel, from scratch. Benefiting from the Unix modular structure and the work of the FSF, he did not need to write a whole suite of system programs in addition to his kernel. Instead, he used the GNU tools; adding what the FSF was still missing, namely, a kernel—the result was the GNU/Linux system, which is currently taking the server market storm and seems poised to make significant inroads in the desktop and embedded systems markets in the near future. Suprising for many is that Linus’s largest contribution to the community was not the Linux kernel, but a new software development methodology. Before Linux, it was accepted that a software system as complex as an OS kernel could only be developed by a closed group of specialists—in what is sometimes called the cathedral style of development. Linus demonstrated that there was an alternative—called the bazaar style. After an initial design and first code is developed by a single person (or small group), a large project can be pursued by a loosely coupled group of developers communicating via the Internet, using (a subset of) their user base as a huge distributed testing team.\(^4\) Chances are that this development will ultimately define how at least infrastructure software is developed in the future.

The result of all this was a renaissance of the idea of distributing programs in source form—an idea widely known as open source software today. Deeply entwined with the development of Unix and free software is the hacker culture.\(^5\) It describes the tradition of the people who collectively made Unix and the Internet possible. A comprehensive and enjoyable account of the development of hacker culture, Unix, and open source software is provided by Eric Raymond.\(^6\)

6.2 Unix—Small Tools and a Bit of Glue

We have emphasised a couple of times that modularity was a key issue in the design of Unix. In the following, we will have a look at some of the most important tools provided by a Unix system and the mechanisms that allow users to combine these tools to solve more complex tasks. The latter consist of mechanisms for communicating data between programs as well as support for turning sequences of shell commands into programs, called shell scripts.

\(^2\)http://www.gnu.org/
\(^3\)Free software is called ‘free’ not because it doesn’t cost anything, but because users are free to examine the source code, modify, and re-distribute it.
\(^5\)We mean “hacker” here in the traditional meaning of the word, where it describes “a person who enjoys exploring the details of programmable systems and how to stretch their capabilities, as opposed to most users, who prefer to learn only the minimum necessary.” (cf., http://www.tuxedo.org/˜esr/jargon/html/entry/hacker.html)
\(^6\)It is split over two documents: http://www.tuxedo.org/˜esr/writings/cathedral-bazaar/hacker-history/ and http://www.tuxedo.org/˜esr/writings/cathedral-bazaar/hacker-revenge/
6.2.1 Interactive Shells

Whenever you log into a Unix system remotely (i.e., using the ssh, rlogin, or telnet command) and whenever you open an xterm (or other terminal) window, Unix starts a command shell (or just shell) for you. A shell is a textual interface to all Unix commands, which may be a bit harder to learn than your average graphical user interface, but—after you have mastered it—is also a much more powerful way to interact with a computer.

An interactive shell waits for a line of commands from the user, which it executes. Such a command can be something simple like the command `ls` or more complex expressions combining multiple Unix tools into one command. Commands can also get arguments (like in a function application), where we distinguish two types of arguments: options (also called switches) and normal arguments. The former are distinguished by starting with one or two dashes (-) and the latter are usually file names. For example, we could have

```
$ cp command -R dir1 dir2
```

Unix has a number of different shells with varying feature sets. The original Unix shell is called the Bourne Shell and is available as a program called `sh`. The GNU project has re-implemented and significantly improved the original shell: `bash`, the Bourne Again Shell, is available on any GNU/Linux system and on other well-configured Unix systems.

6.2.2 Simple File and Directory Utilities

Let us start with command tools for manipulating directories. An interactive session, in Unix, always has a current working directory associated with it. In other words, if you specify only a file name (like `week10.tex`) instead of a complete path name to a file like

```
/home/chak/lectures/cs1011/notes/week10/week10.tex
```

then there is a default location where the file will be assumed to be located. The current working directory can be queried with the command `pwd`. Moreover, the current working directory can be changed with the command `cd`, which changes to the user’s home directory if no argument is given.

```
$ pwd
/home/chak/lectures/cs1011/notes/week10
$ cd ..
$ pwd
/home/chak/lectures/cs1011/notes
$ cd week11
$ pwd
/home/chak/lectures/cs1011/notes/week10
$ cd
$ pwd
/home/chak
```

Moreover, the command `ls` can be used to list the contents of a directory. Furthermore, `mkdir` and `rmdir` can be used to create and remove directories, respectively. Files can be copied, moved, and removed to a new location with `cp`, `mv`, and `rm`.

As said earlier, Unix protects users and their programs from each other. Therefore, with each file and directory, we associate file access permissions. When we give the command `ls` an extra argument of `-l`, it gives us a more verbose listing of the entries in a directory:

```
Many hackers enjoy these kind of word games.
```
The first entry represents the permissions. The permission information consists of 10 entries. The first is d if the file is a directory and it is - for a regular file. The other 9 entries are three user categories of three entries each:

```
d rwx r-x r-x
  user  group  other
```

The category “user” denotes the owner of the file. The category “group” denotes the user group to which the file is associated. Finally, “other” applies to all other users. In each category, “r” denotes read permission, “w” denotes write permission, and “x” denotes execute permission. In the case of a directory, execute permission means that files inside the directory may be accessed by members of that user category. A directory with only execute permission but without read permission allows you to access files whose name you already know; it gives users access to the file without allowing them to list any directory in our network.

File permissions may be altered with `chmod`. For the details of the arguments to that command, refer to info chmod.\(^8\)

### 6.2.3 Pathname Expansion

Many of the file manipulation tools can accept a list of filenames as arguments and when there is some regularity in the list of names, we can use patterns to specify the files we are interested in, instead of enumerating them all. For example, the pattern `*.hs` stands for all files ending with the three letters `.hs`—i.e., all Haskell files. In a process called pathname expansion (or globbing) the shell replaces this pattern by all file names in the current directory that match the pattern.

Generally, patterns may contain the characters `*` and `?` as well as character sequences enclosed in square brackets (`[` and `]`). Any occurrence of `*` matches any number of characters (also no character) and `?` matches exactly one character. Within square brackets, we can just have a list of characters, which means that the pattern matches any of these characters. For example, `[adx]0` would match the file names `a0`, `d0`, and `x0`. If the characters are consecutive, we can specify a range instead of enumerating all characters. For example, `[a-z]0` matches all files starting with a lower case letter followed by a 0.

Moreover, alternative strings can be enclosed in curly braces `{` and `}` and be comma-separated. For example, `{abc,def}.hs` matches both `abc.hs` and `def.hs`.

Due to their special meaning in file names, slash characters (`/`) and a (a dot) at the start of a filename or immediately after a / are treated specially. They are never matched by a pattern unless they are explicitly specified.

### 6.2.4 Inspecting and Searching In Files

The following list of commands is very helpful for viewing text files and extracting information from them. Check the details of the usage of these commands using the `info` command.

- `file`: Determine what kind of data is contained in a file.
- `cat`: Copy the contents of a file to the console.

\(^8\)info info will explain the info command to you.
- **more**: Like `cat`, but lets you interactively page through the file contents.
- **less**: A significantly improved variant of `more` from the GNU project.
- **head**: Display the first 10 (or optionally, any other number) of lines of a file (see also `tail`).
- **cut**: Cut columns of information out of text files containing tables.
- **grep**: Search in a file for occurrences of a specific string or pattern.

### 6.3 Glue, Part I: Redirection and Pipes

The individual tools discussed so far are useful on their own, but their real power and the elegance behind the Unix approach becomes only apparent when considering the mechanism provided to combine these tools, to perform more complex tasks. For a Unix system, a command conceptually acts as a filter that maps a stream of input values into a stream of output values:

```
Input (stdin)  ---|--- Unix command ---|--- Output (stdout)
                             \___/ Error output (stderr)
```

Furthermore, a stream of error messages might be produced, which we usually want to handle independently from the rest of the output. When used on their own, Unix commands are usually used to read their input from one or more files and to write the output to the console. However, Unix provides two mechanisms to change this behaviour:

1. We can **redirect** input or output to come from or go to any file.
2. We can **pipe** the output of one command into the input of another command.

Input/output redirection uses the following notation:

- `<(filename)` redirects stdin to read from a file.
- `>(filename)` redirects stdout to a file.
- `>>>(filename)` redirects stdout to append to a file.
- `2>(filename)` redirects stderr to a file.

For example,

```
grep cost *.hs >result
```

would write all lines containing the identifier `cost` in any Haskell file in the current directory into the file `result`. If afterwards we issue

```
grep PriceList *.hs >>result
```

the lines containing `PriceList` would be appended to the results from the previous command (note the use of two `>`).

By separating two commands with a vertical bar (`|`), we can pipe the output of the command to the left of the vertical bar into the input of the command to the right of the bar. In other words, `<command1> | <command2>` gets us the following setup:

```
Input ---|--- <command1> ---|--- <command2> ---|--- Output
```
For example, given that the command `grep wed tute-allocations` produces a long list of output (which doesn’t fit on the screen at once), we can pipe the output of `grep` into `less` so that we can page through the output:

```
grep wed tute-allocations | less
```

Sometimes, it makes also sense to apply the same command twice:

```
grep wed tute-allocations | grep tuba
```

restricts the output not only to the lines containing the string `wed`, but also the string `tuba`.

In the same way, more than two commands can be joined together.

### 6.4 Interpreter Versus Compiler

Before we continue investigating the features provided by a Unix shell, let us have a closer look at how a computer executes programs. Any computer has a native language, called its machine language, and it can execute programs in that language without any further software support. Unfortunately, the machine language of a computer depends on its brand (in fact, it depends on the type of microprocessor or CPU that is used in the machine). Moreover, machine language tends to be rather unwieldy for humans. A textual representation of a machine program looks as follows:

```
.globl Main_main_entry
Main_main_entry:
    leal -16(%ebp),%eax
    cmpl 88(%ebx),%eax
    jb .Lnul
    addl $24,%edi
    cmpl 96(%ebx),%edi
    jbe .Lnuk
.Lnul:
    jmp stg_gc_enter_1
.Lnuk:
```

Even for simple programs it goes on and on like this, which has motivated the invention of more human-oriented, high-level languages. Nevertheless, developers of, for example, operating systems and compilers, sometimes have to resort to the use of machine language for parts of their code.

High-level languages (such as Haskell, Java, C, C++, Perl, Visual Basic, to name just a few) are much easier to use, but unfortunately a computer cannot execute programs written in such a language without additional help. In fact, computer hardware can strictly only execute programs that are in machine code. For any other language, translation software is needed that converts the high-level language into machine language. Such software essentially works in one of two possible ways: as an interpreter or a compiler.

An interpreter translates commands of the high-level language while executing the interpreted program. In particular, if a program is executed twice, the translation process will be repeated. As a result, interpreted code is usually considerably slower than compiled code. On the positive side, interpreters are usually easier to handle than compilers. Often (as in the case of ghci), you can enter commands which are immediately executed and their result displayed.

Compilers avoid the overhead of repeatedly translating the same code over and over again by translating a high-level program as a whole (before it is executed) into a machine code program. This program in machine code is then saved into a file and can later be executed like any other.

---

9 Actual machine code is just a string of numbers, but the **symbolic assembler code** listed here is a closely related representation.
Unix command without requiring any further reference to the compiler. We call such a machine code program an executable.

GHC can actually, besides interpreting Haskell programs, also compile them. However, this requires us to meet a couple of constraints: among the modules constituting a compiled Haskell program, there must be one module called Main. In this module, there must be a function main. This function must be an I/O function. I/O functions will be discussed in more detail later. For the moment, it suffices that there is a function called print, which can print the result of an expression upon program execution. Here is an example of a simple program meeting the mentioned constraints:

```haskell
module Main where

sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList xs

main = print (sumList [1..10])
```

It defines the familiar function sumList and prints the result of evaluating sumList [1..10] when executed. We can compile this program using the command

```
ghc -o sum Main.hs
```

This will generate an executable called sum (the -o option prefixes the name of the executable to which the output should be written to). You can now run this newly compiled program by invoking

```
./sum
```

The two characters ./ in front of the program name indicate to the shell that you want to execute a command located in the current directory; remember that . is a special name for the current directory.

If your program runs too slow, you can add the option -O (that’s a capital “O”), which will make compilation slower, but also generate faster machine code.

### 6.5 Glue, Part II: Shell Scripts

Shell scripts are a particular instance of what is commonly called an extension language. As a designer of an application, it is usually rather difficult to anticipate all uses that users will find for that application. One convenient and powerful way out of this problem is to add an extension language to the application—i.e., a facility allowing users to extend the application themselves without having to extend the core of the application itself. Extension languages not only helped Unix become successful, but it also played a crucial part in the success stories of other applications. One of them is Emacs.\(^\text{10}\) The extension language of Emacs is a Lisp dialect—Lisp is, like Haskell, a functional programming language, but it is significantly older. Only the core of Emacs is written in the C language; everything else is written in Lisp, which made it significantly easier for users to write their own extensions to Emacs or just to configure it to their mode of usage. Another program that makes heavy use of extension languages is the Gimp\(^\text{11}\), an image manipulation application.

Unix shell scripts are just a collection of shell commands, of the same kind as those used interactively. However, some operations, such as conditionals and loops, are used more frequently in scripts rather than in interactive mode.

\(^{10}\) Although the first Emacs was created by James Gosling, the author of GNU Emacs is Richard Stallman (the founder of the Free Software Foundation)—so, Emacs is in fact part of the GNU project. The version that we are using, XEmacs, is a slight variant (also called a fork) of GNU Emacs. Both versions share most of the code base.

\(^{11}\) http://www.gimp.org
6.5.1 Hello World!

Generally, if a line contains a hash character (#), then all remaining characters up to the end of the line are considered to be comments. However, any shell script should contain on its first line a character sequence starting with the hash mark–exclamation mark pair. This is more than a comment, it specifies which shell the Unix system is supposed to use to interpret the script. A line of the form

```
#!/bin/sh
```

tells the system to use /bin/sh to interpret the following script.

The first script that we want to implement is the infamous “Hello World!” program. The shell command to print a given string to stdout is `echo`. So, “Hello World!” can be implemented as follows:

```
#!/bin/sh

echo "Hello World!"
```

Before we can execute the script, we have to adjust the permissions of the script. If we store the above script in a file `hello`, we could adjust the permissions using `chmod a+x hello`.

6.5.2 Verbose Directory Listing

We have seen that Unix commands can be passed arguments not unlike functions in Haskell. Clearly, it would be useful to be able to pass arguments to shell scripts, too. For any argument passed to a shell script there are special variables `$1`, `$2`, etc., that can be used to access those arguments. As for `$#`, it gives the number of available arguments and `$*` produces all the arguments at once.

We can use `$*` in a shell script, which we call `ll`, that produces the verbose listing of `ls`:\footnote{A note for the discriminating hacker: The more natural approach to defining `ll` is to use an alias. See info bash for details about the definition of aliases.}

```
#!/bin/sh

# This will produce a verbose directory listing
#
ls -l $*
```

We would like to use scripts such as these independently of the current working directory we happen to be in. Unix shells inspect the special purpose variable `$PATH` whenever a command is to be executed. It contains a colon-separated list of directories where the system looks for commands. The standard place for personal scripts is the directory `/bin/` (i.e., a subdirectory `bin/` in the users home directory), which should then be added to `$PATH`. Unix systems use a whole range of such variables, which can be used to configure the system. These variables are called `environment variables`.

Generally, we can assign values to variables in shell scripts using an equality sign, as in

```
colour="green"
```

Note that there must be no space to the left or right of the equals sign; otherwise, a syntax error will occur. Whenever the value of a shell variable is queried, the variable name has to be prefixed with a dollar character, as in

```
echo $colour
```
*** Testing submission of Angie Foo <abcd123>

--- Submitted Program ---
module Main
where

-- super cool function incrementing all values in a list
--
incList :: [Int] -> [Int]
incList l = map (+1) l

-- what to execute in a test run
--
main = print (incList [1..10])

--- Success ---
Test passed (1 mark)

Figure 6.2: Example testing report

6.6 Autotest Script

As a more extensive example of a shell script, we are going to discuss the basics of a script that can be used for automatically determining performance marks for student assignments. Our scripts are supposed to execute in the following environment:

- Directory data/: Student data, test data, and submissions.
- File data/stuname: Associates usernames of students with real names.
- Directory data/submissions/: One subdirectory for each student.

As an example, let us assume that there are two students with usernames abcd123 and vwxy567—with corresponding directories below data/submissions/. Each of the student directories contains a file Inc.hs, which is the Haskell program that is to be assessed.

In this environment, our autotesting software shall perform the following tasks:

- Load each submission into the Haskell interpreter GHCi.
- Run a simple test—i.e., query the type of a function and evaluate an expression.
- Write a report for each submission into a file result in the students submissions/ directory.

Figure 6.2 contains an example report of a test run of a single submission.

6.6.1 Design

When implementing a shell script, we should follow the same rules for good software design as those that we discussed in the context of Haskell. In particular, we should decompose the problem at hand properly and strive to achieve a clear program structure. Furthermore, we should write clean code and document it properly.

So, what would be a good decomposition for the autotesting software? Certainly, it makes sense to write a shell script that can assess a single student first, and then extend this to a system handling all students. So, let us write two scripts as follows:
1. One script to test a single submission and write the report to stdout—let’s call this script `testone`.

2. Another script that applies `testone` to each submission and places the reports into 
   
   `data/submissions/(student)/result`
   
   Let’s call this script `testall`.

### 6.6.2 Testing a Single Submission

The first question that we have to solve is, how does the `testone` script know which student to assess? Clearly, a command argument seems to be in order here. Now, one problem with shell scripts is that regardless of the number of arguments that are actually used in the script, we can call them with any number of arguments without the system complaining about any mismatch—note that this is clearly different from the situation that we have in Haskell. This can lead to unexpected and difficult-to-find errors when a script is called in the wrong way, possibly from another script. We can alleviate this problem a little by adding checks to our scripts, which detect and report this and similar problems as early as possible.

Getting back to our present problem, we want to test at the start of the script `testone` whether the number of arguments is exactly one. For this we need a conditional statement, which in a shell script has the following general form:

```bash
if [ $(conditional expression) ]; then
  (commands if true)
else
  (commands if false)
fi
```

Note the space characters to the left of `[` and the right of `]`. They are syntactically necessary and omitting them will lead to error messages. The following code will now test whether the number of arguments is correct or not:

```bash
if [ $# -ne 1 ]; then
  echo "$0: Exactly one argument expected"
  exit 1
fi
```

Note that it has no `else` branch, which is in fact optional in shell scripts. If the condition is not fulfilled, execution proceeds immediately after the keyword `fi`. The variable `$0` contains the name under which the current shell script was invoked and should by Unix coding conventions always be included into error messages.

The command `exit 1` terminates the execution of the shell script and sets the `exit status` to 1. In Unix, every terminating program is assigned an exit status, which can be used by the calling process (possibly a shell script) to determine whether the program terminated abnormally and maybe even determine the reason for termination. The default is an exit status of 0, which implies that no error occurred and the program terminated normally. An exit status greater than 0 usually signals an error condition and the documentation for the standard Unix commands contains information on the various exit codes and their meaning, which varies between programs.

In order to make shell scripts more readable, it is advisable to assign the value of argument variables, such as `$1`, to variables with more expressive names and use them in the remainder of the script. In fact, we define the following set of variables:

```bash
namedb=stunames
modname=Inc.hs
```
username=$1
prgmname=submissions/$1/$modname

The variable $namedb contains the filename of student database, $modname the name of the Haskell module that we want to test, username contains the argument to the shell script (i.e., the username of the student to assess), and finally, prgmname contains the full path to the student’s Haskell code.

6.6.3 Computing the Report Header

The header of the report in Figure 6.2 contains not just the username, but also the real name of the student who is assessed in that particular report. However, as the script testone initially only gets the username, it has to extract the missing information, i.e., the real name, out of the student database stunames. The contents of the latter, in our example setup, is

```
abcd123;Angie Foo
vwxy567;John Smith
```

Given a username like abcd123, how can we get the name (i.e., Angie Foo)? Following the Unix philosophy of combining existing tools to achieve a particular result, we may try to use grep to extract the correct line (or row) out of stunames. In particular, we have

```
$ grep abcd123 stunames
abcd123;Angie Foo
```

Moreover, the command cut can extract a particular column (or set of columns out of a tabular text file). In our case, we have a table, where columns are separated by ; characters and all real names are contained in the second column. The -d option of cut lets us specify the separator and -f the column that we want to extract:

```
$ cut -d ';' -f 2 stunames
Angie Foo
John Smith
```

Clearly, the intersection of the result computed by grep and that computed by cut would give us the desired result (i.e., the name of the student identified by a particular username). How can we compute this intersection? Unix pipes come to the rescue—piping the result of grep through the cut command extracts the desired column for the single row identified by grep. So, we have

```
$ grep abcd123 stunames | cut -d ';' -f 2
Angie Foo
```

The next problem is to assign the result of a Unix command to a shell variable. This is achieved by enclosing the command in backquotes. So, in testone, we write

```
stuname=`grep $username $namedb | cut -d ';'; -f 2`
```

Now, we can print the header of the report with the following command

```
echo "*** Testing submission of $stuname <$username>"
```

13In this case, there is actually another more powerful program called awk that could solve the problem. However, awk is a bit tricky to use; if you are interested, check out info awk.
6.6.4 Running GHC

We want to test a submission in two ways: (1) check the type of the function \textit{incList} and (2) run the function. Given that GHC allows us to compile programs, the latter is relatively straightforward. More problematic is the former, as it requires to use the interactive GHCi. This implies that the shell script testone not only has to invoke GHCi, but it also has to somehow ‘type’ commands into an interactive GHCi session to query the type of \textit{incList}. Here the full power of Unix capability to redirect stdin and stdout come into play. The GHCi commands that we want to execute during testing are the following

\begin{verbatim}
:t incList
:q
\end{verbatim}

We store them in a file, which we call \textit{ghci.in}. Now, the following command invokes GHCi and executes the commands:

\begin{verbatim}
ghci $prgmname <ghci.in
\end{verbatim}

The output is as follows:

\begin{verbatim}
<...deleted GHCi startup logo...>
Loading package std ... linking ... done.
Compiling Main ( submissions/abcd123/Inc.hs, interpreted )
Main> :t incList
[Int] -> [Int]
Main> :q
Leaving GHCi.
\end{verbatim}

To further process this output, we redirect it into a file called \textit{test1.out} and we use the Unix \texttt{tail} command to remove all, but the last 4 lines (to eliminate the startup messages):

\begin{verbatim}
ghci $prgmname <ghci.in 2>/dev/null | tail -$msglen >test1.out
\end{verbatim}

The argument \texttt{2>/dev/null} removes all output that GHCi sends to \texttt{stderr}. Here the elegance of the Unix model becomes clear. GHCi was never designed to be operated by a program instead of a human user. Nevertheless, the Unix redirection mechanism enables us to operate GHCi within a shell script.

6.6.5 Inspecting the Result

All that remains to be done is to inspect the output written by GHCi into \textit{test1.out} and determine whether it is correct. First, we check whether GHCi raised an error message when loading the Haskell module. GHCi always indicates an unsuccessful load operation, printing that it \texttt{Failed} to load files; thus, we can use \texttt{grep} to check for this string in the output. If it is present, the test failed. But how does our shell script know whether \texttt{grep} failed or not? Earlier we discussed the use of a numerical exit status to communicate failure to the calling process. The documentation below the menu point “Diagnostics” in the info manual for \texttt{grep} (available with \texttt{info grep}) states that \texttt{grep} exits with 0 when it finds at least one match, and otherwise, with 1. In a shell script, the special variable \$? contains the exit status of the last command. So, overall we can test for \texttt{Failed} in \textit{test1.out} as follows:

\begin{verbatim}
grep Failed test1.out >/dev/null
if [ $? = 0 ]; then
  echo " === Errors While Loading Program ==="
cat test1.out
exit
fi
\end{verbatim}
Note how we redirect any output of `grep` to the special file `/dev/null`. The latter is the informational equivalent of a black hole. Any bits sent into it are gone forever.\footnote{Also known as the \textit{bit bucket}, cf. http://www.tuxedo.org/~esr/jargon/html/entry/bit-bucket.html}

If the program loaded without errors, we compare the output of GHCi with the output generated by a correct implementation of the Haskell program. The Unix tool for comparing two text files is called `diff`. When given the option `-y`, it compares the two files side by side marking any lines where they differ. Its exit status is 0 if the files are equal, which in our application means that the test was successful. As we want to include the output of `diff` into the test report only if the test failed, i.e., if there are any differences, we redirect the output of `diff` into a file `diff.out` and use a conditional expression to handle the two possible cases:

```
    diff -y -W 80 expected test1.out >diff.out
    if [ $? != 0 ]; then
        echo "=== FAILURE ==="
        cat diff.out
    else
        echo "=== Success ==="
        echo "Test passed (1 mark)"
    fi
```

The option `-W 80` instructs `diff` to produce output which is only 80 characters wide.

The complete code of `testone` is listed in the appendix.

For the moment, I omit the discussion of the script `testall`, because it is one of our exercises. As a final note, for more information on `bash`, consult \texttt{info bash}.

### 6.7 Autotesting Scripts

The first script, called `testone`, tests one submission.

```
#!/bin/sh

# Autotest one program
#
# Manuel M. T. Chakravarty, 2000 & 2001

# constants
#
# namedb=stunames
modname=Inc.hs
msglen=4

# test for correct number of arguments
#
if [ $# != 1 ]; then
    echo "$0: Exactly one argument expected"
    exit 1
fi

# the argument is the username of the person whose solution we should test
#
username=$1
prgmname=submissions/$1/$modname
```
# extract the full name from the database
#
stuname=`grep $username $namedb | cut -d ';' -f 2`

# print header and copy the solution into the protocol
#
# echo "### Testing submission of $stuname <$username>"
# echo
# echo "### Submitted Program ==="
cat $prgmname
# echo

# Invoke ghci and test the type
#
# ghci $prgmname <ghci.in 2>/dev/null | tail -$msglen >test1.out

# Compile the program and run it
#
# ghc -o inclist $prgmname >test2.out
./inclist >>test2.out

# Check for load errors of the Haskell program
#
grep Failed test1.out >/dev/null
if [ $? = 0 ]; then
   echo "### Errors While Loading Program ==="
cat test1.out
   exit
fi

# Check whether we got the correct result (we do that once for each test)
#
diff -y -W 80 expected1 test1.out >diff1.out
if [ $? != 0 ]; then
   echo "### Testing Inc.hs (type)... FAILED [0 mark(s)]"
cat diff1.out
else
   echo "### Testing Inc.hs (type)... success [1 mark(s)]"
fi
# echo
#
diff -y -W 80 expected2 test2.out >diff2.out
if [ $? != 0 ]; then
   echo "### Testing Inc.hs (incList [1..10])...FAILED [0 mark(s)]"
cat diff2.out
else
   echo "### Testing Inc.hs (incList [1..10])...success [1 mark(s)]"
fi

# clean up
#
rm -f test?.out diff?.out inclist submissions/$1/*.o submissions/$1/*.hi

*Note:* The code for testall is not included as it is part of an exercise.
Chapter 7

Input and Output

“Communicate! It can’t make things any worse.”
—Unknown

So far, all of our programs have been self-contained functions. Given a set of arguments, they returned a result value. We call such functions pure functions and say that they perform computations. They are pure in the sense that they exactly match the concept of a function in math.

However, most programs do more than just computing some value—they interact with their environment. In fact, many programs are solely described by the effect that they have on the environment. For example, the essential point of a text editor is to change files on a computer’s disk, and it is rather difficult to speak about a value that an editor computes. We call program fragments that have effects—such as changing files—actions and say that they perform interactions, i.e., they interact with their environment. Interestingly, even programs such as editors, whose main purpose is to execute interactions, internally perform many computations. For example, if you use the search function of an editor, it searches for a given substring in the list of characters that represent the text file that you are currently editing.

The most essential kinds of action are input and output actions. They either perform textual input from the keyboard and textual output to the screen or read and write files. More complex forms of interactions, such as communication via networks or graphics, are covered in detail in more advanced courses.

7.1 Interpreter Versus Compiler Revisited

Let us assume the following program:

```haskell
module Main where

sumList :: [Int] -> Int
sumList [] = 0
sumList (x:xs) = x + sumList xs

main = print (sumList [1..10])
```

It is a program that can be compiled and executed, as it contains a function main in a module Main. We can compile and execute it as follows:

```
% ghc -o sumlist Main.hs
% ./sumlist
55
```
When the program is started, it executes the function main, which instructs the machine to evaluate \texttt{sumList [1..10]} and, then, to print the result.

Now compare this to using the interpreter GHCi:

```
Main> :load Main.hs
Compiling Main ( Main.hs, interpreted )
Ok, modules loaded: Main.
Main> sumList [1..5]
15
Main>
```

Here we merely state the expression to be evaluated, \texttt{sumList [1..5]}, without explicitly requesting that the result is being printed. Obviously, we need to distinguish between operations performed by our program and operations performed by GHCi, which is merely one of many possible implementations of the programming language Haskell. GHCi, itself a program\footnote{In fact, GHCi itself is also written in Haskell. So, it is a Haskell program that can execute Haskell programs.}, consists of computational operations and interactions. After startup, it waits for a Haskell expression from the keyboard, evaluates it, and finally prints the result. Then, it waits for the next Haskell expression (or a command such as \texttt{:q}). This loop in which GHCi evaluates Haskell expressions and prints the result of the evaluation is called an \textit{eval-print loop}, and it is typical for language interpreters such as GHCi. Here, both reading the Haskell expression and printing the result are I/O actions, whereas the evaluation of the expression is a computation.

In contrast, when we compile a program, there is no \textit{eval-print} loop, and thus, no opportunity to type in commands. Instead, the command that is to be executed on program startup needs to be defined as part of the function \texttt{main}. This function, then, also has to take care of explicitly printing any result. As we have seen, the \texttt{print} function prints its argument, which may be a number, a list, or any other type for which a textual representation is defined. If we want to print the contents of a string, there are two more functions, namely \texttt{putStr} and \texttt{putStrLn}. Both functions output a string, but the latter also starts a new line (which becomes important if there are any further I/O operations following). With this knowledge, we can implement the Hello World! program in Haskell:

```
module Main
where
  main = putStrLn "Hello World!"
```

If you wonder why we have not used \texttt{print} to produce the output, try using \texttt{print} in the program yourself and see what the difference is.

### 7.2 Input/Output Actions

The situation found in the case of a text editor, where the main purpose of a program is interaction, but internally many computations are executed, is typical. In general, we can regard
programs to be composed of an inner computational kernel and an outer interaction shell. Depending on the type of program, the computational (e.g., program computing weather forecasts) or the interaction component (e.g., file copying software) may contribute to the majority of the program code. However, any serious program will contain both components. This decomposition is displayed in Figure 7.2.

Although good programming style requires a clean separation between the computation and the interaction components, most programming languages do not make a clear distinction between these two kinds of program code. Nevertheless, Haskell—being a very clean language—distinguishes between computations and actions using types. If nothing special is said (as Haskell is a functional language), the default is that a type denotes a computation. Hence all the programs that we have discussed up to now denoted only computations, i.e., we have only defined pure functions. For example, the expression "Hello " ++ "World!" of type \texttt{String} denotes a computation (namely that of joining two strings) and the function

\[
(\text{++}) :: [a] \rightarrow [a] \rightarrow [a]
\]

is a pure function—it just computes a result string from two input strings, nothing else.

Input/output (I/O) actions always have a type of the form \texttt{IO a}, where \(a\) is the result of the I/O operation. For example, the expression

\[
\text{putStr "Hello World!"}
\]

is of type \texttt{IO ()}, where the () indicates that no result is returned. The only effect of this operation is to print the string \texttt{Hello World!} onto the screen. On the other hand, the expression \texttt{getLine} has type \texttt{IO String}, as it reads a string from the keyboard and returns it as the action’s result. Now, an example of a program performing both a computation and an interaction is

\[
\text{putStr ("Hello " ++ "World!")}
\]

It concatenates two strings and outputs the result. Note that the overall type is still \texttt{IO ()}, i.e., the computation is not visible in the type. Given our earlier model in which a program consists of a computational core surrounded by an interaction shell, this should not be surprising. All we see from the outside is the interaction shell—the computations are hidden away in the inside of the program.

Now, let us have a look at the type of \texttt{putStr} itself (without argument):
putStr :: String -> IO ()

It clearly is a function. However, the result of this function is not only a value, but an interaction—indeed, it is an interaction that does not return an interesting result itself (as the () indicates). We call it an I/O function or interaction function (as opposed to pure functions such as ++).

### 7.2.1 Combining Actions

So far, we considered individual actions such as putStr and getline. In a program, however, we must combine multiple actions into more complex compound actions. Unlike pure functions, we cannot combine I/O functions merely by nesting them. For example, if we would like to execute getline first, and then putStr on the read string to echo it to the screen,

```
putStr getline                        -- TYPE ERROR
```

will not work. Why? The expression getline has type IO String, whereas putString clearly expects a pure String as its argument—rather than an I/O action that ultimately produces a string. This is what we meant, when we wrote earlier, that Haskell distinguishes between pure computations and interaction by types. Expressions of type String and IO String denote essentially different things and Haskell prevents us from confusing them.

**Order of Execution**

When we consider the evaluation of a pure computation (i.e., without an I/O action), we see that there is often considerable freedom in the order that subcomputations may be performed. For example, consider the following:

\[
(1 + 2) \times (3 + 4) \Rightarrow 3 \times (3 + 4) \Rightarrow 3 \times 7 \Rightarrow 21
\]

\[
(1 + 2) \times (3 + 4) \Rightarrow (1 + 2) \times 7 \Rightarrow 3 \times 7 \Rightarrow 21
\]

The order in which the two additions are performed does not have an impact on the final result. However, it is clear that both additions have to be executed before the multiplication can be executed. This is obvious from the data dependency that exists between the additions and the multiplication. Generally, we say

There exists a data dependency from a computation A to a computation B if and only if A depends on the result of B.

Based on data dependencies, the Haskell system can always determine an appropriate order of execution for pure functions.

The situation is, however, different in the case of I/O actions. Consider

```
putStr "The result is " print 21
print 21 putStr "The result is "
```

```
The result is 21
```

Obviously, the order of executing the putStr and the print statement has a severe impact on the resulting output; however, there is no data dependence between the two. In other words, I/O actions require us to make the order of execution explicit, independent of any data dependencies. Haskell supports this with its do construct.
The do Notation

When we want to combine actions, we use the do notation, which has the general form

\[
\text{do} \quad \langle \text{statement } \#1 \rangle \quad \langle \text{statement } \#2 \rangle \quad \ldots \quad \langle \text{statement } \#n \rangle
\]

where each statement contains either an elementary or a compound I/O action. Whenever we are interested in the result of an action, we can use a statement of the form

\[
v \leftarrow \langle \text{action} \rangle
\]

to bind the result of the action to some variable \( v \).

Now, we can combine `getLine` and `putStr` as follows to each line of text:

\[
\text{do} \quad \text{input} \leftarrow \text{getLine} \\
\text{putStr input}
\]

As `getLine` is of type `IO String`, the variable `input` denotes the string read by `getLine`, i.e., it is of type `String`. Hence, it can—in contrast to `getLine` itself—be passed as an argument to `putStr`.

As usual, we can use a function definition to give an expression a name, so that we can use it repeatedly:

\[
\text{echoLine} :: \text{IO } () \\
\text{echoLine} = \text{do} \\
\hspace{1em} \text{input} \leftarrow \text{getLine} \\
\hspace{1em} \text{putStr input}
\]

Given this definition,

\[
\text{do} \\
\hspace{1em} \text{echoLine} \\
\hspace{2em} \text{echoLine}
\]

reads two lines and echoes them.

7.2.2 Defining Interaction Functions

As with computations, it is useful to define functions that encapsulate useful interaction patterns. For example, we may want to define an interaction function `ask` that given a question (as a string), poses it to the user and waits for an answer in the form of one line of input. Such a function would receive a `String` as input and result in some I/O action that finally produces a string (the user’s answer)—i.e., the result is of type `IO String`. Overall, we can define it as follows:

\[
\text{ask} :: \text{String } \rightarrow \text{IO String} \\
\text{ask question} = \text{do} \hspace{1em} \text{putStrLn question} \\
\hspace{2em} \text{getLine}
\]
We can use `ask` in a program that asks for a person’s name, and then outputs a personalised greeting:

```haskell
main :: IO ()
main = do
  name <- ask "May I ask your name?"
  putStrLn ("Welcome " ++ name ++ ")")
```

Note that although most of this program is concerned with interaction, there is a little computation—namely, the invocations of `++` to construct the welcome message.

### 7.2.3 Showing and Reading Values

The I/O actions `getLine` and `putStrLn` can be used to read and write strings of characters. For values of other types, Haskell provides us with the functions `readLn` and `print`, which read and write a wide range of values.

For example, we can read an integer number as follows, where we prompt the user to first input a number:

```haskell
getNumber :: IO Int
getNumber = do
  putStrLn "Please input a number: 
  readLn
```

Based on this function, we can write a program that reads two numbers, adds them (the computational part), and then outputs the result:

```haskell
main :: IO ()
main = do
  num1 <- getNumber
  num2 <- getNumber
  putStrLn "The sum is 
  print (num1 + num2)
```

Both `readLn` and `print` also work on other types and it is in fact the type declaration that determines what kind of value the program expects to read. For example, by changing the type of `getNumber` to

```haskell
getNumber :: IO Float
```

it expects to read floating point values.

### 7.2.4 Returning a Result From Within an Action

Now assume we want to implement an action `getTwoNumbers` that reads two numbers and returns them as a pair—i.e., it should have the type `IO (Int, Int)`. In contrast to previous routines, we cannot just let the result of the last statement in the `do` expression be the overall result. Instead, we have to combine the results of two I/O actions to form a compound result. We can achieve this using the function `return`, which returns a given value as the result of an I/O operation:

```haskell
getTwoNumbers :: IO (Int, Int)
getTwoNumbers = do
  num1 <- getNumber
  num2 <- getNumber
  return (num1, num2)
```

The function `return` has type

```haskell
return :: a -> IO a
```
7.2.5 Conditionals and Recursion in I/O Actions

As a first recursive interaction, consider the task of printing a list of strings, each on a line of its own.

```haskell
putStringList :: [String] -> IO ()
putStringList [] = return ()
putStringList (str: strs) = do
    putStrLn str
    putStringList strs
```

The function uses pattern-matching as usual, but results in I/O actions rather than just producing a value.

There is, however, another way of doing the job. It exploits the fact that we can put control characters into strings. The control character for indicating that a new line should be started is \n. So, the string "one\ntwo" prints the two on a line of its own. We can use this to convert a list of strings [String] into a single string with appropriate occurrences of \n.

```haskell
unlines :: [String] -> String
unlines [] = ""
unlines (str: strs) = str ++ "\n" ++ unlines strs
```

Which of the two alternatives is better style?

Like purely computational functions, interaction functions can use conditionals and recursion to implement repetitive behaviour. For example, given a function

```haskell
allUpper :: String -> String
allUpper str = map toUpper str
```

that converts all letters in a given string into upper case letters, let us implement a program that reads a line of a text and outputs it after converting all letters in this line to uppercase. The program should repeat the process until an empty line is entered.

After reading a line of text with `getLine`, we check whether it is empty. If so, we return; otherwise, the text is converted to uppercase, printed, and the process repeated.

```haskell
upperLine :: IO ()
upperLine = do
    line <- getLine
    if line == ""
        -- this is the base case
        return ()
    else do
        putStrLn (allUpper line)
        upperLine
```

There are two important points that should be emphasised about the syntax in this program:

1. The `then` and `else` branches of the conditional expression are indented. This is because they belong to the overall statement started by the keyword `if`. As for function declarations, Haskell uses the “further right than the first character of the first word rule” rule to determine how far one statement extends.

2. As there are two actions in the `else` branch, we must use a second `do` expression for the `else` branch.

We can make this into a complete program by adding
main :: IO ()
main = do
    putStrLn "Input of an empty line terminates the program."
    upperLine

7.2.6 Adding Up Numbers

Another example of recursive function, where the recursion depth depends on the input, is getNumList. It reads numbers from the keyboard until 0 is entered:

```haskell
getNumList :: IO [Int]
getNumList = do
    number <- readLn
    if number == 0
       then
           return []
       else do
           numbers <- getNumList
           return ([number] ++ numbers)
```

We can use getNumList in the following program, which sums up a list of numbers:

```haskell
main :: IO ()
main = do
    putStrLn "Input a list of numbers, terminated by 0"
    numbers <- getNumList
    putStr "The sum is 
    print (sum numbers)
```

7.3 File I/O

All interaction so far was about reading from and writing characters to the console. Equally important, programs must be able to access and modify files on permanent storage, e.g., a hard disk. The simplest way of manipulating files is by the following three interaction functions:

```haskell
type FilePath = String

writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile :: FilePath -> IO String
```

Values of type FilePath denote files and have the same syntax as that used with Unix commands, e.g., /home/chak/.emacs. The first of the three functions, writeFile, creates a new file of the given name and writes the given string into the file. If a file by that name already exists, it is overwritten—so be careful which file names you choose. The second function, appendFile, instead of replacing the old contents of the file, simply adds the string to the end of the file, i.e., it extends the file. Finally, readFile reads the contents of the given file and returns it as a string.

Let us use this functionality to write a program that reads the name of a file from the keyboard, then reads the file from disk and outputs it:

```haskell
main :: IO ()
main = do
    putStr "Please input a file name: 
    fname <- getLine
    contents <- readFile fname
    putStr contents
The entire contents of the file are—in the form of a single string—associated with the variable contents, which is finally printed.

7.4 Examples

7.4.1 Showing and Reading Values Revisited

We discussed the functions readLn and print, which can read and write non-string values of various types. These two functions are, in fact, not primitive, but defined in terms of simpler functions. These simpler functions are read and show. They are not actions, rather pure functions. They convert the string representation of a value into that value and vice versa. For example, we can use read to convert the string "42" into the integer value 42; conversely, show applied to 42 will give us the string "42".

This distinction between values and their denotation in string form is subtle, but important. Keep in mind that the string "42" is just a shorthand for the list of characters ['4', '2']. On the other hand, the integer value 42 is just that.

Both functions show and read can be used for various types and, as in the case of other overloaded functions, such as + and ==, a type class constrains the functions that can be converted to strings and back. More precisely, the types of show and read are

\[
\begin{align*}
\text{show} & : \text{Show} a \Rightarrow a \rightarrow \text{String} \\
\text{read} & : \text{Read} a \Rightarrow \text{String} \rightarrow a
\end{align*}
\]

Values of all types that are in the type class Show can be converted to a string using show and, similarly, values of all types in the class Read can be converted from string form into the actual value. All of the types that we discussed so far—except functions—can be converted to string form and back. For example, we have

\[
\begin{align*}
\text{show} 42 & \Rightarrow "42" \\
\text{show} 3.141 & \Rightarrow "3.141" \\
\text{show} \text{True} & \Rightarrow "\text{True}" \\
\text{show} \ [1..3] & \Rightarrow "[1,2,3]" \\
\text{show} \ [(1, \text{True}), (5, \text{False})] & \Rightarrow "[(1,\text{True}),(5,\text{False})]"
\end{align*}
\]

With this knowledge, we can now be more precise about readLn and print. They are defined as follows:

\[
\begin{align*}
\text{print} & \ : \text{Show} a \Rightarrow a \rightarrow \text{IO} () \\
\text{print} \ \text{value} & = \text{putStrLn} \ (\text{show} \ \text{value}) \\
\text{readLn} & \ : \text{Read} a \Rightarrow \text{IO} \ a \\
\text{readLn} & = \ \text{do} \\
& \ \ \ \ \text{string} <- \text{getLine} \\
& \ \ \ \ \text{return} \ (\text{read} \ \text{string})
\end{align*}
\]

We also can use read and show in combination with the file operations. This combination allows us in particular to write complex data structures like lists to a file and then to reread the structures again.

\[
\begin{align*}
\text{main} & \ : \text{IO} () \\
\text{main} & = \ \text{do} \\
& \ \ \ \ \text{putStr} \ "\text{Please input a number: }" \\
& \ \ \ \ \text{n} <- \text{readLn} \\
& \ \ \ \ \text{writeFile} \ "\text{test}" \ (\text{show} \ [1..\text{n}])
\end{align*}
\]
On the input of, say, 10 by the user, this program writes the list \([1,2,3,4,5,6,7,8,9,10]\) to a file named `test`.

### 7.4.2 Supermarket

Let us extend the supermarket example from last week. More precisely, we assume the types

```haskell
type Cents = Int
type PriceList = [(String, Cents)]
type ShoppingList = [(String, Int)]
```

and the function

```haskell
cost :: PriceList -> ShoppingList -> Cents
```

from last week.

In a POS (Point Of Sale) system, we would expect the price list of the supermarket to be stored somewhere on permanent storage, so that the person operating the POS system only has to input the purchased items and their quantity. We need two auxiliary routines: one for reading the price lists from a file and one for performing the conversation with the user during input of the shopping list. The former is as follows:

```haskell
readPriceList :: String -> IO PriceList
readPriceList fname = do
    contents <- readFile fname
    return (read contents)
```

Assuming that file `pricelist` contains

```haskell
[("Mango", 249),
 ("Water, 2l", 199),
 ("Cheese", 399),
 ("Bread", 150)]
```

the invocation of `readPriceList "pricelist"` would result in reading in the value

```haskell
[("Mango",249),("Water, 2l",199),("Cheese",399),("Bread",150)]
```

The function defining the user interaction repeatedly prompts the user for an item and its quantity. If the empty string is entered in place of an item, the process is terminated:

```haskell
readShoppingList :: IO ShoppingList
readShoppingList = do
    putStr "Enter an item: ">
    item <- getLine
    if item == "]"
        then
            return []
        else do
            putStr "Enter quantity: ">
            quantity <- readLn
            items <- readShoppingList
            return ((item, quantity) : items)
```

Given these two interaction functions, the main routine can be implemented as
main :: IO ()
main = do
  prices <- readPriceList "pricelist"
  slist <- readShoppingList
  putStrLn ("The total sum is " ++ show (cost prices slist))

In this example, the decomposition of the program into a computational kernel and an interaction shell is very clear. All the code from last week forms the computational kernel, whereas the new code implements the interaction shell.

7.5 Exercises

1. Why does

   aString :: String
   aString = "This is a string constant"

   have type String, but the function getline have type IO String? Discuss how a complete program consist out of a computational kernel and an interaction shell, where the latter manages the information flow between the computations encoded in the program and the "outside world."

2. Write a program that queries two numbers from the user, and then, prints their sum:

   % ghc -o adNums AddNums.hs
   % ./adNums
   Enter first number: 3
   Enter second number: 4
   7

   Write two versions of the program: one using readLn and the other using getline and read. Discuss the difference.

3. Write a program that accepts a list of numbers (in Haskell’s list syntax) and outputs the sum:

   ghc -o addNumList AddNumList.hs
   % ./addNumList
   Enter a list of numbers: [4,2,7,4,5,0]
   22
Chapter 8

User-defined Data Types

So far, we have used type synonyms to give new names to already existing types—for example,

```haskell
type Colour = String
  type ColourPoint = (Int, Int, Colour)
```

We can verbalise definitions such as the above ones as follows:

- A colour is a string.
- A coloured point consists of two integers and a colour.

This form of verbalising type definitions will prove useful when trying to understand more advanced forms of type definitions (see below).

8.1 Parametric Type Synonyms

Slightly more complicated type synonyms than those we have already considered are

```haskell
type Cents = Int
  type PriceList = [(String, Cents)]
type ShoppingList = [(String, Int)]
```

We can describe them as follows:

- A price list is an (association) list consisting of strings and cents.
- A shopping list is an (association) list consisting of strings and integers.

Obviously, the definitions of PriceList and ShoppingList are similar. So, once again, we can ask whether we cannot avoid repeating the pattern by defining it just once in a more abstract form, and then, instantiating it for the specific cases. Indeed, what we want is something like

An association list is, **given a key and value type**, a list of keys and values.

The interesting part is set in **bold** face. We assert a precondition on the definition of association lists by insisting that we are given a type for each of keys and values. In fact, we can render this in Haskell in a very similar manner:

```haskell
type AssocList key value = [(key, value)]
```

The new type AssocList is here parametrised with two type variables and the right handside of the definition depends on these two variables. More precisely, we call AssocList a type constructor rather than just a type. We can now implement PriceList and ShoppingList in terms of this general definition of association lists:

```haskell
type PriceList = AssocList String Cents
  type ShoppingList = AssocList String Int
8.2 Enumeration Types

Up to now, all of the type definitions only introduced new names for already existing types. However, we often need entirely new data types. Consider, for example, the situation where we would like to define type `Day`, which contains all of the days of the week:

A `Day` is either `Monday` or `Tuesday` or ... or `Sunday`.

We call data types like this one, which come with a fixed number of alternative values, *enumeration types*, and we can define them in Haskell by closely following the verbal description—for example,

```haskell
data Day = Monday
         | Tuesday
         | Wednesday
         | Thursday
         | Friday
         | Saturday
         | Sunday
```

(The values could also be arranged on one line.) Using the new keyword `data`, we introduce a new type—in this case, `Day`—together with all of its possible values, which are called `data constructors`. Data constructors, like types, must begin with an uppercase letter. We can read the vertical bar (`|`) separating the alternatives as "or".

Interestingly, we already know an enumeration type—namely, `Bool`:

```haskell
data Bool = False
           | True
```

A `Boolean` value is `false` or it is `true`. We have also used functions over Boolean values, such as,

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

We have used pattern-matching to implement an exhaustive enumeration of all possible cases. This is what we normally do with enumeration types.

So, suppose we wish to implement a function `nextDay` that computes, for any day, the following one. We can implement it as

```haskell
nextDay :: Day -> Day
nextDay Monday = Tuesday
nextDay Tuesday = Wednesday
nextDay Wednesday = Thursday
nextDay Thursday = Friday
nextDay Friday = Saturday
nextDay Saturday = Sunday
nextDay Sunday = Monday
```

Another example function is for checking whether a `Day` is on the weekend:

```haskell
isWeekend :: Day -> Bool
isWeekend Saturday = True
isWeekend Sunday = True
isWeekend Monday = False
isWeekend Tuesday = False
isWeekend Wednesday = False
isWeekend Thursday = False
isWeekend Friday = False
```
8.3 Parametrised Alternatives

It, however, seems quite wasteful to include all the cases for Days that are not on the weekend. We observe that there is a general pattern to these alternatives, where only the argument to \texttt{isWeekend} varies. The general way to abstract in such a situation is to introduce a variable, e.g.,

\begin{verbatim}
isWeekend :: Day \rightarrow \text{Bool}
isWeekend Saturday = True
isWeekend Sunday = True
isWeekend day = False
\end{verbatim}

Since the alternate definitions for a given function are checked top to bottom, the last \emph{catch-all} case will only be reached if the argument is neither Saturday nor Sunday.

Another alternative is to use an underscore, which is a generic variable that can be used in a pattern if we do not need its value:

\begin{verbatim}
isWeekend :: Day \rightarrow \text{Bool}
isWeekend Saturday = True
isWeekend Sunday = True
isWeekend _ = False
\end{verbatim}

Using our newly acquired knowledge, we revisit the following definitions of coloured points:

\begin{verbatim}
type Colour = String
type ColourPoint = (Int, Int, Colour)
\end{verbatim}

The use of Strings here to encode colours is potentially problematic. There is no guarantee that only strings that represent colour names are used. Using enumeration types, we can fix this problem if we are dealing with a fixed set of colour names—for example,

\begin{verbatim}
data Colour = White
           | Black
           | Red
           | Green
           | Blue
\end{verbatim}

It should be noted that in many systems, the set of colours is not fixed (or at least it is huge), in which case, we may still have to resort to strings to represent colour names.

### 8.3 Parametrised Alternatives

Unfortunately, pure enumerations are often not enough. Let us consider the geometric forms in Figure 8.1. Using an enumeration, we could define
data Shape = Circle
    | Square
    | Triangle

This would allow us to distinguish between these three shapes, but it still leaves a lot of information unspecified. For example, we might know that a given shape is a circle, but we don’t know what kind of circle. For that, we would also have to know the position of the centre of the circle, which might be specified as

    type Point = (Int, Int)

In addition, we need to know the radius of the circle. We can achieve this by giving arguments to the data constructors of the enumeration (we usually do not call the result an enumeration):

    data Shape = Circle Point Int -- center & radius
                | Square Point Point -- center & vector of one side
                | Triangle Point Point Point -- center & vector of two sides

Given such a definition, we can form values, such as \texttt{Circle (0, 0) 4}. Unfortunately, we cannot just evaluate such an expression, as this leads (in GHCi) to

\[
\text{Shapes} \triangleright \text{Circle (0, 0) 4}
\]

\textbf{No instance for `Show Shape'}

\textbf{arising from use of `print' at <No locn>}

This message tells us that the system is not able to apply the \texttt{show} function (which converts a value into a printable string) to the given expression. We have already discussed how GHCi completes such an expression into a complete Haskell program (see page 97), which would contain a definition such as

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

\[
\text{main = print (Circle (0, 0) 4)}
\]

and that the type of \texttt{print}—as outlined earlier—is

\[
\text{print :: Show a => a -> IO ()}
\]

Given this, the problem of a lack of a definition of \texttt{show} for \texttt{Shape} becomes clear.

We could now extend the definition of \texttt{show} to cover the new type, but luckily, Haskell allows us to automatically derive a definition when we define the data type:

\[
\text{data Shape = Circle Point Int}
\]

\[
\text{deriving (Show)}
\]

The same works for other type classes, such as \texttt{Eq}.

As an exercise, implement the following functions

---

\[
\text{-- check if the given shape is round}
\]

\[
\text{-- isRound :: Shape -> Bool}
\]

\[
\text{-- yield the position of a shape}
\]

\[
\text{-- posOfShape :: Shape -> Point}
\]
8.4 Recursive Data Type Definitions

In the same way that we can define recursive functions, we can define recursive data types. A recursive function makes reference to itself in its own definition. Thus, a recursive data type refers to itself in its own definition. To see how this works, let us consider one of our early recursive function definitions:

```hs
sumInts :: [Int] -> Int
sumInts [] = 0
sumInts (x:xs) = x + sumInts xs
```

When applied to a list of integers \([x_1, x_2, \ldots, x_n]\), the following computation is performed:

\[
\text{sum} \ (x_1 : (x_2 : (\cdots : (x_n : [])\cdots))) \Rightarrow x_1 + (x_2 + (\cdots + (x_n + 0)\cdots))
\]

8.4.1 Example: Defining Lists of Int

The recursive list notation using cons (:) and nil ([]) makes the match between the recursion in the data structure and the recursion in the computation explicit.

In fact, we can regard cons (:) and nil ([]) as being data constructors defined in a definition such as the following:

```hs
data [Int] = []          -- base alternative
          | Int : [Int]  -- recursive alternative
```

Strictly speaking, this definition would not be valid, because of the special syntax used for lists—i.e., [Int] and [] are not valid identifiers. However, we can achieve exactly the same effect by a corresponding definition using valid identifiers:
data IntList = Nil  
  | Cons Int IntList

Using IntList, the list [1, 2, 3], which corresponds to 1 : (2 : (3 : [])), would be denoted Cons 1 (Cons 2 (Cons 3 Nil)). For use with IntList, the function sumInts would be denoted as

\[
\begin{align*}
\text{sumIntList} & : \text{IntList} \rightarrow \text{Int} \\
\text{sumIntList Nil} & = 0 \\
\text{sumIntList (Cons x xs)} & = x + \text{sumIntList xs}
\end{align*}
\]

8.4.2 Polymorphic Lists

Since the definition of IntList forces the elements of a list to be integers, we need similar definitions if we want to have lists containing elements of different types. For example,

data StringList = NilSL  
  | ConsSL String StringList

for lists of strings. Unfortunately, there is a serious disadvantage to defining new types for each new element type. Consider writing a function computing the length of a list:

\[
\begin{align*}
\text{lengthIntList} & : \text{IntList} \rightarrow \text{Int} \\
\text{lengthIntList Nil} & = 0 \\
\text{lengthIntList (Cons _ xs)} & = 1 + \text{lengthIntList xs}
\end{align*}
\]

\[
\begin{align*}
\text{lengthStringList} & : \text{StringList} \rightarrow \text{Int} \\
\text{lengthStringList NilSL} & = 0 \\
\text{lengthStringList (ConsSL _ xs)} & = 1 + \text{lengthStringList xs}
\end{align*}
\]

Although the definitions for lists of integers and lists of strings are very similar, we still have to write out the definition twice. In general, if we use a whole range of different lists, we would end up with a huge amount of code duplication.

As before in similar situations, abstraction can help us generalise the definition of lists, to avoid duplicating code. We replace the element type by a type variable and add this variable to the description of the type:

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

With this definition, lists of integers are denoted by List Int and lists of strings by List String. Now, we can write a generic definition of the length function:

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

This function works for all sorts of lists and is called a polymorphic function. It employs the same principle as the list functions—such as length—from the Prelude.

8.5 Intermezzo: Seeing the Meaning for All the Syntax

What is the point of defining our own version of lists if lists are already provided by Haskell? It is not that the self-defined lists are any more powerful than the predefined ones. It is rather that we want to understand the inner workings in the implementation of lists—and in fact, the Haskell system implements lists in much the same way as we did it.
8.5 Intermezzo: Seeing the Meaning for All the Syntax

8.5.1 Looking Behind the Scenes

Generally, when you learn to program or when you learn about a new programming language or other software system, you usually already start out with a complex environment. For example, in our case, Haskell and the GHCi system are a complex system. In such a situation, the first learning step is to learn to use the system, rather than understanding how the system works. This includes learning about pre-defined functionality—as we learned to use lists in the beginning of this course. Only after you have gained an understanding of the use of the available functionality will you start to learn about how this pre-defined functionality itself is implemented—in our case, learning how lists can actually be defined using more elementary constructions, such as data definitions.

Software systems generally are constructed in many layers. Each of these layers introduces a new level of abstraction that hides many of the details of underlying layers. Given the huge complexity of modern computer and software systems, this is the only approach that allows us to keep these systems manageable. For example, consider what happens from the moment that you type a key on a computer’s keyboard until the moment the corresponding character appears in an editor window. There is a complex system of electronic circuits involved: the keyboard transmits a code representing the key over a serial link to the motherboard, which has some I/O circuitry that notifies the main processor about the keystroke. On top of the electronics implementing the motherboard runs the operating system of your computer, which will intercept this notification and select the user process that gets the key stroke. Finally, that user process will use a pre-defined set of graphics routines within the computer’s windowing system to properly draw an image corresponding to the keystroke into a window. However, usually when we type on the keyboard, we don’t consider this whole process. Even the person implementing the editor software will usually not consider the operation on the level of electronic circuits, or even the operating system. Instead, she will use library routines that allow her to read characters from the keyboard and print to the screen. So, these layers of abstraction help the programmer of the editor get the job done without being bogged down in details.

However, as programmers we often have to switch between different levels of abstraction. Sometimes we can work on higher levels of abstraction and sometimes we need to consider low-level stuff—for example, when implementing a new device driver for an operating system. For the learning process this means that we first study a system on a relatively high level of abstraction, but then, at some point, break these abstractions open and look inside.

8.5.2 Syntax Versus Semantics

The syntax of an object (like a list) is its representation—i.e., in Haskell for a list containing the values 1, 2, and 3 the following sequence of characters: \([1, 2, 3]\). On the other hand, the semantics of an object is its meaning—i.e., for the syntax \([1, 2, 3]\) the statement “that this is a list containing the three numbers 1, 2, and 3”. For a given semantics, we can have more than one syntactic representation. For example, \([1, 2, 3]\) corresponds to \(1:2:3:[]\). These are different representations for the same thing.

When you learn to program you have to learn to see the meaning behind the syntax. It is quite natural to initially have problems in separating syntax and meaning. Nevertheless, this distinction is very important since otherwise we would not be able to transfer our knowledge of programming from one programming language to another. Interestingly, programming languages also allow us to invent new forms of representing meaning. For example, given

```haskell
data IntList = Nil
             | Cons Int IntList
```

what is the difference between

- \([1, 2, 3]\),
The difference is only one of syntax. It is very much like translating an English sentence into another language, such as Chinese, German or Japanese. If we write down a given sentence in all these languages, the appearance (i.e., the syntax) is very different; nevertheless, the meaning may well be the same.

8.6 Type Classes for User-Defined Data Types

We discussed earlier that as part of the definition of data types, we can obtain canonical implementations for some type classes. These are called derivable type classes and there are some restrictions on the types for which they can be obtained.

8.6.1 Eq and Ord

Only types that belong to the type class Eq can be compared using (==) and (/=). Moreover, only on types in Ord do we have an ordering relation, which is needed for (>), (>=), and so on. We can derive Eq and Ord for any newly defined data types provided that all arguments to all data constructors of the new type are themselves contained in Eq and Ord, respectively. For example, in our definition of Day, we can use

```haskell
data Day = Monday |
          | Tuesday |
          | Wednesday |
          | Thursday |
          | Friday |
          | Saturday |
          | Sunday |
          deriving (Eq, Ord) -- derive both
```

This allows us to compare days, as in Monday == Monday or Monday < Wednesday. The ordering is defined such that constructors that occur earlier in the definition are regarded to be smaller than those that appear later.

8.6.2 Enum

A type class that we did not discuss so far is Enum. This type class determines which types can be used in expression such as [a..b] and [a, b..c]. We can derive Enum for any newly defined enumeration type. In other words, we can define

```haskell
data Day = Monday |
          | Tuesday |
          | Wednesday |
          | Thursday |
          | Friday |
          | Saturday |
          | Sunday |
          deriving (Eq, Ord, Enum) -- derive three instances
```

which makes [Monday..Friday] evaluates to the list [Monday, Tuesday, Wednesday, Thursday, Friday]. This allows us to write a function like isWorkingDay quite concisely as
isWorkingDay :: Day -> Bool
isWorkingDay day = day `elem` [Monday..Friday]

Note that this definition makes use of both that Day is an instance of Eq and that it is an instance of Enum, as the type of elem is

elem :: Eq a => a -> [a] -> Bool

8.6.3 Show and Read

As with the previously discussed type classes, we can also derive instances for the classes Show and Read, which govern the use of the functions show and read, respectively:

show :: Show a => a -> String
read :: Read a => String -> a

As we have seen already, these functions are in particular important as the I/O functions print and readLn make use of show and read, respectively. So we have

print :: Show a => a -> IO () -- output a file
readLn :: Read a => IO a -- read a value from stdin

As a consequence, values of any data type that belongs to both Show and Read can be written to and read from the console or a file.

8.7 Exercises

1. Online book sellers usually also sell video tapes and CDs. So, they need to maintain a product database that is flexible enough to deal with all three types of goods. In the following, we will define a module `BookVideoCD` that contains the necessary data types to deal with such a database.

(a) First, define a data type `Product` that can represent any of the three types of goods. For each item, the following additional information has to be maintained:

- Book: the title and author
- Video: the title
- CD: the title, artist, and number of tracks

Define appropriate type synonyms for titles, authors, and artists.

(b) Extract the title information from a product: Implement a function `getTitle` that, given a product, extracts its title. (Note that all three types of products contain title information.)

(c) Extract titles from a product list: Now, use `getTitle` to implement `getTitles`, which, given a list of products, produces a list with all titles. There are two ways to implement this function: with an explicit recursive definition or by using the higher-order function `map`. Discuss both variants and compare them.

2. Partial functions revisited: As a preparation, recall the term partial function that we discussed earlier. So far, we raised an error whenever a partial function was applied to an argument on which it wasn’t defined. An alternative would be to return a special value (let’s call it Nothing) whenever we previously raised an error. The advantage of that is that we can actually test for the error in the calling function, rather than aborting the whole program.

Define a data type `Maybe a` that can be used for this purpose.
Safe div: What’s the result of evaluating \( 10 \div 0 \)? Implement a safe division function `safeDiv :: Int -> Int -> Maybe Int`. 10 \( 'safeDiv' \) 0 returns `Nothing`, rather than aborting with an error message.
Chapter 9

Formal Reasoning

“We have long felt that programmers working on real-time control systems should have the privilege of taking the first ride on the operational prototype. In other words, if your code implements the life support systems on the space shuttle, then you get to be launched into space and debug any last minute glitches personally. This would surely bring a whole new focus to product quality.”

—Peter van der Linden, Expert C Programming—Deep C Secrets

Correctness of software becomes an increasingly important topic as software is used more and more in systems that directly affect the well-being and safety of human beings. Airplanes and power plants are standard examples of applications that crucially depend on properly working software; however, as digital technology penetrates every aspect of our lives, computers also affect everyday devices, such as automobiles. In fact, even in cases where human life is not at stake, bugs in computer systems can cause tremendous financial damage as has been demonstrated by the crash of the Ariane 5 rocket and the FDIV debacle that surrounded one of Intel’s chip designs.

This situation, combined with Edsger Dijkstra’s famous quote “Program testing can be used to show the presence of bugs, but never their absence”, leads us to a serious problem: how can we ensure that a given piece of software works flawlessly? As testing alone cannot provide a guarantee that there are no more bugs hidden in a program, we have to look to more rigorous approaches to establishing the correctness of software. Generally, in science and engineering, mathematical proofs and calculations are the standard approach to establishing the truth of a statement beyond any reasonable doubt. So, it should not come as any surprise that the same techniques can be used to reason about the correctness of computer programs.

Investigating a rigorous mathematical treatment of programs also helps us understand the meaning of programs more precisely—an issue that is of central importance for computing as a science. In particular, for studying the semantics of programming languages, a precise mathematical understanding of programming languages, hence of programs, is crucial. This understanding, important in its own right, can help compilers for programming languages generate better code (i.e., faster code using less memory) when there are provably correct rules by which they can transform a program into a more optimised form. Clearly, a compiler should only use transformation rules that are absolutely guaranteed to work in all possible cases.

9.1 Calculating with Programs

Given the two arithmetic expressions \((b + b + 2 + a)/2\) and \(a + b\), anybody with basic knowledge of algebra is able to conclude that they are the same. Proceeding formally, we can establish the
equality as follows:

\[
\frac{b + b + 2 \times a}{2} = \frac{2 \times b + 2 \times a}{2} = \frac{x \times y}{y} = x
\]

\[
b + a
\]

\[
a + b
\]

Interestingly, we can treat functional programs in much the same way. This becomes clear immediately when considering the following two function definitions:

```haskell
foo :: Int -> Int -> Int
foo a b = (b + b + 2 * a) / 2

foo' :: Int -> Int -> Int
foo' a b = a + b
```

The previous calculation established that \( \frac{b + b + 2 \times a}{2} = a + b \), and thus, \( \text{foo } a \ b = \text{foo' } a \ b \) for all integer values of \( a \) and \( b \). In other words, we can say that both functions are the same.

Generally, we can easily extend the above style of reasoning to functions that call other user-defined functions in their body by inlining the used function during calculation. For example, given the slightly modified functions

```haskell
foo'' :: Int -> Int -> Int
foo'' a c = (c + 2 * a) / 2

foo' :: Int -> Int -> Int
foo' a b = a + b
```

we can establish \( \text{foo'' } a \ (b + b) = \text{foo' } a \ b \) by calculating as follows

\[
\text{foo'' } a \ (b + b)
\]

\[
= \{\text{Inlining of foo''}\}
\]

\[
(b + b + 2 \times a) / 2
\]

\[
= \{(b + b + 2 \times a) / 2 = a + b \text{ as demonstrated before}\}
\]

\[
a + b
\]

\[
= \{\text{By the definition of foo'}\}
\]

\[
\text{foo' } a \ b
\]

In the last step, we have to use a form of “reverse inlining” to get from a complex expression to a function application representing the expression. Generally, we call inlining of a function definition unfolding and the reverse process folding. We will later see that we can use this technique even in the presence of recursion to prove the equality of two functions.

How does all this help us with the initial goal of establishing the correctness of programs? One of the most useful cases where calculations are of use is when we have an easy to understand but inefficient program, and a heavily optimised but difficult to understand program achieving the
same result. If we can prove that they do indeed compute the same result for all (relevant) inputs, we can base any argument about the correctness on the easy to understand program, but use the more efficient program in the actual implementation. In this case, we usually call the easy to understand program the \textit{specification}; the efficient program is one possible \textit{implementation} of the specification.

In more sophisticated cases, an equation such as

\[ \sqrt{x} \times \sqrt{x} = x \]

may serve as the specification by determining the meaning of \texttt{sqrt} without actually providing an executable function definition.

\subsection{9.1.1 Case Distinction}

Earlier we used stepwise evaluation of Haskell programs to compute the result of Haskell expressions. The main difference between that and our current situation is that in the stepwise evaluation scenario, the expression usually did not contain any free variables. However, when we consider the possible equality of functions, we generally have expressions containing free variables. Without concrete values for some arguments, the treatment of conditional expressions, guards, and pattern matching becomes more challenging.

Let us assume that we have a function \texttt{mulp \(x, y\)}, for which we know that

\[ \texttt{mulp \(x, y\)} = x \times y \text{, for all } x \geq 0 \] (Premise)

The restriction of the domain of \(x\) may appear to be random, but as we shall see later, it makes perfect sense for a particular form of definition of multiplication. Furthermore, we have

\begin{verbatim}
mul :: Int -> Int -> Int
mul x y | x < 0 = -(mulp (-x) y) -- mul.1
        | otherwise = mulp x y -- mul.2
\end{verbatim}

How can we now establish that \texttt{mul \(x, y\)} = \(x \times y\) for all integer values \(x\)?

\textbf{Proof.} We split the proof into two parts, matching the structure of the function definition.

- Case \((x < 0)\):

  \[
  \texttt{mul} \ x \ y \\
  = \{ \text{As } x < 0, \text{ with } \texttt{mul.1} \} \\
  - (\texttt{mulp} (-x) y) \\
  = \{ \text{As } -x > 0 \text{ with (Premise)} \} \\
  - ((-x) \ast y) \\
  = \{ \text{Arithmetic} \} \\
  x \ast y
  \]

- Case \((x \geq 0)\):

  \[
  \texttt{mul} \ x \ y \\
  = \{ \text{As } x \geq 0, \text{ with } \texttt{mul.2} \} \\
  \texttt{mulp} \ x \ y \\
  = \{ \text{As } x > 0 \text{ with (Premise)} \} \\
  x \ast y
  \]
The two cases cover the whole domain of possible values, so we have that \( \text{mul}\ x\ y = x * y \) for all integer values \( x \).

Note the notation \( \text{mul}.1 \) and \( \text{mul}.2 \) to distinguish between the two equations in the definition of \( \text{mul} \). Henceforth, we will use this notation to distinguish equations without explicitly annotating the labels as comments of the function definition.

### 9.1.2 Induction

Probably the most interesting language construct to handle in equational proofs is recursion. It presents us with a number of questions, such as how far shall we unfold a recursive function? And it also implies the presence of a case distinction in every recursive step—otherwise, the recursion would never terminate due to the lack of a base case. To prove statements about recursive programs, we need a proof structure that resembles recursion. Such a structure is provided by proofs by induction.

A proof by induction proceeds by inspecting various values of an induction variable, which is not unlike the argument that is being stepwise processed by a recursive function. Furthermore, it has a base case—just like a recursive function—which we usually prove by making use of the base case of the recursion. The recursive step in a recursive function is matched by the induction step (or stepping case) in the proof. The induction step is established with the help of an induction hypothesis, which basically states that the property that we would like to prove holds for a certain value of the induction variable. It is the purpose of the induction step to establish that the property then also holds when we advance the induction variable by one step (e.g., one more list element or an incremented numeral).

More formally, induction over the natural numbers works as follows: to show that some property \( P(n) \) holds for any natural number \( n \geq 0 \), it is sufficient to show the following:

1. **Base case** (0): Show that \( P(0) \) holds.
2. **Induction** \((n + 1)\): Show that if \( P(n) \) holds, then \( P(n + 1) \) also holds.

The tricky part is the second case, where we assume that \( P(n) \)—the induction hypothesis—holds, and then prove \( P(n + 1) \). As in the case of recursive definitions, these two cases are sufficient for an inductive proof over natural numbers, because any natural number is either zero or can be expressed as the successor of another natural number. We explicitly prove that the property is valid for zero; this and the second case tells us that it also holds for 1, from which we can deduce it holds for 2, and so on.

As an example, consider the following recursive function.

```haskell
mulp :: Int -> Int -> Int
mulp x y | x == 0 = 0
          | otherwise = y + mulp (x - 1) y
```

It implements multiplication recursively by summing up its second argument as often as the first one indicates. We would like to prove that

\[
\text{mulp}\ x\ y = x * y
\]

for cases where \( x \geq 0 \).

**Proof.** The proof proceeds by induction over the first argument \( (x) \) to \( \text{mulp} \), which is also the argument used for keeping track of the depth of the recursion in the definition of \( \text{mulp} \). So, we prove \( P(x) \), which is \( \text{mulp}\ x\ y = x * y \) where \( x \geq 0 \). The base case of the induction coincides with the base case of the recursion definition. It is the purpose of the inductive step to establish \( P(n + 1) \) under the condition that \( P(n) \) has already been established.
1. **Base case** (0):

   \[
   \text{mulp } 0 \ y \\
   = \{\text{mulp.1}\} \\
   0 \\
   = \{0 \text{ is the neutral element of } \cdot\} \\
   0 * y
   \]

2. **Induction** (n + 1):

   \[
   \text{mulp } (n + 1) \ y \\
   = \{\text{mulp.2 as } n + 1 \neq 0\} \\
   y + \text{mulp } (n + 1 - 1) \ y \\
   = \{\text{Arithmetic}\} \\
   y + \text{mulp } n \ y \\
   = \{\text{Induction hypothesis}\} \\
   y + n * y \\
   = \{\text{Arithmetic}\} \\
   1 * y + n * y \\
   = \{\text{Arithmetic}\} \\
   (1 + n) * y \\
   = \{\text{Arithmetic}\} \\
   (n + 1) * y
   \]

We have now established the premise that we used in the proof about the function \text{mul}, which means that the two functions \text{mul} and \text{mulp} together implement multiplication over the full range of integer numbers.

### 9.1.3 Example

The following Haskell code is a straightforward implementation of the exponentiation function:

```haskell
(^) :: Float -> Int -> Float
x^0 = 1
x^n = x * x^(n - 1)
```

Furthermore, we know from math that

\[x^{(m+n)} = x^m * x^n\]

so we would expect that the above definition of \(^\) satisfies this law. More precisely, we would like to prove that for all natural numbers \(n,m \geq 0\) and for all floating-point values \(x\), we have

\[x^{(n + m)} = (x^n) * (x^m)\]

**Proof.** The proof proceeds by induction over \(n\), i.e., our induction hypothesis is

\[P(n) \equiv x^{(n + m)} = (x^n) * (x^m), n, m \geq 0\]
1. **Base case** \((0)\): We must show that \(P(0)\) holds, i.e.,

\[
x^{-}(m + 0) = (x^{-}m) * (x^{-}0)
\]

We show this by independently reducing both sides of the equality. First, for the left side, we have

\[
x^{-}(m + 0)
= \{0 \text{ is the neutral element of } +\}
= x^{-}m
\]

Second, for the right side, we have

\[
(x^{-}m) * (x^{-}0)
= \{^*.1\}
= (x^{-}m) * 1
= \{1 \text{ is the neutral element of } *\}
= x^{-}m
\]

Both sides are equal as they simplify to the same result.

2. **Induction** \((n + 1)\): We now have to show that \(P(n + 1)\) holds, i.e.,

\[
x^{-}(m + (n + 1)) = (x^{-}m) * (x^{-}(n + 1))
\]

We show this again by reducing both sides of the equality independently. First, for the left side, we have

\[
x^{-}(m + (n + 1))
= \{\text{Associativity of } +\}
= x^{-}((m + n) + 1)
= \{^*.2\}
= x * x^{-}(m + n)
= \{\text{Induction hypothesis}\}
= x * x^{-}m * x^{-}n
\]

Second, for the right side, we have

\[
(x^{-}m) * (x^{-}(n + 1))
= \{^*.2\}
= (x^{-}m) * (x * x^{-}n))
= \{\text{Commutativity and associativity of } *\}
= x * x^{-}m * x^{-}n
\]

Again, both sides simplify to the same expression, which proves the induction step and completes the proof.

Please note that simplifying both sides of an equation is often a useful proof strategy.
### 9.1.4 Structural Induction

Similar to inductive proofs over the natural numbers, it is possible to apply inductive proofs to functions over all forms of algebraic data types. Generally, this form of induction is called *structural induction*—as it follows the structure of a data structure, such as a list. Below, we shall now have a look at proofs by induction over list processing functions.

We can state the induction principle for structural induction over lists as follows: to show that some property $P(xs)$ holds for any finite list $xs^2$, it is sufficient to show the following:

1. **Base case** ($[]$): Show that $P(\text{[]})$ holds.
2. **Induction** ($x:xs$): Show that, if $P(xs)$ holds, then $P(x:xs)$ holds also.

Obviously, the structure of such a proof is very similar to the proofs over natural numbers that we considered so far. The reasoning that leads us to believe that the induction principle for structural induction over lists is valid also follows the same argument that we briefly discussed in the context of natural numbers.

As an example, consider the following two function definitions:

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

(++): [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

We would expect that the length of two lists is *invariant* under concatenation or, more formally,

$$\text{length} \ (xs \ ++ \ ys) = \text{length} \ xs \ ++ \ \text{length} \ ys$$

By now it should be obvious that we are very often interested in proving *invariants*, i.e., properties that do not change in the presence of a given computation. They form a central concept in any framework for reasoning about programs. Can you figure out why?

**Proof.** We proceed by induction over $xs$—try doing it over $ys$ to see if the choice of induction variable matters. So, we have

$$P(xs) \equiv \text{length} \ (xs \ ++ \ ys) = \text{length} \ xs \ ++ \ \text{length} \ ys$$

1. **Base case** ($[]$): We have to show that $P(\text{[]})$ holds, i.e.,

   $$\text{length} \ (\text{[]} \ ++ \ ys) = \text{length} \ [\text{[]} \ ++ \ \text{length} \ ys$$

   For the left side, we have
   
   $$\text{length} \ (\text{[]} \ ++ \ ys)$$
   
   $$\equiv \{++ .1\}$$
   
   $$\text{length} \ ys$$

   For the right side, we have

   $$\text{length} \ [\text{[]} \ ++ \ \text{length} \ ys$$

---

$^2$As we didn’t discuss computations over infinite data structures in this course, we will not consider them on our journey into formal reasoning.
Formal Reasoning

\[ \begin{align*}
= & \{\text{length.1}\} \\
= & 0 + \text{length } ys \\
= & \{\text{Arithmetic}\} \\
= & \text{length } ys \\
\end{align*} \]

2. **Induction** \((x:xs)\): We have to show that \(P(x:xs)\) holds, i.e.,

\[ \text{length } ((x:xs) ++ ys) = \text{length } (x:xs) ++ \text{length } ys \]

For the left side, we have

\[ \begin{align*}
\text{length } ((x:xs) ++ ys) & = \{++ .2\} \\
& = \text{length } (x:(xs ++ ys)) \\
& = \{\text{length.2}\} \\
& = 1 + \text{length } (xs ++ ys) \\
& = \{\text{Induction hypothesis}\} \\
& = 1 + \text{length } xs + \text{length } ys \\
\end{align*} \]

For the right side, we have

\[ \begin{align*}
\text{length } (x:xs) ++ \text{length } ys & = \{\text{length.2}\} \\
& = 1 + \text{length } xs + \text{length } ys \\
\end{align*} \]

\[ \blacksquare \]

9.2 Working with Lemmata and Generalisation

Unfortunately, proofs do not always proceed as smoothly as the ones, we have discussed so far—largely because most functions have a more complicated structure than the ones that we have examined so far. It is easy to see that the following naive definition of \textit{reverse} is in effect reversing the order of its argument list:

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

However, it also has the disadvantage that it is quite inefficient. To see this, consider the definition of \textit{++}:

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

Obviously, the number of recursive calls to \textit{++} needed to concatenate two lists is proportional to the length of the first list \((xs)\). Now, as \textit{reverse} invokes \textit{++ once for every recursive step}, of which there are as many as elements in the list that we want to reverse, it should become clear that overall we have \(n + (n - 1) + (n - 2) + \cdots + 2 + 1\) calls to \(;\), where \(n\) is the number of elements in the list that we \textit{reverse}. In other words, the number of calls to \(;\) is \(\sum_{i=1}^{n} i\), which is in the order of \(n^2\)—it is \(\frac{n(n+1)}{2}\), to be precise. This seems a bit excessive for a function that is supposed to reverse the elements in a list of length \(n\).

In fact, there is a more efficient definition of \textit{reverse}—let us call it \textit{freverse for fast reverse}—that only takes order-\(n\) function calls for a list of length \(n\):
freverse :: [a] -> [a]
freverse xs = fr xs []
where
    fr [] ys = ys
    fr (x:xs) ys = fr xs (x:ys)

However, this definition is not only faster, but also less intuitive. Hence, it makes sense to prove
the quality of reverse and freverse. So, let’s try to prove by structural induction the property

\[ P(xs) \equiv reverse \; xs = freverse \; xs \]

for all finite lists \( xs \).

**Proof.**

1. **Base case (\([\;]\))**: We must show that \( P([\;]) \) holds, i.e.,

\[
reverse \; [] = freverse \; []
\]

For the left side, we have

\[
reverse \; [] = reverse \; [\;] = f \text{reverse} \; [\;] \]

For the right side, we have

\[
freverse \; [] = frev \; [\;] \]

So, the base case is established.

2. **Induction \((x:xs)\)**: We must show that \( P(x:xs) \) holds, i.e.,

\[
reverse \; (x:xs) = freverse \; (x:xs)
\]

For the left side, we have

\[
reverse \; (x:xs) = reverse \; xs ++ [x] = \text{Induction hypothesis} = freverse \; xs ++ [x] = freverse \; (x:xs)
\]

After that, there does not seem to be anything else we can do.

For the right side, we have

\[
freverse \; (x:xs)
\]
\[ \text{formal reasoning} = \{\text{freverse}\} \]
\[ \text{fr} \ (x:xs) \ [] \]
\[ = \{\text{fr.2}\} \]
\[ \text{fr} \ xs \ [x] \]

Unfortunately, we are stuck here! Intuitively, \( \text{fr} \ xs \ [] \ ++ \ [x] \) and \( \text{fr} \ xs \ [x] \) appear to be the same, but there does not seem to be a direct way to formally prove it.

At this point, we might think about showing

\[ \text{fr} \ xs \ [] \ ++ \ [x] = \text{fr} \ xs \ [x] \]  \hspace{1cm} (9.1) \]

by induction over \( xs \)—but this does not work out either.

In such a situation, it is often a useful strategy to take a step back and consider whether there is a useful property of \( fr \) that we can prove and that would imply Equation 9.1. Indeed, if we were to establish the more general

\[ \text{fr} \ xs \ ys \ ++ \ zs = \text{fr} \ xs \ (ys \ ++ \ zs) \]  \hspace{1cm} (9.2) \]

then, because \( [] \ ++ \ [x] = [x] \), we would have established Equation 9.1, which in turn would be all we needed to finish the above induction step. Luckily, we can indeed show that Equation 9.2 holds and, therefore, we will now prove a lemma that establishes exactly that. We proceed by induction over \( xs \) in Equation 9.2.

1. **Base case** (\( [] \)): We have to show that \( Q([],) \) holds, i.e.,

\[ \text{fr} \ [] \ ys \ ++ \ zs = \text{fr} \ [] \ (ys \ ++ \ zs) \]

For the left side, we have

\[ \text{fr} \ [] \ ys \ ++ \ zs \]
\[ = \{\text{fr.1}\} \]
\[ ys \ ++ \ zs \]

For the right side, we have

\[ \text{fr} \ [] \ (ys \ ++ \ zs) \]
\[ = \{\text{fr.1}\} \]
\[ ys \ ++ \ zs \]

2. **Induction** (\( x:xs \)): We have to show that \( Q(x:xs) \) holds, i.e.,

\[ \text{fr} \ (x:xs) \ ys \ ++ \ zs = \text{fr} \ (x:xs) \ (ys \ ++ \ zs) \]

For the left side, we have

\[ \text{fr} \ (x:xs) \ ys \ ++ \ zs \]
\[ = \{\text{fr.2}\} \]
\[ \text{fr} \ xs \ (x:ys) \ ++ \ zs \]
\[ = \{\text{Induction hypothesis}\} \]
\[ \text{fr} \ xs \ ((x:ys) \ ++ \ zs) \]
\[ = \{++ .2\} \]
\[ \text{fr} \ xs \ (x: (ys \ ++ \ zs) \]


It might not be immediately obvious that we can apply the induction hypothesis here. However, keep in mind that induction is only over \( xs \), i.e., the first argument of \( fr \). The fact that the hypothesis does only hold for \( Q(xs) \) does not constrain any other arguments to \( fr \).

For the right side, we have

\[
fr (x:xs) (ys ++ zs)
\]

\[
\begin{align*}
= & \{fr.2\} \\
fr xs (x:(ys ++ zs))
\end{align*}
\]

The use of lemmata to support a proof and proving properties that are more general than the ones we actually need are essential techniques needed for many proofs.

### 9.3 Program Derivation

As discussed earlier, formal reasoning can help eradicate bugs when we have a specification (which is inefficient, but easy to understand) of a program as well as an implementation (which is tricky, but efficient) and we can prove that both implement the same function. However, this postmortem application of formal techniques, where we already have the implementation and only check whether it is correct, has a number of drawbacks. First, it might be difficult to find a good implementation and we might want the specification to guide us towards a good solution. Second, if during proving the equality of specification and implementation, we discover a problem, we have to redo the implementation, and then check again in the hope that we caught the first bug and didn’t introduce a new one.

Therefore, it seems more convenient to use proof techniques to \textit{derive} an efficient implementation from a specification—not unlike the optimiser in a compiler that generates efficient code by repeatedly applying a set of optimising program transformations to the original input program. As an example of a program transformation, let us consider the two auxiliary functions \texttt{take n xs} and \texttt{drop n xs}, which take and drop the first \( n \) elements of a list \( xs \):

\begin{align*}
take & :: \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
take \ 0 \ xs & = [] \\
take \ n \ [] & = [] \\
take \ n \ (x:xs) & = x : \text{take} \ (n-1) \ xs
\end{align*}

\begin{align*}
drop & :: \text{Int} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
drop \ 0 \ xs & = xs \\
drop \ n \ [] & = [] \\
drop \ n \ (x:xs) & = \text{drop} \ (n-1) \ xs
\end{align*}

They can be used to define a function \texttt{splitAt} that splits a list at a given element into two parts:

\begin{align*}
\text{splitAt} & :: \text{Int} \rightarrow \text{[a]} \rightarrow (\text{[a]}, \text{[a]}) \\
\text{splitAt} \ n \ xs & = (\text{take} \ n \ xs, \text{drop} \ n \ xs)
\end{align*}

Provided that \texttt{take} and \texttt{drop} are correct, there is little doubt that \texttt{splitAt} does what it is supposed to do. However, both \texttt{take} and \texttt{drop} traverse the list \( xs \) once, although we can imagine that it must be possible to implement \texttt{splitAt} such that it traverses the list only once. So, let’s consider the above definition as our specification for \texttt{splitAt}.

We could now go off and try to find a more efficient implementation of \texttt{splitAt}, and then perform a postmortem proof to try to show that it is functionally equivalent to the above definition. However, this time we will instead directly derive the efficient implementation from
the specification. As we will use equational reasoning to perform the derivation, the result will already be guaranteed to be correct.

Interestingly, we have two base cases: one where the first argument to \texttt{splitAt} is 0 and one where the second argument is the empty list.

1. **First base case** (\texttt{splitAt} 0 n): Starting from \texttt{splitAt} 0 n, reduce the expression to something simpler using equational reasoning steps:

   \[
   \text{splitAt} \ 0 \ \textit{xs} \\
   = \text{splitAt} \\
   \quad \text{(take 0 xs, drop 0 xs)} \\
   = \text{take.1} \\
   \quad (\textit{[]}, \text{drop 0 xs)} \\
   = \text{drop.1} \\
   \quad (\textit{[]}, \textit{xs})
   \]

   This gives us the following equation

   \[
   \text{splitAt} \ 0 \ \textit{xs} = (\textit{[]}, \textit{xs})
   \]

2. **Second base case** (\texttt{splitAt} n \textit{[]}): We proceed in a similar way as in the first base case:

   \[
   \text{splitAt} \ n \ \textit{[]} \\
   = \text{splitAt} \\
   \quad \text{(take n [], drop n [])} \\
   = \text{take.2} \\
   \quad (\textit{[]}, \text{drop n []}) \\
   = \text{drop.2} \\
   \quad (\textit{[]}, \textit{[]})
   \]

   This gives us the second equation

   \[
   \text{splitAt} \ n \ \textit{[]} = (\textit{[]}, \textit{[]})
   \]

3. **Recursive case** (\texttt{splitAt} n (x:xs)): The most interesting case is where we handle a non-empty list. Note that since we allow a general \textit{n} here as a first argument to \texttt{splitAt}, we can assume in the following derivation that \textit{n} is *not* equal 0, because of the first base case.

   \[
   \text{splitAt} \ n \ (x:xs) \\
   = \text{splitAt} \\
   \quad \text{(take n (x:xs), drop n (x:xs))} \\
   = \text{take.3, } n \neq 0 \\
   \quad (x : \text{take (n-1) xs, drop n (x:xs)}) \\
   = \text{drop.3, } n \neq 0 \\
   \quad (x : \text{take (n-1) xs, drop (n-1) xs}) \\
   = \{\text{Rearrange expression}\} \\
   \quad (x : \text{ys, zs}) \\
   \quad \text{where (ys, zs) = (take (n-1) xs, drop (n-1) xs)} \\
   = \{\text{fold } \texttt{splitAt}\}
   \]
This gives us the recursive equation

\[
\text{splitAt } n \ (x:xs) = (x:ys, zs)
\]

where

\[
(ys, zs) = \text{splitAt } (n - 1) \ xs
\]

The most tricky part in the derivation is to bring everything together into the right form, so that we can apply a fold step for \(\text{splitAt}\), so that we get a proper recursive equation. Overall, we get the following definition for \(\text{splitAt}\):

\[
\text{splitAt } :: \text{Int} \to [\text{a}] \to ([\text{a}], [\text{a}])
\]

\[
\text{splitAt } 0 \ xs = ([], xs)
\]

\[
\text{splitAt } n \ [] = ([], [])
\]

\[
\text{splitAt } n \ (x:xs) = (x:ys, zs)
\]

where

\[
(ys, zs) = \text{splitAt } (n - 1) \ xs
\]

9.4 Exercises

1. Consider the following two function definitions, which compute the absolute value and signum of an integer, respectively:

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

\[
\text{abs } k \mid k \geq 0 = k
\]

\[
\text{abs } k \mid \text{otherwise} = -k
\]

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

\[
\text{signum } k \mid k > 0 = 1
\]

\[
\mid k == 0 = 0
\]

\[
\mid k < 0 = -1
\]

Prove that, for all integer values \(n\), we have

\[
\text{abs } n \times \text{signum } n = n
\]

*Hint:* First, think about which cases you have to distinguish in the proof. Then, prove each case individually and ensure that all cases together cover all possible input values.

2. Induction: Given

\[
(++) :: [\text{a}] \to [\text{a}] \to [\text{a}]
\]

\[
[] \ ++ \ ys = ys
\]

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

prove that \((++)\) is associative, i.e.,

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

3. What is the induction principle and why does it work?

4. What is an induction hypothesis?

5. Given the following functions:
mulNats :: Int -> Int
mulNats 0 = 1
mulNats n = n * mulNats (n - 1)

natList :: Int -> [Int]
natList 0 = []
natList n = n : natList (n - 1)

prodList :: [Int] -> Int
prodList [] = 1
prodList (x:xs) = x * prodList xs

prove that, for every natural number n, we have

mulNats n = prodList (natList n)
Chapter 10

Modularisation & Program Decomposition

The source code for programs often grows to hundreds of thousands or even millions of lines of code. It should be clear that it is unfeasible to maintain the program code for projects of this size in a single file. Therefore, we have to split the code over multiple files. Each of these files is usually called a module. Modules, however, are not only used to reduce the file size, but also to structure the code into logically related components and to regulate the usage of the entities (types, functions, and so on) defined in one module and used in other modules.

With the help of modules, large software projects can be broken into smaller pieces—this is called system decomposition. The goal is to bring together strongly related bits of code (e.g., function and type definitions) of a large program into self-contained, partially closed units. As a result, many of the details of each of these units are relevant only within, but not across, units. This makes it easier to develop these units largely in isolation and, for example, to split a project over sub-groups of a project team. Modules are a programming language mechanism that support program decomposition by enforcing the isolation of various program components from each other. Overall, modules provide the following benefits:

- Simplification of program design and a tool for structuring
- Simultaneous development and testing of sub-components (multiple teams)
- Parts of the program can be understood in isolation
- It is easier to isolate bugs
- It is easier to change parts of the program without adverse effects on other parts
- Program code can be re-used (libraries).

In this section, we will have a look at some basic design principles, which are to a large degree independent of the programming language we are working with. All modern general purpose languages support decomposition, but the terminology may differ; instead of modules, the units may be called, for example, classes or components. After discussing the module support in Haskell, we will then look at some concrete examples of modular design.

10.1 Modular Design

It is easy to see that huge programs should be split into separate modules, but it is all but trivial to decide how this should be done. In fact, the mapping of functionality to modules (or components or classes) is an important step in the software design process and has a huge influence on the quality and maintainability of the resulting system. There are some guidelines you should keep in mind when doing the design of a larger program:
A module should have a clearly defined purpose that can be understood by looking at it in isolation.

The effect of design decisions that are likely to change should be confined to a single module. For example, for an initial prototype, performance might not be a big issue, so a fast (in terms of development costs), straightforward implementation can be chosen and later on replaced by a more efficient implementation. This should affect only a minimal part of the system, ideally only a single module.

Each module should correspond to a single abstraction.

Each module should be easy to test.

Hardware and application dependencies should be restricted to a small number of modules.

There are two central concepts related to a modular design:

- **Cohesion**: The relationship of definitions inside one module. We can identify a number of different types of cohesion, for example:
  
  - **Functional Cohesion** (good): Functions providing similar functionality should be in the same module. For example, all functions for I/O might be in a single module, although they don’t depend on each other; or mathematical functions, like \( \sin \), \( \cos \) and so on.
  
  - **Informational Cohesion** (good): Different functions working on the same data structures should be placed into the same module, for example, all list manipulating functions.
  
  - **Logical Cohesion** (should be avoided): Different implementations of the same functionality. For example, if a system has different user interfaces, they should be placed in separate modules.
  
  - **Coincidental Cohesion** (should be avoided): Definitions end up in the same module for no specific reason. This happens more often than one would think — for example, some definitions are put into a module “just for now”, because there is no obvious module they fit in, and then stay there.

- **Coupling**: The relationship between different modules.

  In Haskell, we say a module *imports* from another module if it uses definitions of the latter. A module *exports* definitions if it allows other modules to use it. We will see in the next section how this is done.

  - **Data Coupling** (good): In a functional programming language, functions can only communicate by explicitly passing values—that is, the result of one function may constitute the input for another function. For this reason, coupling in functional languages is usually restricted to “good” coupling, namely data coupling. Of course, this does not mean that you cannot have a bad design in a functional language!

In general, we can say that a good, modular design shows high cohesion, that is, all the definitions in one module are closely related, and low coupling, that is, the number of dependencies between modules should be low. In particular, *circular dependencies*, where a module A uses definitions of module B, and vice versa, are often an indication of a poor design. Circular dependencies may also be indirect, that is module A import from module B, which, in turn, imports from module C.
10.1 Modular Design

10.1.1 Interface Design

The details of how a module presents itself to the rest of the system are captured in the *interface* of the module. The interface specifies which functions and types of a module are visible to and can be used by other modules. The part of the module that implements the interface is usually called its *implementation*. The documentation of the types and functions in the interface should contain all the information that is needed by a programmer to use these types and functions *without* looking at the details of how they are actually implemented. Moreover, the module interface should not make overly strong assumptions about the actual implementation of the interface. It should be clear that modules are yet another mechanism for implementing *abstractions*.

A sizable program can easily consist of hundreds of modules. As a result, keeping an overview of the relationship between modules becomes a challenging task. Often hierarchical diagrams are used, with an arrow from module \( A \) to module \( B \) if \( A \) imports \( B \). These diagrams visualise the *module hierarchy* of a program. Generally, when \( A \) imports \( B \), we call \( A \) a *client* of \( B \) and \( B \) a *provider* to \( A \).

10.1.2 Exporting and Importing in Haskell

The module name is followed by the export list, which enumerates all exported types and functions—i.e., it determines the interface of the module. This is followed by a sequence of *import* statements, which determine which modules are imported.

In Haskell an export list either enumerates all exported names or, if the export list is omitted, all top-level definitions are exported (i.e., all definitions except local ones done with `where` and `let`). Import statements come in several variations:

- If we want to import all entities (all top level definitions, types and functions) from the interface of a given module, we just write
  
  ```haskell
  import List
  ```

- If we, however, want to selectively import only a subset of the exported entities, we write
  
  ```haskell
  import List (transpose)
  ```
  
  where we enumerate all the imported entities (this is usually better style as it documents the module relationship in more detail).

- Finally, in the case where we want to import all but a selected set of entities, we write
  
  ```haskell
  import List hiding (transpose)
  ```
  
  where we enumerate all entities that we do *not* want to import. This is especially helpful when we want to avoid importing some of the entities provided by the *Prelude*, as it is very large and enumerating all the entities that we do want to import would be rather awkward:

  ```haskell
  import Prelude hiding (lines)
  ```

Data types can be exported and imported with or without their type constructors. For example, `IntList` is defined as

```haskell
data IntList = Nil
  | Cons Int IntList
```
By writing \texttt{module Example (Day)} we export \textit{only} the name of the type \texttt{(IntList)}, but the type constructors are not visible outside of module \texttt{Example}. The type constructors can be exported by writing: \texttt{module Example (IntList(..))}, or by explicitly mentioning those data constructors that should be visible to other modules: \texttt{module Example (IntList(Nil))} (only the constructor \texttt{Nil} is exported). Similarly, it is possible to selectively import the data constructors of a user defined type. It makes only sense to hide the data constructors if the module provides a reasonable set of operations on the data type.

There are two special modules: \texttt{Prelude} and \texttt{Main}. The \texttt{Prelude} contains all types and functions that are predefined in Haskell. It is \textit{implicitly} imported into any Haskell module—thus, all the entities from the interface of the \texttt{Prelude} are generally available. Furthermore, every \textit{complete} Haskell program must contain a module \texttt{Main}, which in turn must always contain an action \texttt{main}. This is the action executed as the program begins.

Finally, we use notation \texttt{Bar.bar} to denote that an entity \texttt{foo} is from module \texttt{Bar}. Other examples of the use of this notation would be \texttt{Prelude.lines}, \texttt{Main.main}, and \texttt{List.nub}.

\subsection{The Supermarket Example Revisited}
Let us revisit the supermarket example in the light of our new knowledge about modules. It consists of two modules. The module \texttt{Bill} contains the computational kernel of the program, i.e., the type definitions of price lists, shopping lists, and the functions used to compute the cost of a shopping list. However, it does not export all of these functions, but only the most essential function, namely \texttt{cost}:

\begin{verbatim}
module Bill (  
    Cents, PriceList, ShoppingList, cost  
) where

import Prelude hiding (lookup)

type Cents = Int
type PriceList = [(String, Cents)]
type ShoppingList = [(String, Int)]

cost :: PriceList -> ShoppingList -> Cents  
...
\end{verbatim}

The module imports the \texttt{Prelude} without \texttt{lookup}, because it defined its own \texttt{lookup} function—\texttt{for the details of the implementation part, see the lecture notes from previous weeks}.

The module \texttt{Bill} is then imported by the \texttt{Main} module of the program, which implements the interaction shell.

\begin{verbatim}
module Main

where

import Bill (Cents, PriceList, ShoppingList, cost)

readPriceList :: String -> IO PriceList
...

readShoppingList :: IO ShoppingList
...

main :: IO ()
\end{verbatim}
Here all entities from the interface of Bill are imported. It is still better style to explicitly enumerate all these types and functions, because in larger programs (with many more imports) it can become difficult to keep an overview of the origin of each function.

10.1.4 The Main Module

Any complete Haskell program must have exactly one module called Main. All other modules of the program are directly or indirectly imported by Main. The module Main must export an I/O action called main, which must be of type IO (). When program execution begins, action Main.main is executed.

How does this requirement fit together with the eval-print loop in GHCi? Assume that we have a module

```haskell
module Average
where

average :: Int -> Int -> Int
average x y = x + y `div` 2
```

After we have loaded the module Average into GHCi, if we enter the expression `average 3 4`, we can interpret GHCi’s behaviour as follows: GHCi implicitly constructs a Main module of the form

```haskell
module Main
where

import Average

main :: IO
main = do
  print (average 3 4)
```

Then, it just runs this Main module. It uses the function Average.average in the main I/O action to compute `average 3 4` and output the result.

10.2 Scopes and Visibility of Names

From what we have discussed so far, it should have become apparent that names are essential in programming languages. Typically there are several different categories of names (e.g., for types and for functions) and there are usually precise rules as to where the different kinds of names can be used. We will now discuss in detail these rules and the underlying concepts.

10.2.1 Categories of Names

In programming languages, the names of functions, types, and so on are usually referred to as identifiers. We can categorise Haskell identifiers in various ways. First, there are alphanumeric and symbolic names. Alphanumeric names, such as `toUpper` or `xs'`, can contain alphabetic characters, digits, underscores (\_) and apostrophes (’). They are always used in prefix mode. As
for symbolic names, such as `++` and `->`, they are formed from symbols and are always used in infix mode.

Furthermore, we can distinguish alphanumeric identifiers depending on whether they start with a lowercase or an uppercase letter. All lowercase identifiers are used for variables names. These may be naming functions and values or type variables. As for uppercase identifiers, they are always used for type names (often also called type constructors), type classes, or data constructors, which we will discuss later in this course. As an example, consider the following piece of code:

```haskell
average :: Int -> Int -> Int
average x y = x + y `div` 2
```

The four identifiers `average`, `x`, `y`, and `div` denote variables. This is clear, since they start with a lowercase letter. Both `average` and `div` are function variables, i.e., they refer to functions. The two variables `x` and `y` are argument variables—they represent the arguments passed to the function `average`. Of note is the infix use of `div`: despite being an alphanumeric identifier, it is used infix instead of prefix (i.e., it is used as `a `div` b` instead of `div a b`). This is possible by enclosing it in back quotes (` `), i.e., writing ``div` instead of just `div`.

The identifier `Int` denotes a type name, so, it begins with an uppercase letter. Moreover, we have `+` and `->` as two examples of infix symbol-based identifiers.

### 10.2.2 Definition Versus Use

Occurrences of identifiers in program code are usually categorised as defining and use occurrences. Whenever an identifier occurs in a defining position, (some of) its meaning is fixed. In contrast, use occurrences refer to the meaning assigned to the identifier in its defining occurrences. In the following program all defining occurrences of identifiers are underlined—all the other occurrences are use occurrences.

```haskell
type Colour = String
type ColourPoint = (Int, Int, Colour)

origin :: Colour -> ColourPoint
origin colour = (0, 0, colour)

distance :: ColourPoint -> ColourPoint -> Float
distance (x1, y1, colour1) (x2, y2, colour2) =
  sqrt (fromInt (dx * dx + dy * dy))
  where
dx = x2 - x1
dy = y2 - y1

distanceFromOrigin :: ColourPoint -> Float
distanceFromOrigin point = distance point (origin "white")
```

There are a couple of interesting observations to be made about this code. In Haskell, occurrences of identifiers to the left of an equals sign (=) are usually defining occurrences (an exception is for identifiers occurring in guards). Use occurrences of an identifier can be found either before or after the corresponding defining occurrence and function names are defined both in their type signature and in equations.

### 10.2.3 The Scope of Definitions

Depending on where the defining occurrence of an identifier might be, the identifier will only be usable in a restricted part of the program. This part, where an identifier can be used with a
specific meaning, is called the scope of the identifier. Instead of saying that an identifier can be used at some point in a program, we also say that an identifier is visible at that point. The scope of many identifiers is quite restricted and it is useful to know where it is visible to avoid program errors.

Generally, we distinguish between identifiers having a global scope (we call these global identifiers and regard them as being defined by global definitions) and identifiers having local scope (they are, thus, local identifiers defined in a local definition). A global identifier is visible throughout the whole module in which it is defined—however, it is not necessarily visible in other modules (that depends on the modules’ export and import lists). On the other hand, local identifiers are only visible in part of the program—for example, within one function definition.

Consider the definition of distance:

```haskell
distance :: ColourPoint -> ColourPoint -> Float
distance (x1, y1, colour1) (x2, y2, colour2) =
  sqrt (fromInt (dx * dx + dy * dy))
where
  dx = x2 - x1
  dy = y2 - y1
```

The definition of distance itself is global (all global definitions in Haskell modules start in the leftmost column). However, the definitions of the argument variables x1, x2, and so on as well as the definitions of dx and dy are local. These variables cannot be used outside of the definition of distance.

It is possible that the scope of local variables is even more restricted. Consider for example, the following (contrived) example:

```haskell
foo x y = let a = x + y
          in
          let b = x * y
          in
          a + b
```

Variables that are defined using the let construct, such as a and b, are visible from throughout the whole let expression, but not outside of the defining let. In other words, in the example, both a and b are visible in the lines below their defining occurrence—i.e., a is visible in the whole body of foo, whereas b is only visible in part of it. As a consequence, we could be writing

```haskell
foo x y = let a = x + b -- ERROR!
          in
          let b = x * y
          in
          a + b
```

However,

```haskell
foo x y = let a = x + b -- ERROR!
          in
          let b = x * y
          in
          a + b
```

is not legal, since b is not visible in the first line.

Equipped with this knowledge, we can actually understand one of the more subtle differences between where clauses and let expressions. Compare the following two definitions
The definition of \texttt{foo} is legal, whereas the definition of \texttt{bar} is not. The reason for this is that the scope of a definition in a \texttt{where} clause is the whole function definition, whereas the identifier defined in the \texttt{let} clause of \texttt{bar} is only visible \textit{after} its definition, i.e., only to the right of the first equals sign (=).

Also interesting is the scope of identifiers defined by the binding construct in the \texttt{do} notation. For example, in

\begin{verbatim}
main :: IO ()
main = do
  prices <- readPriceList "pricelist"
  slist <- readShoppingList
  putStrLn ("The total sum is " ++ show (cost prices slist))
\end{verbatim}

the two variables \texttt{prices} and \texttt{slist} are defined within the \texttt{do} expression and they are only visible in the remaining lines of the \textit{same} \texttt{do} expression \textit{following} the definition. In particular, we cannot write

\begin{verbatim}
main :: IO ()
main = do
  prices <- readPriceList "pricelist"
  slist <- readShoppingList
  putStrLn ("The total sum is " ++ show result)
  where
    result = cost prices slist -- ERROR!
\end{verbatim}

The \texttt{where} is outside of the \texttt{do} expression, and thus, \texttt{prices} and \texttt{slist} are not visible in the \texttt{where} clause.

### 10.2.4 Visibility and Modules

Only global identifiers can be exported from a module; when they are imported into another module, they will have global scope. It is impossible to have local imports. The visibility of identifiers across modules is fully specified by the export list of the provider and the import list of the client.

### 10.3 Exercises

Consider a simple application to manage your telephone list. The initial implementation has a text-based user interface, and allows you to enter, update, lookup and delete entries. And entry consists of a telephone number associated with the name of the person or the business.
1. Design the application in a way that it is easy to improve the program later on, for example, by replacing the text-based user interface with a graphical interface, or by extending the data type that models the entries to contain additional information, like address and so forth.

2. Specify the interface of each of the modules and draw a diagram to visualise the module hierarchy. Describe briefly the purpose of each of the exported data types and functions.
Chapter 11

Work Complexity

“The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence.”
—Fred Brooks, Jr.

So far, we discussed the quality of programs in terms of their design, clarity and structure. Now, we will turn our attention to an important factor that we previously only discussed superficially, namely the performance of a program. Program performance has a couple of aspects of which the two most notable ones are a program’s running time and its memory consumption. In the following, we restrict ourselves to the discussion of a program’s running time as the most obvious and most commonly discussed performance measure.

To get an idea for the problems involved, let us measure the running time of the following sorting program whose structure we already discussed previously:

```haskell
isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insertSorted x (isort xs)

insertSorted :: Ord a => a -> [a] -> [a]
insertSorted e [] = [e] -- base case 1
insertSorted e (x:xs)
  | e < x = e : x : xs -- base case 2
  | otherwise = x : insertSorted e xs -- stepping case
```

Figure 11.1 displays the running times of `isort` on lists of size 1000 to 9000 elements in seconds, run in GHCi on a 800MHz processor. The three curves show the times for a list that is already sorted (dashed line at the bottom), a list with random elements (middle), and a list in reverse order (dotted line at the top). Looking at the curve in the middle, we can see that something strange is happening. Compare the running times for 2000 and 4000 elements, or 3000 and 6000 elements: the running times increase in each case by roughly a factor of four, although the size of the input lists is only doubled. If the curve continues in this way, it means that for very long input lists, our sorting program takes far too much time: for about 250,000 elements it will run for almost two hours!

We could argue that we simply have to compile the program instead of using the interpreter GHCi, use a faster processor, a better compiler, or a more low-level language like C or even assembler code. Each of these measures might lead to a speed up, but it would not solve the basic problem, that the running time grows quadratically with the input size. In other words, we would have to use a processor that is four times as fast just to be able to run it on lists twice the length in the same time.
Obviously, something is seriously wrong with the way we are sorting the elements in the isort function—there has to be a better way solve this problem. Before we look at more efficient sorting algorithms, however, let us see if it is really necessary to actually measure the running time of a program to reason about its performance. Running benchmarks is tedious and time consuming, and in a more complex program, we still would not know which functions exactly are problematic. It should be possible to get some idea of how an increase of the input size influences the running times of a program (or a specific function) by analysing the program code.

### 11.1 Complexity Analysis

By analysing the program code, can we find out how long it takes to evaluate a specific expression, such as for example, insertSorted 5 \([1,2,6]\)? Well, we cannot predict how many microseconds it will take, since this very much depends on the machine we are running the program on. However, we know that when the machine evaluates an expression, it repeatedly applies the rules specified in the function definitions until the result is obtained. Such an application of a definition is called a reduction step (or evaluation step). For our example expression, the following three reduction steps are required:

```
insertSorted 5 \([1,2,6]\)
= \{\text{insertSorted.3 (stepping case)}\}
1 : (insertSorted 5 \([2,6]\)
= \{\text{insertSorted.3 (stepping case)}\}
1 : (2 : (insertSorted 5 \([6]\)))
= \{\text{insertSorted.2, (base case 2)}\}
1 : 2 : 5 : \([6]\)
```

In addition, in each iteration, one step is necessary to evaluate the boolean expression in the guard:

```
5 > 1 = \{(>\}\) 5 > 2 = \{(>\}\) 5 > 6 = \{(>\}\)
False False True
```

So, overall it takes six reduction steps to determine the result. Independent of how long exactly it takes the machine to execute one reduction step, a program that executes fewer steps will be faster. From now on, instead of measuring the running times of programs, we will reason in terms of reduction steps about efficiency.
Now, it is not very interesting to determine the number of reduction steps for a specific list, we need to know how many steps it takes to insert an element into an arbitrary list of a given length \( n \). In other words, can we determine the function \( T \), where \( T_{\text{insertSorted}}(n) \) is equal to the number of reduction steps required to insert an element \( x \) into a list of length \( n \)? If the input list in our example is empty, that is \( n = 0 \), it takes only a single reduction step to evaluate the expression:

\[
\text{insertSorted} \ x \ [ \ ]
\]
\[
= \text{insertSorted.1}
\]
\[
[ x ]
\]

Otherwise, the number of steps depends on the values in the list: if the first element in the list is less than \( x \), we need only two steps:

\[
\text{insertSorted} \ x \ (y:ys)
\]
\[
= \{ \text{insertSorted.2}\} \quad x < y
\]
\[
(x:y:ys)
\]
\[
= \{ (\langle \rangle, \text{assumption}) \}
\]
\[
\text{True}
\]

Otherwise, we have

\[
\text{insertSorted} \ x \ (y:ys)
\]
\[
= \{ \text{insertSorted.3}\} \quad x < y
\]
\[
(y : (\text{insertSorted} \ x \ ys))
\]
\[
= \{ (\langle \rangle, \text{assumption}) \}
\]
\[
\text{False}
\]

That is, we need two steps plus the number of steps it takes to insert \( x \) in the tail \( ys \) of the original list \( y:ys \). Since the number of steps depends on the values of \( x \) and \( y \), we cannot determine the exact number steps without actually looking at the values of the list. However, we can calculate the number of steps required in the best case \( T_{\text{best}} \), in the worst case \( T_{\text{worst}} \), and on average \( T_{\text{av}} \).

For \text{insertSorted}, we clearly have \( T_{\text{best}} \text{insertSorted}(n) = 2 \).

In the worst case, however, we have to traverse the entire list and insert the element at the end, so we have \( n \) recursive calls to \text{insertSorted}, each requiring two steps; thus, \( T_{\text{worst}} \text{insertSorted}(n) = 2 * n \). On average, half of the elements of the list will be smaller than the element that has to be inserted, so \( T_{\text{av}} \text{insertSorted}(n) = 2 * (n/2) = n \). This means that if we double the size of a list, it will take on average twice as long to insert an element into a list, which is what we would expect.

Now we can use these results to determine the function \( T_{\text{isort}} \) for \text{isort}. Again, we consider three cases: the best case, the worst case, and the average case.

**Best Case:** If \( n = 0 \), it takes only one single step:

\[
isort \ [ \ ]
\]
\[
= \{ \text{isort.1}\}
\]
\[
[ \ ]
\]

Otherwise, we have

\[
isort \ (x:xs)
\]
\[
= \{ \text{isort.2}\}
\]
\[
isort \text{Sorted} \ x \ (\text{isort} \ xs)
\]

That is, it takes one step plus the number of steps it takes in the best case to insert an element into a list of length \( n - 1 \) plus the number of steps to sort a list of length \( n - 1 \). So, we end up with a recursive equation for \( T_{\text{best}} \text{isort} \):

\[
T_{\text{best}} \text{isort}(0) = 1
\]
\[
T_{\text{best}} \text{isort}(n) = T_{\text{best}} \text{insertSorted}(n - 1) + T_{\text{best}} \text{isort}(n - 1) + 1 = 4 + T_{\text{best}} \text{isort}(n - 1)
\]
which is equivalent to the closed form \( T_{\text{best}}^{\text{isort}}(n) = 4n + 1 \). So, in the best case (here, when the input list is already sorted), the program takes for, say 1000 elements, 4001 steps, for 2000 elements 8001, that is, the time only grows linear with the number of elements to be sorted. If we compare this to the runtime results in Figure 11.1, we can see that, indeed, the curve for the sorted list shows linear behaviour.

**Worst Case:** The equations for the worst case are similar:

\[
T_{\text{isort}}^{\text{worst}}(0) = 1 \\
T_{\text{isort}}^{\text{worst}}(n) = 1 + T_{\text{insertSorted}}^{\text{worst}}(n - 1) + T_{\text{isort}}^{\text{worst}}(n - 1) \\
= 2 + 2(n - 1) + T_{\text{isort}}^{\text{worst}}(n - 1) \\
= 2n - 1 + T_{\text{isort}}^{\text{worst}}(n - 1)
\]

For this case, it is more difficult to find a closed form for \( T_{\text{isort}}^{\text{worst}} \). We have:

\[
T_{\text{isort}}^{\text{worst}}(n) = 2n - 1 + (2(n - 1) - 1) + (2(n - 2) - 1) \ldots + (2 - 1) + 1 \\
= \left( \sum_{i=1}^{n} (2i - 1) \right) + 1 \\
= 2 \sum_{i=1}^{n} i - \left( \sum_{i=1}^{n} 1 \right) + 1 \\
= 2 \sum_{i=1}^{n} i - n + 1
\]

And, as we know that

\[
\sum_{i=1}^{n} i = n \times (n + 1)/2
\]

we can derive the closed form

\[
T_{\text{isort}}^{\text{worst}}(n) = 2n \times (n + 1)/2 - n + 1 \\
= n^2 + n - n + 1 \\
= n^2 + 1
\]

According to this result, \texttt{isort} requires, in the worst case (that is, if the input list is sorted in reverse order) about more than one million steps to sort a list of only 1000 elements!

**Average Case:** The equations for the average case look similar to the worst case:

\[
T_{\text{isort}}^{\text{av}}(0) = 1 \\
T_{\text{isort}}^{\text{av}}(n) = 1 + T_{\text{insertSorted}}^{\text{av}}(n - 1) + T_{\text{isort}}^{\text{av}}(n - 1) \\
= 1 + (n - 1) + T_{\text{isort}}^{\text{av}}(n - 1) \\
= n + T_{\text{isort}}^{\text{av}}(n - 1)
\]

Proceeding as previously, we get

\[
T_{\text{isort}}^{\text{av}}(n) = n + (n - 1) + (n - 2) + \ldots + 1 \\
= \sum_{i=1}^{n} i \\
= 0.5 \times n^2 + 0.5 \times n
\]

So, even in the average case, our program is quite slow. It takes about 750,000 steps to sort a list of 1000 elements.
11.1 Complexity Analysis

11.1.1 Some More Examples

Before introducing the next concept in dealing with program complexity, let us consider a couple of examples.

**Complexity of tail**

The function `tail` which is defined as

\[
\text{tail} :: \mathbb{[a]} \rightarrow \mathbb{[a]}
\]

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

evaluates in exactly one step, no matter how long the list is, so we have \( T_{\text{tail}}(n) = 1 \).

**Complexity of take**

Determining the complexity of the function `take` is more involved:

\[
\text{take} :: \text{Int} \rightarrow \mathbb{[a]} \rightarrow \mathbb{[a]}
\]

\[
\text{take} 0 \text{ } \text{xs} = \text{ []}
\]

\[
\text{take} \text{ } \text{n} \text{ } \text{[]} = \text{ []}
\]

\[
\text{take} \text{ } \text{n} \text{ } (x:xs) = (x : (\text{take} \text{ } (\text{n}-1) \text{ } \text{xs}))
\]

The number of steps necessary to evaluate `take` depends on two parameters: the length of the list \( m \) and the number of elements \( n \) to be retrieved from the list.

\[
T_{\text{take}}(0, m) = 1
\]

\[
T_{\text{take}}(n, 0) = 1
\]

\[
T_{\text{take}}(n, m) = 1 + T_{\text{take}}(n - 1, m - 1)
\]

\[
= \min(m, n)
\]

**Complexity of substring**

The function `substring` returns `True` if the string that is passed as the first argument is a substring of the second argument:

\[
\text{substring} :: \text{String} \rightarrow \text{String} \rightarrow \text{Bool}
\]

\[
\text{substring} \text{ } \text{""} \text{ } \text{str} = \text{ True}
\]

\[
\text{substring} \text{ } \text{str} \text{ } \text{""} = \text{ False}
\]

\[
\text{substring} \text{ } \text{str1} \text{ } \text{str2} =
\]

\[
(\text{prefix} \text{ } \text{str1} \text{ } \text{str2}) \text{ } || \text{ } (\text{substring} \text{ } \text{str1} \text{ } \text{tail} \text{ } \text{str2})
\]

\[
\text{prefix} :: \text{String} \rightarrow \text{String} \rightarrow \text{Bool}
\]

\[
\text{prefix} \text{ } \text{""} \text{ } \text{str} = \text{ True}
\]

\[
\text{prefix} \text{ } \text{str} \text{ } \text{""} = \text{ False}
\]

\[
\text{prefix} \text{ } (s1:s1) \text{ } (s2:str2)
\]

\[
\mid s1 == s2 = \text{ prefix} \text{ } \text{str1} \text{ } \text{str2}
\]

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

Let \( n \) be the length of first string, \( m \) the length of the second. How many reduction steps does the evaluation require for both functions in the worst case?

\[
T_{\text{worst}}^{\text{prefix}}(n, 0) = T_{\text{worst}}^{\text{prefix}}(0, m) = 1
\]

\[
T_{\text{worst}}^{\text{prefix}}(n, m) = 2 + T_{\text{worst}}^{\text{prefix}}(n - 1, m - 1)
\]
Or in closed form:

\[ T_{\text{prefix}}(n, m) = 2 \times \min(n, m) + 1 \]

We use this result to determine \( T_{\text{substring}} \):

\[
\begin{align*}
T_{\text{substring}}(n, 0) &= T_{\text{substring}}(0, m) = 1 \\
T_{\text{substring}}(n, m) &= 1 + T_{\text{substring}}(n, m) + T_{\text{substring}}(n, m - 1) \\
&= 1 + 2 \times \min(n, m) + 1 + T_{\text{substring}}(n, m - 1) \\
&= 2 + 2 \times \min(n, m) + T_{\text{substring}}(n, m - 1)
\end{align*}
\]

Which leads us to the closed form:

\[
T_{\text{substring}}(n, m) = 1 + \sum_{i=1}^{m} (2 + 2 \times \min(n, i))
\]

In general, the string we are searching for will be much shorter than the text in which we are searching, so \( \min(n, i) = n \) for most \( i \). We approximate \( T_{\text{substring}} \) therefore:

\[
T_{\text{substring}} \leq 1 + \sum_{i=1}^{m} (2 + 2n)
= 1 + 2 \times m + 2 \sum_{i=1}^{m} n
= 1 + 2 \times m + n \times m
\]

This means that the running time of substring is proportional to the product of the length of the two argument strings.

11.2 The \( \mathcal{O} \)-Notation

The results we obtained by estimating \( T(n) \) did not tell us anything about the running time in seconds, as we do not know how long a single reduction takes on a given computer, but it provides us with an abstract notion of complexity in terms of reduction steps. We call this the work complexity of a function (or program). So far, our complexity estimates were rather accurate, but it turns out that it often does not really matter if a function requires \( n^2, 5 \times n^2, \) or \( 5 \times n^2 + 7 \times n + 1 \) reduction steps—the dominating component in each of these functions is \( n^2 \). Similarly, is is not important if isort, in the best case, needs \( n, 4 \times n, \) or \( 4 \times n + 10 \) steps. Since in this case, \( n \) dominates the function, we know that the running time grows linear with the length of the input list. Similarly, in the case of tail, we only care that the work complexity is constant; the exact number of steps is secondary. To capture the dominating component of a complexity function, we introduce the so-called \( \mathcal{O} \)-notation (pronounced “big oh”).

For example, instead of saying isort requires on average \( 0.5 \times n^2 + 0.5 \times n + 1 \) steps for a list of length \( n \), we say it takes \( \mathcal{O}(n^2) \), or “big oh of \( n^2 \)”. In the best case, sorting required \( T_{\text{isort}}(n) = 4 \times n + 1 \) steps, so we say that \( T_{\text{isort}} \) is \( \mathcal{O}(n) \). Informally, we can say that to determine the \( \mathcal{O} \)-class of a function, we are allowed to drop everything but the greatest variable factor.

11.2.1 Definition of \( \mathcal{O} \)

Let \( T(n) \) be a function that describes the running time of a program measured in terms of an input size \( n \). So \( n \) is a non-negative integer and the value of \( T(n) \) is non-negative for all \( n \). Let \( f(n) \) be some function defined on non-negative integers, then
$T(n)$ is $O(f(n))$

if for all $n$ bigger than an integer $n_0$, $T(n)$ is at most constant times $f(n)$. More formally, we say that $T(n)$ is $O(f(n))$ if there exists an $n_0$ and a constant $c > 0$ such that, for all integers $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$. Therefore, the expression $O(f)$ describes a whole class of functions over integers. The values $n_0$ and $c$ are called witnesses to the fact that $T(n)$ is in $O(f(n))$.

### 11.2.2 An Example

Find the witnesses to prove that $T(n) = 20(n + 2)^2$ is $O(n^2)$. According to the above definition, we have to find $c$ and $n_0$ such that for all $n \geq n_0$

$$20(n + 2)^2 = 20n^2 + 80n + 80 \leq c \cdot n^2$$

Obviously, for such an $n_0$ to exist, $c$ has to be bigger than 20. Let’s choose $c = 21$:

$$20n^2 + 80n + 80 \leq 21 \cdot n^2$$

which is equivalent to

$$80n + 80 \leq n^2$$

This holds for all $n \geq 81$. We found witnesses $c = 21$ and $n_0 = 81$ proving that $T(n) = 20(n + 2)^2$ is $O(n^2)$.

### 11.2.3 Some Observations

The $O$-notation tells us only something about how steeply a function grows at most. For example, every function that is in $O(1)$ is also in $O(n)$, and every function in $O(n)$ is in $O(n^2)$, but of course, not vice versa! In particular, we can make the following observations, where $c$ is a constant value and $f$ and $g$ functions:

1. $O(c)$ is equal to $O(1)$: every function that is in $O(c)$ is also in $O(1)$, and vice versa. This class of functions is usually referred to as $O(1)$. For a program to be in this class means it only needs a constant number of steps to evaluate, independent of its input size. The list operations head and tail are examples of such functions.

2. $O(c \cdot f)$ is equal to $O(f)$: this is actually just a more general version of the previous observation. For example $O(10 \cdot n^2)$ describes the same class of functions as $O(n^2)$.

3. $O(f + g)$ is equal to $O(f)$ if all functions that are in $O(g)$ are in $O(f)$. For example, since every function that is in $O(n)$ is also in $O(n^2)$, we can conclude that $O(n^2 + n)$ is equal to $O(n^2)$.

To classify a function $T(n)$, we are generally interested in the smallest $O$-class the function is in. That is, if $T(n)$ is $O(n)$, we want to say so, rather than making the technically sound, but weaker, statement that it is in $O(3 \cdot n + 5)$. But on the other hand, it does not make any sense to say $T(n)$ is in $0.0001 \cdot O(0.0001 \cdot n)$, although it is smaller, since constant factors do not matter.

The following table provides a list of some common $O$-classes:

<table>
<thead>
<tr>
<th>$O$</th>
<th>Informal name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>Cubic</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>Exponential</td>
</tr>
</tbody>
</table>
11.3 More Efficient Sorting Algorithms

Our first attempt at implementing a function sorting list elements, namely isort, was not very successful, as it turned out to perform poorly. For our second attempt, we will use a powerful algorithmic technique called divide-and-conquer. The technique is based on the idea that sometimes when solving a problem is not straightforward, it may help to split the input (typically but not necessarily in two) parts of similar size; then, solve the problem on each of the smaller parts; and finally, try to combine the sub-results into the final result. The following program scheme illustrates this:

\[
\text{divide-and-conquer } p \\
\quad | \text{ (solution for } p \text{ is trivial) } \rightarrow \text{ solve } p \\
\quad | \text{ otherwise } \\
\quad \quad \text{let} \\
\quad \quad \quad (p1, p2) = \text{ (split input into two parts)} \\
\quad \quad \quad \text{sol1} = \text{ divide-and-conquer } p1 \\
\quad \quad \quad \text{sol2} = \text{ divide-and-conquer } p2 \\
\quad \quad \text{in } \text{ (combine the solutions sol1 sol2)}
\]

11.3.1 Mergesort

How can we apply this idea to sorting? The trivial case is easily identified: a list that has only a single element or none at all is always sorted, so we have the first part of the new sorting program:

\[
\text{sort } xs \\
\quad | \text{ length } xs \leq 1 = xs \\
\quad | \text{ otherwise } = \text{ (rest of the function)}
\]

The next question is how to split the problem into two parts? The straightforward solution is just to split the list into two sublists using the predefined function `splitAt :: Int -> [b] -> ([b], [b])` which splits the list at a given index. Since the subproblems should have roughly the same size, we split the list in the middle, and call `sort` for each sublist, which brings us to

\[
\text{sort } xs \\
\quad | \text{ len } \leq 1 = xs \\
\quad | \text{ otherwise } = \text{ (combine solutions)} \\
\quad \quad \text{where} \\
\quad \quad \quad \text{len} = \text{ length } xs \\
\quad \quad \quad (xs1, xs2) = \text{ splitAt (len ‘div‘ 2) xs} \\
\quad \quad \quad xs1Sorted = \text{ sort } xs1 \\
\quad \quad \quad xs2Sorted = \text{ sort } xs2
\]

Now, we have two sorted lists and all that is left to do is to combine those two lists into a single list which is still sorted, or to `merge` the two lists. For example, given the sorted lists \([1, 4, 7, 8, 9]\) and \([2, 3, 5, 6, 10]\), the merger of these lists is \([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]\).

Thus, our next task is to define a function `merge :: Ord a => [a] -> [a] -> [a]` which given two sorted lists combines these two lists to a single sorted list. In case one of the two lists is empty (the bases cases of `merge`), the definition is straightforward, `merge` simply returns the other list. Since `xs` and `ys` are sorted, the result will obviously be sorted as well.

\[
\text{merge} :: \text{Ord a } => [a] -> [a] -> [a] \\
\text{merge} [] ys = ys \\
\text{merge} xs [] = xs \\
\langle \text{recursive step} \rangle
\]
For the recursive step, i.e., the case when both lists have at least one element, we have to check which of the two elements is smaller and put it at the head of the resulting list, and call `merge` recursively on the remainders of the lists:

\[
merge :: \text{Ord } a \Rightarrow \text{[a]} \to \text{[a]} \to \text{[a]}
\]

\[
merge [] ys = ys
\]

\[
merge xs [] = xs
\]

\[
merge (x:xs) (y:ys) =
\]

\[
| x \leq y = (x : merge xs (y:ys))
\]

\[
| \text{otherwise} = (y : merge (x:xs) ys)
\]

The sorting algorithm that we just developed is, due to the combination step, called `mergesort`. We define

\[
\text{mergesort } xs
\]

\[
| \text{len} \leq 1 = xs
\]

\[
| \text{otherwise} = \text{merge } x1\text{Sorted } x2\text{Sorted}
\]

where

\[
\text{len} = \text{length } xs
\]

\[
(x1, x2) = \text{splitAt } (\text{len} \div 2) xs
\]

\[
x1\text{Sorted} = \text{sort } x1
\]

\[
x2\text{Sorted} = \text{sort } x2
\]

Figure 11.2 shows the results of (a) splitting and (b) merging the lists in the recursive calls. Since the original list is of length 8, they can be split into sublists of exactly the same length. We can see in (a) how the first list is split into two, each of these is split again until the lists contain only one element each. Then, the lists are merged until finally we obtain the original list sorted.

### 11.3.2 Analysing Mergesort

The initial motivation for `mergesort` was to obtain a more efficient program for sorting lists, but it is not clear whether we have achieved our goal. Using the techniques from Section 11.1, we will now analyse the running time of `mergesort`.

**Analysis of `split` and `merge`**

We begin by analysing the two functions `split` and `merge`. The definition `splitAt` is rather similar to `take :: Int \to [a] \to [a]`:
splitAt:: Int -> [b] -> ([b], [b])
splitAt 0 xs = ([], xs)
splitAt n [] = ([], [])
splitAt n (x:xs) = let
    (xs', ys') = splitAt (n-1) xs
  in ((x:xs'), ys')

We can show that the timing function for splitAt n xs is \( T_{\text{splitAt}}(n) \) is \( O(n) \) (we omit the length of the list as a parameter).

Deriving the timing function \( T_{\text{merge}}(n, m) \) (where \( n \) is the length of the first input list, \( m \) the list of the second) for merge is more complicated. Looking at the base cases of merge, we get the following two equations:

\[
T_{\text{merge}}(0, m) = 1 \\
T_{\text{merge}}(n, 0) = 1
\]

If both lists are non empty, we have, depending on the values, either

\[
T_{\text{merge}}(n, m) = 2 + T(n - 1, m) \quad \text{or} \\
T_{\text{merge}}(n, m) = 2 + T(n, m - 1)
\]

In the best case, the elements are only taken from one list, say the first, and merge would require \( n \) iterations. However, in the worst case the elements are taken alternately from the two lists, thus requiring \( n + m - 1 \) iterations. Therefore, we have

\[
T_{\text{merge}}(n, m) = 2 \times (n + m - 1) + 1
\]

We can show that \( T_{\text{merge}}(n, m) \) is in \( O(n + m) \).

### Simplifying Recursive Equations

We could continue deriving \( T_{\text{mergesort}} \) using \( T_{\text{merge}} \). However, we are only interested in the \( O \)-class of \( \text{mergesort} \), not the exact number of steps. Fortunately, it is sufficient in this case to approximate \( T_{\text{merge}} \) with any member of its \( O \)-class.

More formally, if the timing function \( f \) is defined by the equation

\[
f(n) = g_1(n) + g_2(n) + \ldots
\]

for some functions \( g_i \) and \( g_i \) is in \( O(g'_i) \), then \( f \) is in the same \( O \)-class as

\[
f'(n) = g'_1(n) + g'_2(n) + \ldots
\]

and \( O(g') \) is equal to \( O(g) \).

This basically says that, if we are only interested in the \( O \)-class of a recursively defined function, we do not lose any information if we replace every sub-function by another representative of its class. For example, if we have the recursive equation

\[
f(n) = \underbrace{3 \times n + 1 + f(n - 1)}_{O(n)}
\]

and only want to determine the \( O \)-class of \( f \), we can simplify the recursive equation to

\[
f(n) = n + f(n - 1)
\]
For the analysis of \texttt{mergesort}, this means that if we know \texttt{length} is in $O(n)$, we can simply use the $T_{\text{length}}(n) = n$ in the recursive equation for \texttt{mergesort}. The base cases are straightforward:

\[
T_{\text{msort}}(0) = 1 \\
T_{\text{msort}}(1) = 1
\]

In the recursive step (the list has $n > 1$ elements), we have to add reduction steps as follows:

1. The boolean operation $\leq$: 1 step.
2. \texttt{length} is $O(n)$: $n$ steps.
3. Split the list at position $n/2$: we can choose any representative of $O(n/2)$: $n$ steps.
4. Merging two lists each of size $n/2$: $T_{\text{merge}}(n/2, n/2) = n/2 + n/2 = n$ steps.
5. Two applications of \texttt{mergesort} to lists of length $(n/2)$: $2 * T_{\text{msort}}(n/2)$.

Adding the step count of (1) to (4) results in $3 * n + 1$, which is in $O(n)$, so according to the theorem, we obtain the following (already simplified) recursive equation:

\[
T_{\text{msort}}(n) = n + 2 * T_{\text{msort}}(n/2)
\]

We can see that \texttt{mergesort} is called as many times as $n$ is divisible by two, that is $log_2 n$ times. For example,

\[
T_{\text{msort}}(16) = 16 + 2 * (8 + 2 * (4 + 2 * (2 + 2 * 1))) \\
= 16 + 16 + 16 + 16 + 16 \\
= 16 * (1 + log_2 16)
\]

In general, $T_{\text{msort}}(n) = n * (1 + log_2 n)$, which is in $O(n * log_2 n)$. This is a significant improvement over the asymptotic work complexity $O(n^2)$ which we obtained for \texttt{isort}. In Figure 11.3 we can see that the growth of $n * log n$ is not much worse than the linear curve, whereas the quadratic curve grows very steeply.
11.3.3 Quicksort

Mergesort puts all the effort into merging sub-results and does splitting in the simplest possible way. Another divide-and-conquer sorting algorithm, called quicksort, is based on the idea that if we split the list in the right way, we can combine the sub-results by simply appending the lists. For splitting, we pick a pivot element from the list and collect all the list elements smaller than the pivot in the first list, all greater in a second list, and the elements equal to the pivot in a third list. The third list is already sorted, since it only contains elements with the same value, so only the first two sublists have to be sorted. Since all the elements in the first list are smaller than those in the second, which, in turn, are smaller than the elements in the third, we just have to append them to obtain a the result. Overall, we have

```haskell
quicksort xs
  | len <= 1 = xs
  | otherwise =
    smaller ++ equal ++ greater
where
  piv = head xs
  smaller = quicksort (filter (< piv) xs)
  equal = filter (== piv) xs
  greater = quicksort (filter (> piv) xs)
```

In the best case, quicksort splits the input list into two lists of more or less equal size, and the work complexity is the same as for mergesort. However, in the worst case, the pivot may always be either the smallest or biggest element in the list, so one of the lists is empty and the size of the other is only one less than the original list. Each iteration takes $O(n)$ steps, and this repeated $n$ times leads to an asymptotic complexity of $O(n^2)$. Nevertheless, many implementations are based on quicksort since an optimised implementation can be faster on average than mergesort. However, the implementations choose the pivot more carefully to minimise the chances for the occurrence of the worst case.

11.4 Solving Recurrence Relations

When trying to determine the timing function for a recursive function, we end up with a recursive equation that defines the timing function. These recursive equations are called a recurrence relation. Usually, we need to obtain a closed form for such a recurrence relation, which can be difficult. Fortunately, there is a limited number of kinds of recurrence relations which typically occur in the context of timing functions. The following table lists the most common relations and their $O$-class:

<table>
<thead>
<tr>
<th>Recursive Equation</th>
<th>$T(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T(n) = T(n-1) + b \cdot n^k$</td>
<td>$O(n^{k+1})$</td>
</tr>
<tr>
<td>$T(n) = c \cdot T(n-1) + b \cdot n^k, c &gt; 1$</td>
<td>$O(c^n)$</td>
</tr>
<tr>
<td>$T(n) = c \cdot T(n/d) + b \cdot n^k, c &gt; d^k$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>$T(n) = c \cdot T(n/d) + b \cdot n^k, c &lt; d^k$</td>
<td>$O(n^k)$</td>
</tr>
<tr>
<td>$T(n) = c \cdot T(n/d) + b \cdot n^k, c = d^k$</td>
<td>$O(n^k \cdot \log n)$</td>
</tr>
</tbody>
</table>

11.5 Exercises

1. Given the following definition:

```haskell
myFold :: a -> (a -> a -> a) -> [a] -> a
myFold n f [] = n
myFold n f (x:xs) = myFold (f n x) f xs
```
How many reduction steps are necessary to compute the result of \( \text{myFold} \ n \ f \ x:xs \) for list \( x:xs \) of length \( n \) if for every \( x \) and \( y \) evaluating \( f \ x \ y \) takes exactly three reduction steps?

2. For the examples from Section 11.1.1, find the smallest \( O \)-classes of each of the timing functions.
Chapter 12

Tree Structures

“Breadth-first search is the bulldozer of science.”
—Randy Goebel

Lists present us with a linear structure of values. Each element in a list is attached to a list of following elements and this continues until the end of the list. However, very often in computing, we need to store collections of values, where each value is followed by two or more other values. For example, in a file system, a directory contains a collection of files, each of which can again be a directory, and so on. Generally, trees are useful for organising data hierarchically and they can often lead to more efficient implementations than lists.

12.1 Data Structures and Efficiency

In all the previous examples of user defined data structures, the design of the structure was only guided by two priorities:

1. The data structure has to be able store all the necessary information.
2. Types should not be too general, e.g., don’t use integers to encode Boolean values, strings for days of the week, to avoid runtime checks.

So far, we did not consider how the choice of the data structure may influence the efficiency of the programs working on this structure. We will use a simplified version of a very common problem as an example, namely storing and looking up an unknown number of sorted data items.

**Problem:** Write a program that offers the following functions: inserting a new element into a data base and checking if a given item is already in the data base. The data items are of unknown type, but we can assume that an order is defined on them (i.e, operations like $<$, $>$ can be applied to the data items). Which data structure is suitable to store the data if efficiency is important for both inserting and checking?

So far, we used lists whenever we needed an ordered data structure of unknown size. We first define a function `insertSorted`. To speed up searching for an element in the list we keep the list sorted by making sure that if the list is sorted, `insertSorted` will not destroy the order. Since the empty list is sorted, the list will always be sorted if we only use `insert` to update it.

```haskell
insertSorted:: Ord a => a -> [a] -> [a]
insertSorted x [] = [x]
insertSorted x (y:ys)
  | x <= y = (x:y:ys)
  | otherwise = y : (insertSorted x ys)
```
Now, if we want to check if an element is already in the list, we do not have to search through the whole list, we stop as soon as we find an element greater than the element we are searching for. As the list is ordered, the element cannot come afterwards:

```haskell
searchList:: Ord a => a -> [a] -> Bool
searchList x [] = False
searchList x (y:ys) |
| x < y = False   
| x == y = True   
| otherwise = searchList x ys
```

How many steps does inserting and searching require on average, in the worst and in the best case? If the list has \( n \) elements,

- it will take \( O(n) \) on average and in the worst case, and
- it will take \( O(1) \) in the best case.

This is clearly not optimal. Imagine you had to search through a stack of sorted papers, you probably would not go through them from top to bottom. Instead, a so-called binary search seems more effective: halve the stack and check whether it is in the top or bottom half, and then continue in this way (this is just another application of a divide-and-conquer strategy): it requires \( O(\log_2 n) \) steps.

Unfortunately, we cannot do this on lists, as it is a linear structure: to reach, for example, the last element of a list \( e_n \), we have to traverse the whole list:

\[
\begin{array}{c}
e_0 \\
\vdots \\
e_1 \\
\vdots \\
e_n \\
\end{array} [ ]
\]

So, if we want to avoid linear complexity for the functions, we have to choose another data structure. Let us consider an example data base consisting of the elements \( e_0 \) to \( e_{14} \), where \( e_1 \leq e_{i+1} \). Instead of going from \( e_0 \) to \( e_1 \) and so on, for the binary search we want to start with \( e_7 \), and then go to \( e_3 \) or \( e_{11} \), and so on. The following diagram shows the possible paths:

Looking at the diagram, it becomes clear that we need a structure which, from every node value, offers two possible alternatives: (1) all the elements smaller than the current values and (2) all those greater than the current value. Such a structure is called a tree (with some imagination, the diagram looks like an upside-down tree). Any such a tree storing integer values consists of the following three pieces:

1. An integer node value.
2. One tree containing all elements smaller than the node.
3. Another tree containing all elements greater than the node value. Alternatively, it can also be just an empty tree (i.e., a leaf). This translates into the following Haskell definition:

```
data IntTree = Node Int IntTree IntTree |
| Leaf
deriving (Show)
```

The statement `deriving (Show)` ensures that the new data type is included in the type class `Show`. Therefore, values of type `IntTree` can be converted into strings and printed.

To see the advantage of `IntTree` in contrast to lists in terms of accessing the single elements, consider the sorted sequence 1 to 7. It can be stored in the tree as follows:

```
Node 4 (Node 2 (Node 1 Leaf Leaf) |
| (Node 3 Leaf Leaf))
| (Node 6 (Node 5 Leaf Leaf) |
| (Node 7 Leaf Leaf))
```

Accessing the middle element of the sequence takes a single step, since it is in the top most node (called root node). Accessing 2 and 6 each takes two steps, and all other three. So, in at most three \( O(\log_2 n) \) steps any value can be reached.

Compared to lists, the definition of trees has a big disadvantage: it can only store elements on type `Int`, whereas lists can store elements of arbitrary type.

As in the case of lists, we can define generic trees by using a type variable (\( a \) in the following definition) instead of a concrete type:

```
data Tree a = Leaf |
| Node (Tree a) (Tree a)
deriving (Show)
```

We could then define a tree of integer values in terms of the polymorphic tree structure

```
type IntTree = Tree Int
```

Note how both in the definition of `Tree` and `IntTree` the element type is used just like arguments in case of functions.

Before we discuss the properties of this structure any further, let us have a look at how to define operations on trees. We start with `insertTree`, the corresponding function to `insertSorted`, which takes an ordered tree and an item as argument and inserts the item in the tree such that the resulting tree is still ordered.

How does inserting on this data structure work?

- If the tree is empty, i.e., a leaf, return a node with the new element and empty subtrees (i.e., leaves).
- If the tree has at least one node, compare the new element to the element in the tree:
  - Call insert recursively on the left subtree if it is smaller or equal, or
  - call insert recursively on the left subtree if it is greater.

Then, build the new node using the updated left or right subtree.

So, we can define insert as follows:

```
insertTree :: Ord a => a -> Tree a -> Tree a
insertTree x Leaf = Node x Leaf Leaf
| x <= y = Node y (insertTree x t1) t2
| otherwise = Node y t1 (insertTree x t2)
```
Searching the tree for a given value works much in the same way as inserting a new element. Depending on whether the node value of the tree is less or greater than the element we are looking for, we continue by searching the left or right subtree:

```haskell
searchTree:: Ord a => a -> Tree a -> Bool
searchTree x Leaf = False
searchTree x (Node y t1 t2)
  | x == y = True
  | x < y = searchTree x t1
  | otherwise = searchTree x t2
```

Let us have a look at the results of `insertTree` applied to three values 1, 2, and 3, depending on the order in which the values are inserted into an initially empty tree (i.e. a leaf)

```haskell
Tree> insertTree 1 (insertTree 2 (insertTree 3 Leaf))
Node 3 (Node 2 (Node 1 Leaf Leaf) Leaf) Leaf

Tree> insertTree 1 (insertTree 3 (insertTree 2 Leaf))
Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf)

Tree> insertTree 3 (insertTree 2 (insertTree 1 Leaf))
Node 1 Leaf (Node 2 Leaf (Node 3 Leaf Leaf))
```

In all the cases, the resulting tree is a sorted tree containing the same numbers. However, we can see that the shape depends on the order in which the elements are inserted into the structure. In the examples in Figure 12.1 this is even more obvious. Both trees store the elements 1 to 7, but their shape is completely different. The tree on the left hand side is called a **perfectly balanced tree**, as for each node the left subtree has exactly the same shape as the subtree on the right. The tree on the right hand side looks more like a list and is therefore called a **degenerated tree**. Searching this tree requires $O(n)$ steps, not $O(\log_2 n)$ as we hoped (and which is what it would take to search a perfectly balanced tree). So, just by choosing binary trees as data structure, we do not have the guarantee that searching on inserting can be done efficiently. We have to make some assertions about the shape of the tree.

### 12.2 AVL Trees

The number of recursive calls necessary for `searchTree` and `insertTree` in the worst case depends on the depth of the tree, that is the number of levels the tree consists of. More formally, we define the depth of a tree as follows:
1. The depth of a leaf is 0.

2. The depth of a node is 1 plus the maximum of the depth of its left and its right subtree.

A perfectly balanced tree of depth $n$ holds exactly $2^n - 1$ elements, and a degenerated tree $n$ elements. Therefore, no perfectly balanced tree exists with, for example, 20 elements.

The function `insertTree` guarantees that, provided the input tree is sorted, the resulting tree is sorted as well. But as we have seen, for efficiency reasons, we need a stronger property: we have to make sure that we do not end up with a degenerated tree. Ideally, we would like to have a perfectly balanced tree, but this is obviously not possible for every number of elements, since such a tree can only have 1, 3, 7, 15, etc elements. AVL trees, named after their inventors, Adelson-Velskii and Landis, solve this problem. They are “almost balanced” and have $O(2^n)$ elements, so searching and inserting is still $O(\log_2 n)$.

**Definition.** A tree is an **AVL tree** if the following two conditions are met:

1. The depth of the left and the right subtree of an AVL tree differ by at most one.

2. Both subtrees are AVL trees.

For example, `Leaf` trivially is an AVL tree, since all it’s subtrees are AVL. Also, every tree of depth one (there is only exactly one: `Node x Leaf Leaf`) and two are AVL. By definition, the depth of their subtrees cannot differ by more than one. Figure 12.2 displays various trees of depth three (the node value is a pair of depth of left subtree, depth of right subtree). The trees (a) - (c) are all AVL trees—the difference between the depth of the left and right subtree are at most one. In case of (a), the left and the right subtree of each node have exactly the same depth, so it is a perfectly balanced tree. Looking at the root node of tree (d), however, we can see that the left subtree has depth zero, the right depth two, so the tree is not an AVL tree (in fact, it is a degenerated tree).

So the question is, can we improve the `insertTree` function so that it makes sure that, given the initial tree is AVL, the resulting tree has the AVL property as well? As it turns out, it is useful to store in each node how balanced or unbalanced its subtrees are. For AVL trees, we use an additional field `Balance` in each node which is zero if both subtrees have the same depth, has a
positive value \( n \) if the depth of the left subtree is \( n \) plus the depth of the right, and is \(-n\) if it is the other way round.\(^1\)

```haskell
type Balance = Int  -- 0: both subtrees have the same depth
               -- n: the left subtree is n deeper than the right
               -- -n: the right subtree is n deeper than the left

-- The tree has AVL property if Balance is in each node either
-- -1, 0 or 1
data AVLTree a = Node Balance a (AVLTree a) (AVLTree a)
  | Leaf
  deriving (Show)
```

How can we insert elements in an AVL tree without destroying its central property? As (almost) always, the base case, that is inserting into an empty AVL tree, is straightforward:

```haskell```

\[
\text{insertAVL} :: \text{Ord a} \Rightarrow a \rightarrow \text{AVLTree a} \rightarrow \text{AVLTree a}
\]

\[
\text{insertAVL} a \text{ Leaf} = \text{Node } 0 a \text{ Leaf Leaf}
\]

It is easy to see that the resulting tree is balanced. If we insert into a non-empty tree whose subtrees have the same depth (i.e., balance is zero), then inserting a new value into either its left or right subtree cannot destroy the AVL property, as it may increment the depth of the subtree by at most one, resulting in balance 1 or \(-1\), both of which is acceptable. However, the critical case is if we insert a value into the left subtree which already is one level deeper than the right (balance = 1), or into the left if the balance is equal to \(-1\). But even this case is only problematic if the insertion operation increased the depth of the subtree. Then, the resulting tree would have the balance \(+2\) or \(-2\). To summarise, the following cases are possible:

1. **balance = 0**: insert value in proper subtree. If operation increments depth of subtree, set balance to 1 (if inserted in left subtree), \(-1\) otherwise. Operation does increment the depth of the tree.

2. **balance = 1**: if value has to be inserted, proceed as follows:
   - In left subtree **and** insertion increments depth of subtree, balance becomes 2, resulting tree is not AVL!
   - In left subtree and insertion does not increments depth of subtree, resulting tree is still AVL, balance stays 1.
   - In right subtree, operation increments depth, set balance to 0.
   - In right subtree, operation does not increment depth, balance remains 1.

3. **balance = \(-1\)**: symmetric to previous case.

In the cases where balance becomes \(+2\) or \(-2\), we are not finished with the insertion process. Let us just consider \(+2\), as the other case is exactly symmetrical and can be solved in the same way.

After inserting a value into a subtree, we need to know whether the insertion altered the depth of the tree. We therefore need internally a function:

```haskell```

\[
\text{insertAVL'} :: \text{Ord a} \Rightarrow a \rightarrow \text{AVLTree a} \rightarrow (\text{AVLTree a, Bool})
\]

\(^1\)What would the values of \text{Balance} be for trees (a) to (d) in each node in Figure 12.2?
which, in addition to the new tree, returns a boolean value (True in case the depth of the new
tree is bigger than the depth of the original tree, False otherwise):

\[
\text{insertAVLTree} :: \text{Ord} \ a \Rightarrow a \rightarrow \text{AVLTree} \ a \rightarrow \text{AVLTree} \ a
\]

\[
\text{insertAVLTree} \ a \ t = \text{fst} \ (\text{insertAVLTree'} \ a \ t)
\]

\[
\text{insertAVLTree'} :: \text{Ord} \ a \Rightarrow a \rightarrow \text{AVLTree} \ a \rightarrow (\text{AVLTree} \ a, \text{Bool})
\]

\[
\text{insertAVLTree'} \ x \ \text{Leaf} = (\text{Node} \ 0 \ x \ \text{Leaf} \ \text{Leaf}, \text{True})
\]

Consider the following situation: we just inserted an element into the left subtree of a tree of
depth \(n + 1\) with balance of \(+1\) (i.e., the original left subtree must have depth \(n\), the right subtree
of \(n - 1\)). The insert operation then incremented the depth of the left subtree, resulting in a tree
of the following form:

\[
\text{Node} \ 2 \ X
\]

\[
\text{n+1} \quad \text{n-1}
\]

\[
\text{tree 1} \quad \text{tree 2}
\]

where the triangles represent subtrees, the number in the triangles denote their depth \((n \geq 1)\), the
values in Node the balance and the element itself.

To balance the tree, we have to move at least one element from tree 1 to tree 2. How can
this be done? Let us look more closer at tree 1—it is at least of depth 2, so its topmost value is a
Node:

\[
\text{Node} \ 2 \ X
\]

\[
\text{Node} \ ? \ Y
\]

\[
\text{Node} \ ? \ \text{Y}
\]

\[
\text{Node} \ ? \ \text{tree A}
\]

\[
\text{Node} \ ? \ \text{tree B}
\]

\[
\text{Node} \ ? \ \text{n-1}
\]

\[
\text{Node} \ ? \ \text{tree 2}
\]

What we have to do depends on the balance value in the topmost node of tree 1. As we can see
in Figure 12.3, if the balance is 0 or 1, it is sufficient to rotate a single node value to the right.
Otherwise, we have to perform a double rotate to the right.

Although insertion into an AVL tree requires more steps than the simple tree insertion, it is
still \(O(\log_2 n)\), as it has to perform at most one rotation on each level, and each rotation requires
only a constant number of steps.

### 12.3 Tries

As an illustration of the usefulness of trees for the efficient storage and processing of large
amounts of data, consider storing an English language dictionary, e.g., to implement a spell
checker. The problem here is that a dictionary requires a large amount of memory to store and
that it therefore also takes a long time to search for a word if it is done naively.

To understand how dictionaries can be stored more efficiently, consider the following words:
Single rotate right:

Double rotate right:

Figure 12.3: Right rotation of AVL trees
12.3 Tries

We could, of course, use AVL trees to store the words of a dictionary, which makes searching and inserting to be of $O(\log n)$ for $n$ entries in the dictionary (see Figure 12.4). The disadvantage of this solution is that each word has to be stored completely. In the example, there are 18 letters, but if we take away the overlaps, we actually only have to store 9. The overlap is typical for dictionaries. We achieve this using a tree-based data structure called a trie.\(^2\) A trie is a tree where each path corresponds to one word of the dictionary. Similar words share portions of paths, which leads to a substantial saving of memory. Each node contains a Boolean value that indicates whether the path to the current node already constitutes a complete word (a sub-word like ear of a longer word like earl might be a complete word). Moreover, it holds an association list of characters to the tries that are the children of the current node:

```
data Trie = TrieNode Bool [(Char, Trie)]
deriving (Show)
```

We could add new words into a dictionary? We have to check for each letter in the word if there is already a dictionary associated with the current letter in the list of the current node. If this is the case, we insert the rest of the word into this dictionary. Otherwise, we create a new empty dictionary, insert the rest of the word into this dictionary, and add the pair consisting of the first letter and the dictionary into the list. This continues until we are left with the empty string. The empty string can be inserted by simply setting the Boolean value in the current node to True, which marks the end of a path. The auxiliary function addWordList searches the association list of characters and tries for the entry that matches the first character of the string. If the list is empty, there is no such entry:

```
addWordList:: String -> [(Char, Trie)] -> [(Char, Trie)]
addWordList (fstChar:restStr) [] = -- no dictionaries left, create
  [(fstChar, newTrie)] -- new dictionary assoc. with
  where
  newTrie = Node True []

addWordList (fstChar:restStr) ([(char, trie)]) = -- found a dictionary
  Node isEnd [addWordList restStr (trie)]
```

\(^2\)The word is derived from retrieve
emptyTrie = Node False 
newTrie  = addWord restStr emptyTrie

Note that addWordList is only called when the string has at least one character. This function definition is quite interesting, although addWord is a recursive function, it does not contain a call to itself. Instead, it calls addWordList, which in turn calls addWord. The two functions are therefore said to be mutally recursive.

The next function, makeTrie, takes a dictionary, naively represented as a list of strings, and converts it into a trie. Starting from an empty trie—the base case—the function recursively adds one word after another into the trie using the function addWord:

makeTrie :: [String] -> Trie
makeTrie [] = TrieNode False []
makeTrie (word:words) = addWord word (makeTrie words)

Finally, we have checkWord, which checks whether a given word is in a trie:

checkWord:: String -> Trie -> Bool
checkWord [] (TrieNode isEnd _) = isEnd
checkWord (c:cs) (TrieNode _ tries) = check c cs tries
where
  check c cs [] = False
  check c cs ((c’, trie) : rest) =
    | c == c’ = checkWord cs trie
    | otherwise = check c cs rest

12.4 Exercises

1. Binary ordered trees:

   (a) Define a function minInTree:: Tree Int -> Int which returns the minimum of all values stored in an ordered binary tree.

   (b) Define a function flattenTree:: Tree a -> [a] which dumps all the elements stored in the nodes of the tree into a list. If the tree is ordered, the resulting list should be sorted as well.

2. AVL trees: Write a function checkAVL:: Ord a => AVLTree a -> Bool which checks whether a tree is an AVL tree.

   It should return True if and only if the balance stored in each node is in the legal range, it is actually the correct value (that is, the difference between the depth of the left and the right subtree), and the elements are ordered. Examples:

   (a) Node 2 5
       /  
      /   
     Leaf Leaf

   (b) Node 2 5
       /  
      /   
     Leaf Node 1 3 Leaf
     /  
    /   
   Leaf Leaf

   (c) Node 0 5
       /  
      /   
     Node 0 2 Leaf
     /  
    /   
   Leaf Leaf

   (a) is an AVL tree, but the balance value is incorrect, (b) is not AVL since the left subtree has depth 2, the right depth 0, and (c) is a correct AVL tree.

3. Tries:
(a) Draw a schematic representation of such a dictionary trie which contains the words "arm", "army", and "art".
(b) How does the trie change when you add the word "a"?
(c) How does the trie change when you add the word "argue"?
(d) Sketch the evaluation steps for checkWord "arrow" dic and checkWord "art" dic.
(e) Write a function countWords::Dictionary -> Int which returns the number of words stored in a dictionary.
Chapter 13

Evaluating Arithmetic Expressions

Programming languages—except the machine language directly understood by the computer hardware—are executed in one of two possible ways: they are compiled or interpreted. Compilation means that the original program—which we call the source program—is read by a special program called a compiler. The compiler translates the source into machine code. Once compiled, the hardware can directly execute the program. During compilation, a compiler usually spends a fair amount of time on optimising the program, so that it runs faster and/or consumes less space during execution.

The alternative is to use an interpreter. An interpreter does not generate machine code that can be directly executed by the computer hardware; instead, it executes the instructions itself, right from the source language. It functions as an intermediary between the source and the computer hardware as the latter cannot directly understand the source. The program implementing the interpreter itself, however, is available in machine code (usually as the result of compilation). Examples of interpreters are **ghci**, which interprets Haskell programs, and **bash**, which interprets shell scripts.

13.1 Evaluating Closed Expressions

Below, we will study a simple variant of an interpreter, namely a program that can evaluate arithmetic expressions, such as $3 + 4$, $2 + 7 \times 3$, and $4 \times (3 + 12/2)$.

13.1.1 Representing Expressions

Given the task of evaluating arithmetic expressions, the first question to ask is, how can we represent these arithmetic expressions? A plain string of the form "$4 \times (3 + 12/2)$" does not expose the internal structure of an arithmetic expression appropriately. The simplest representations for expressions are tree structures. The table below illustrates the correspondence between expressions and trees.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3 + 4$</td>
<td>$+ \left(3 \right. \left.4\right)$</td>
</tr>
<tr>
<td>$2 + 7 \times 3$</td>
<td>$+ \left(2 \left.\times \left.3\right)\right)$</td>
</tr>
<tr>
<td>$4 \times (12/2 + 3)$</td>
<td>$\times \left(4 \left.\left(2 \left.\left.\div \left.12\right)\right)\right)\right) + \left.\left.3\right)\right)$</td>
</tr>
</tbody>
</table>

We call trees that represent expressions **expression trees**. Obvious benefits of expression trees over the string representation of an expression are that the formatting of the expression (e.g., use of space) does not matter and that the order of evaluating subexpressions is clear.
We have earlier seen that we can model trees using user-defined data structures. What does a data type representing expression trees look like? The simplest kinds of expressions are constants, such as 42. They form the leaves of an expression tree. Nodes of an expression tree are augmented with the name of a binary operator and have two subexpressions—i.e., we have a binary tree:

```
data Expr = Const Int -- integer constant
  | Op BinOp Expr Expr -- application of a binary operation
deriving (Show)
```

```
data BinOp = Add -- addition
  | Sub -- subtraction
  | Mul -- multiplication
  | Div -- division
deriving (Show)
```

In other words, an expression tree for our simple expressions is a binary tree of integer values augmented with binary operators.

Generally, the first step performed by any compiler or interpreter is to convert the textual representation of a program into a tree, not unlike our expression trees. Afterwards all further manipulation of the program is implemented on that program tree.

### 13.1.2 Pretty Printing Expressions

By deriving `Show`, we can output expression trees so that, for example, the tree corresponding to $4 \times (3 + 12/2)$ is displayed as

```
Op Mul (Const 4) (Op Add (Const 3) (Op Div (Const 12) (Const 2)))
```

Since this is not particularly readable, we first want to implement a pretty printer for arithmetic expressions, i.e., a function that outputs an expression tree in a form more familiar to us, say $4 \times (3 + 12/2)$. We implement the pretty printer through a recursive traversal of the expression tree:

```
pretty :: Expr -> String
pretty (Const i) = show i
pretty (Op op e1 e2) =
  "(" ++ pretty e1 ++ " " ++ opToString op ++ " " ++ pretty e2 ++ ")"
where
  opToString Add = "+
  opToString Sub = "-
  opToString Mul = "*
  opToString Div = "/"
```

We must be careful to add appropriate parentheses to guarantee that the desired order of evaluation is not altered.

### 13.1.3 Parsing Expressions

More important than pretty printing expressions—i.e., converting expression trees to strings—is the inverse function, namely producing an expression tree from the string representation of an expression. This process is generally known as parsing and is, unfortunately, considerably more complex than pretty printing. Hence, we will not discuss the details here. For our purposes, parsing is a function of type

```
parse :: String -> Expr
```

and the implementation of a parser for simple expressions is provided in Appendix B.1.
13.1.4 Expression Evaluation

Finally, we would like to compute the value of a given expression, i.e., we want to implement a function of type $\texttt{eval} :: \texttt{Expr} \to \texttt{Int}$. Similar to pretty printing, we can implement it by a tree traversal. Instead of converting the nodes and leaves of the tree into strings, we determine their values. A possible implementation might be:

```haskell
eval :: Expr -> Int

eval (Const i) = i
eval (Op Add e1 e2) = eval e1 + eval e2
eval (Op Sub e1 e2) = eval e1 - eval e2
eval (Op Mul e1 e2) = eval e1 * eval e2
eval (Op Div e1 e2) = eval e1 `div` eval e2
```

This implementation certainly is correct, but we might wonder whether there is not an alternative implementation that uses only one recursive case—like the code for `pretty`. In fact, such a variant is possible, using a higher-order function `opToFun` that converts a binary operator into the actual Haskell function implementing the operator. The overall result is:

```haskell
eval :: Expr -> Int

eval (Const i) = i
eval (Op op e1 e2) = (opToFun op) (eval e1) (eval e2)

opToFun :: BinOp -> (Int -> Int -> Int)

opToFun Add = (+)
opToFun Sub = (-)
opToFun Mul = (*)
opToFun Div = div
```

The application `(opToFun op)` retrieves the appropriate binary operator and applies it to the subresults produced by `eval e1` and `eval e2`.

13.2 Expressions Containing Variables

In programs, expressions usually contain variables—for example, $x \times (3 + 12/y)$. Generally, we call expressions with variables open and those without variables closed expressions. To handle open expressions, we must first extend our definition of expression trees. We introduce `Ident` as a synonym of `String`, which is used to represent variables by their name. Then, we add a second form of leaves to the definition of expression trees. This second form of leaves stores a variable name instead of an integer constant:

```haskell
type Ident = String

data Expr = Const Int -- integer constant
           | Var Ident -- variable
           | Op BinOp Expr Expr -- application of a binary operation
```

13.2.1 Extended Versions of Pretty Printing and Evaluation

In a second step, we extend the pretty printing function `pretty` such that it covers the new variant of the expression type `Expr`—variables are given by their name. Here is the overall function:

```haskell
pretty :: Expr -> String

pretty (Const i) = show i
pretty (Var ide) = ide
```
pretty (Op op e1 e2) =
  "(" ++ pretty e1 ++ " " ++ opToString op ++ " " ++ pretty e2 ++ "")"
where
  opToString Add = "+
  opToString Sub = "-
  opToString Mul = "*
  opToString Div = "/

Closed expressions can be evaluated without any additional information, whereas open expressions cannot be fully evaluated unless we know the values of the variables contained in the expression. A mapping of variables to associated values for an expression is usually called an environment for the expression. We represent environments as association lists:

    type Env = [(Ident, Int)]

Finally, we need to pass an environment env to eval when we want to determine the value of an expression—we say that this is the value of the expression under the environment env. Moreover, we need an additional equation in the definition of eval that covers the case of a variable. In that case, we have to look up the value associated with that variable in the environment. We do so using the Prelude function

  lookup :: Eq a => a -> [(a, b)] -> Maybe b

If it can find a pair (x, y) in the list when searching for x, it yields Just y; otherwise, it returns Nothing. Consequently, we want to inspect the result returned by lookup with pattern matching. We could do so by using an auxiliary function, but Haskell also allows us to do pattern matching in the middle of an expression by using a case construct. The general form of the case construct is as follows:

  case e of
    p₁       -> e₁
    ...      
    pₙ       -> eₙ

The value of e is matched against the pattern p₁ to pₙ. Then, the expression corresponding to the first matching pattern is evaluated to produce the overall result.

We can use a case construct in the equation for variables to check whether lookup found the variable in the environment or not:

  eval :: Env -> Expr -> Int
  eval env (Const i   ) = i
  eval env (Var ide   ) = case lookup ide env of
                              Nothing -> unknownError ide
                              Just i  -> i
  eval env (Op op e₁ e₂) = opToFun op (eval env e₁) (eval env e₂)

  unknownError :: String -> a
  unknownError ide = error ("Unknown variable " ++ ide ++ ":")

If the variable is not in the environment, an error is raised.

### 13.2.2 Partial Evaluation

Although we can in general not compute a value for an open expression, we can sometimes still simplify such an expression. Consider, for example, \(x \ast (3-2) + y\). It clearly is the same as
the simpler $x + y$. The process of simplifying an open expression as much as possible is called
partial evaluation—the process of reducing closed subexpressions is called constant folding.

We want to implement partial evaluation as a function $\text{peval} : \text{Expr} \rightarrow \text{Expr}$. This
function leaves the leaves of the expression tree (consisting of integer constants and variables)
unchanged, but applies a set of simplification rules to tree nodes. However, before simplification,
it first partially evaluates the subexpressions of the operator, as this may enable subsequent sim-
plifications; for example, in the case of $x \times (y / y)$, we must first simplify $y / y$ to 1 before
we can simplify $x \times 1$ to $x$:

$$
\text{peval} :: \text{Expr} \rightarrow \text{Expr}
\text{peval (Const } i \text{ )} = \text{Const } i
\text{peval (Var } ide \text{ )} = \text{Var } ide
\text{peval (Op } op \text{ e1 e2) = let}
\quad \text{e1}' = \text{peval e1}
\quad \text{e2}' = \text{peval e2}
\quad \text{in}
\quad \text{simplify op e1' e2'}
$$

To code the simplification routine $\text{simplify}$, we should first consider what are the different cases
that the routine can encounter:

1. Both arguments to an operator are integer constants—e.g., $3 + 4$: Evaluate as in $\text{eval}$.
2. An algebraic law can be applied. For example, we have an operation and one of its argu-
ments is the neutral element or the zero of the operation—e.g., $x + 0$: Apply the law.
3. None of the above: Leave the node as it is.

By covering these cases and applying the most common algebraic laws, we get:

$$
\text{simplify} :: \text{BinOp} \rightarrow \text{Expr} \rightarrow \text{Expr} \rightarrow \text{Expr}
\text{simplify op (Const i1) (Const i2)} = \text{Const (opToFun op i1 i2)}
\text{simplify Add (Const 0) rhs} = \text{rhs}
\text{simplify Add lhs (Const 0)} = \text{lhs}
\text{simplify Sub (Const 0) rhs} = \text{rhs}
\text{simplify Sub lhs (Const 0)} = \text{lhs}
\text{simplify Mul (Const 0) rhs} = \text{Const 0}
\text{simplify Mul lhs (Const 0)} = \text{Const 0}
\text{simplify Mul (Const 1) rhs} = \text{rhs}
\text{simplify Mul lhs (Const 1)} = \text{lhs}
\text{simplify Div (Const 0) rhs} = \text{Const 0}
\text{simplify Div lhs (Const 0)} = \text{error "simplify: Division by zero"}
\text{simplify Div lhs (Const 1)} = \text{lhs}
\text{simplify op lhs rhs} = \text{op op lhs rhs}
$$

Partial evaluation is typical of the kind of optimisations used in optimising compilers.

### 13.3 Adding Function Definitions

The final extension that we make to the evaluator is to add user-defined functions. We would like
to be able to define functions such as

```
average x y = (x + y) / 2;
```

and then evaluate expressions such as $\text{average } 2 \ 4 + \text{average } 10 \ 20$. This raises a number
of questions:

1. How do we represent user-defined functions?
2. How do we represent applications of user-defined functions?

3. How do we evaluate applications of user-defined functions?

We consider them in this order.

### 13.3.1 Definition of User-defined Functions

Function definitions can be considered to have three parts, as follows:

\[
\begin{align*}
\text{average} & \quad \text{identifier} \\
(\text{\texttt{x y}}) & \quad \text{arguments} \\
& = (x + y) / 2 ;
\end{align*}
\]

Thus, we can represent them as a triple, but doing so would prevent us from using the Prelude function

\[\text{lookup :: Eq a \to a \to [(a,b) \to Maybe b]}\]

to look up functions in a function environment later. Thus, it is more convenient to use nested pairs:

\[
\text{type FunDef = (Ident, ([Ident], Expr)) -- function definition}
\]
\[
\text{type FunEnv = [FunDef] -- function environment}
\]

Based on these definitions, we implement pretty printing of functions and function environments as follows:

\[
\begin{align*}
\text{prettyFun :: FunDef \to String} \\
\text{prettyFun (ide, (args, expr)) = ide ++ concat (map (’ ‘ : ) args)} \\
& \quad + " = " + " ++ pretty expr
\end{align*}
\]

\[
\begin{align*}
\text{prettyFunEnv :: FunEnv \to String} \\
\text{prettyFunEnv funs = unlines (map (" ;" ) (map prettyFun funs))}
\end{align*}
\]

### 13.3.2 Application of User-defined Functions

Function applications can be viewed as consisting of two parts:

\[
\begin{align*}
\text{average} & \quad \text{identifier} \\
(\text{\texttt{(3 + 7) 20}}) & \quad \text{argument expressions}
\end{align*}
\]

We represent them by extending the definition of \texttt{Expr}:

\[
\begin{align*}
\text{type Ident = String} \\
\text{data Expr = Const Int -- integer constant} \\
& \mid \text{Var Ident -- variable} \\
& \mid \text{Op BinOp Expr Expr -- application of a binary operation} \\
& \mid \text{Appl Ident [Expr] -- application of a user-defined fun}
\end{align*}
\]

We also extend the pretty printing of expressions to include the application of user-defined functions:

\[
\begin{align*}
\text{pretty :: Expr \to String} \\
\text{pretty (Const i ) } = \text{show i} \\
\text{pretty (Var ide ) } = \text{ide} \\
\text{pretty (Op op e1 e2) =}
\end{align*}
\]
"(" ++ pretty e1 ++ " " ++ opToString op ++ " " ++ pretty e2 ++ ")"

where

opToString Add = "+"
opToString Sub = "-"
opToString Mul = "*"
opToString Div = "/"
pretty (Appl ide args) = ide ++ concat (map (':') (map pretty args))

13.3.3 Evaluation of Applications

Given a definition such as

\[
\text{average } x \ y = (x + y) / 2;
\]

and an expression such as \( \text{average } (3 + 7) 20 \), which steps are required to evaluate the expression?

1. Evaluate argument expressions: \([10, 20]\).
2. Lookup the identifier of the function in the function environment.
3. Build an environment from arguments: \([("x", 10), ("y", 20)]\).
4. Evaluate the body under this environment.

These steps are implemented as follows:

\[
eval :: \text{FunEnv} \rightarrow \text{Env} \rightarrow \text{Expr} \rightarrow \text{Int}
eval \text{fenv env (Const } i) = i
\]
\[
eval \text{fenv env (Var ide) } = \text{case lookup ide env of}
\]
\[
\text{Nothing} \rightarrow \text{unknownError ide}
\text{Just } i \rightarrow i
\]
\[
eval \text{fenv env (Op op } e1 e2) = \text{opToFun op}
\quad (\text{eval fenv env } e1) (\text{eval fenv env } e2)
\]
\[
eval \text{fenv env (Appl ide args) } = \text{let}
\quad \text{avals } = \text{map (eval fenv env) args in}
\quad \text{case lookup ide fenv of}
\quad \text{Nothing} \rightarrow \text{unknownError ide}
\text{Just (args, body) } \rightarrow
\quad \text{eval fenv (zip args avals) body}
\]
\[
\text{unknownError} :: \text{String} \rightarrow \text{a}
\text{unknownError ide } = \text{error ("Unknown variable \(" ++ ide ++ ");")}
\]

Appendix B.1 contains the code for the expression parser. It uses techniques from the area of formal languages that are beyond the scope of this book.
Appendix A

Common Properties

A.1 Some Basic Properties

Below is a list of properties that are common for binary operators (including the standard arithmetic ones), most of which have been used in the above proofs. Whether such a property holds for a particular operator has to be established individually for each operator, based on its definition. Here we write general operator symbols, ⊕ and ⊗—they are variables for operators and should not be confused with concrete arithmetic operations + and *.

\[
\begin{align*}
    a \oplus b &= b \oplus a & \text{(commutative)} \\
    (a \oplus b) \oplus c &= a \oplus (b \oplus c) & \text{(associative)} \\
    a \otimes (b \oplus c) &= (a \otimes b) \oplus (a \otimes c) & \text{(distributive)} \\
    e \oplus a &= a & \text{(left neutral)} \\
    a \oplus e &= a & \text{(right neutral)}
\end{align*}
\]
Appendix B

Additional Code

B.1 The Code for the Expression Parser

The following code implements a parser for expressions including variables and the application of user-defined functions as well as a parser for function definitions and lists thereof. The code is included for the sake of completeness and for the curious to study. However, it makes use of techniques from the area of formal languages that are beyond the scope of this book.

```haskell
module ParseExpr (parse, parseFun, parseFunEnv)
where

import Expr

-- parse an expression (using a left-factorised recursive descent parser)
-- * to simplify recognising the end of the input, we add '#' as an end of
-- input marker
--
parsed :: String -> Expr
parsed s | rest == "#" = expr
  | otherwise = syntaxError rest
   where
    (expr, rest) = parseAddSub (eat (s ++ "#"))

-- addition and subtraction has least precedence
--
parsedAddSub :: String -> (Expr, String)
parsedAddSub s = let
  (lhs , s') = parsedMulDiv s
  (rest, s'') = parsedAddSubRest (eat s')
  in
  (rest lhs, s'')
  where
    parsedAddSubRest s@(c:s')
      | c `elem` "+-" = let
        (rhs, s'') = parsedAddSub (eat s')
        in
        (\lhs -> Op (charToOp c) lhs rhs, s'')
      | c `elem` "()#"
        || isAlpha c
        || isDigit c
        = (id, s)
      | otherwise = syntaxError s
```

```
-- multiplication and division
--
parseMulDiv :: String -> (Expr, String)
parseMulDiv s = let
  (lhs , s') = parseAppl s
  (rest, s'') = parseMulDivRest (eat s')
in
  (rest lhs, s'')

where
parseMulDivRest s@((c:s'))
  | c 'elem' "*/" = let
    (rhs, s'') = parseMulDiv (eat s')
in
    (\lhs -> Op (charToOp c) lhs rhs, s'')
  | c 'elem' "+-()#"
    || isAlpha c
    || isDigit c = (id, s)
  | otherwise = syntaxError s

-- parse an application
--
parseAppl :: String -> (Expr, String)
parseAppl s@(c:s') | isAlpha c = parseVarOrAppl s
  | otherwise = parsePrim s

-- primitive components
--
parsePrim :: String -> (Expr, String)
parsePrim s@(c:s') | c == '(' = let
  (expr, s'') = parseAddSub (eat s')
in
  (expr, expect ')' (eat s''))
  | isIntDigit c = let
    (num, s'') = break
      (not . isIntDigit)
      (eat s)
in
    (Const(read num), s'')
  | isAlpha c = let
    (ide, s'') = break
      (not . isAlpha)
      (eat s)
in
    (Var ide, s'')
  | otherwise = syntaxError s

where
  isIntDigit c = isDigit c || c == '-'

-- parse a variable or an application of user-defined function
--
parseVarOrAppl :: String -> (Expr, String)
parseVarOrAppl s = let
  (ide, s') = break (not . isAlpha) (eat s)
  (args, s'') = parseArgs (eat s')
in
  if null args
  then
    (Var ide, s'')
  else
B.1 The Code for the Expression Parser

```haskell
(Appl ide args, s''')
where
parseArgs s@(c:s') | c `elem` ")+-*/#" = ([], s)
   | otherwise =
   let
     (arg, s''') = parsePrim s
     (args, s''') = parseArgs (eat s''')
   in
     (arg:args, s''')

-- convert the characters representing binary symbols into their internal
-- representation
--
charToOp :: Char -> BinOp
charToOp '+' = Add
charToOp '-' = Sub
charToOp '*' = Mul
charToOp '/' = Div

-- parse a function definition
--
parseFun :: String -> FunDef
parseFun s = let
    (ide, rest) = span isAlpha (eat s)
    (argsStr, '=':bodyStr) = span (/= '=') rest
    in
    if null ide
       then syntaxError (s ++ "#")
       else
           (ide, (words argsStr, parse bodyStr))

-- parse a ';' separated list of functions
--
parseFunEnv :: String -> FunEnv
parseFunEnv s | null s' = []
   | otherwise = let
     (this, rest) = span (/= ';') s'
     -- the might be a ';' after the final fun def
     rest' = dropWhile (== ';') rest
     in
     parseFun this : parseFunEnv rest'

where
  s' = eat s

-- raise an error if the next character is not the expected one
--
expect :: Char -> String -> String
expect ec s@(c:s') | ec == c = s'
   | otherwise =syntaxError s

-- eliminate leading white space
--
eat :: String -> String
eat = dropWhile ('elem' " \n\t\v\r")

-- raise a syntax error (drop terminator symbol '#')
--
syntaxError :: String -> a
```
syntaxError "#" = error ("Syntax error at end of input")
syntaxError rest = error ("Syntax error at or before ‘" ++ init rest ++ ",-")
Index

O-Notation, 108
(++) , 17
(/=), 29
(\), 17
(===), 19, 29
[ ], 25

abstraction, 22, 36
abstractions, 95
actions, 57
ad-hoc polymorphism, 29
alias, 49
appendFile, 64
Application programs, 41
argument variable, 3
arguments, 3
association lists, 32
authentication, 7
auxiliary functions, 33
AVL tree, 121

base case, 23, 82
base name, 8
bazaar style, 43
binder, 13
body, 3
Bourne Again Shell, 44
Bourne Shell, 44
BSD Unix, 42

cathedral style, 43
circular dependencies, 94
Cohesion, 94
Coincidental Cohesion, 94
command shell, 44
comments, 4
compiler, 47
compound value, 14
computations, 57
concat, 27
conditional, 12
cons, 17
Coupling, 94
daemons, 41
data constructor, 98
data constructors, 70, 73
Data Coupling, 94
data dependency, 60
declarative, 1
degenerated tree, 120
depth of a tree, 120
derivable type classes, 76
directory, 8
divide-and-conquer, 110, 118
do, 60
dropWhile, 38
elem, 31
enumeration types, 70
environment variables, 49
Eq, 29
error, 24
eval-print loop, 58
evaluation step, 104
executable, 48
expression, 2
expression evaluation, 3
extension language, 48

File, 8
FilePath, 64
filter, 38, 46
first-order functions, 38
folding, 80
fork, 48
Free software, 43
Free Software Foundation (FSF), 43
FreeBSD, 7, 43
fromInteger, 15
function definition, 3
function overloading, 29
Functional Cohesion, 94
functional language, 2
functions, 2

getLine, 59
Solaris, 7
Sorting Algorithms, 110
specification, 81
stepping case, 23
string, 2
structural induction, 85
sum, 17
superuser, 7
symbolic assembler code, 47
syntax, 1, 75
system decomposition, 93
system programs, 41
tail, 17, 25
takeWhile, 38
term, 2
toInteger, 15
total functions, 24
tree, 118
trie, 125
tuples, 14
type, 14, 33
type class, 5, 29
type constraint, 29
type constructor, 69, 98
type error, 3
type variable, 6, 28
types, 2
UID, 7
unfolding, 80
Unix, 7, 41
Unix culture, 42
user id, 7
user mode, 41
user name, 7
Values, 2
well typed, 3
where, 12
witnesses, 109
work complexity, 108
working directory, 44
writeFile, 64