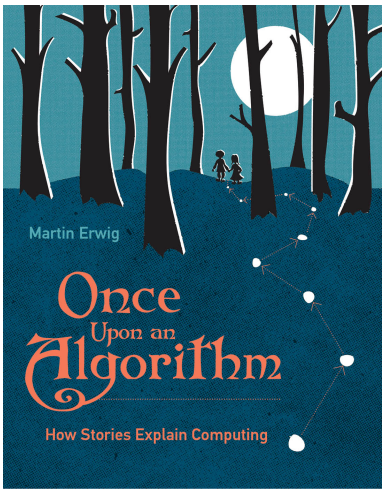
Introduction to the Haskell Code Supplement



Version 0.1, September 2017

This document serves as a guide to the Haskell code examples for the book *Once Upon an Algorithm: How Stories Explain Computing*. It explains how the concepts of computer science introduced in the book can be realized in the Haskell programming language. The guide mainly follows the structure of the book, but it also provides a brief introduction to some basic elements of Haskell. Since many of the examples are based on the examples from the book, this guide assumes that you have read the corresponding chapter before working through the code.

In addition to explaining the code and how it relates to the examples in the book, this guide also occasionally points to parts of the book for additional background information. Finally, the guide provides exercises throughout the text with solutions in the form of Haskell files, which are part of the code supplement. The exercises help readers test their understanding.

What this Guide is Not

Note that this guide is *not* a comprehensive Haskell tutorial. While I have tried to explain all concepts necessary to follow the code examples, you should not expect to become a proficient Haskell programmer just by following the examples in this document. There are many excellent tutorials (wiki.haskell.org/Tutorials) and books (wiki.haskell.org/Books) available for learning Haskell. Specifically, I will not explain Haskell error messages, which can sometimes be difficult to work with. This means that you may have to consult outside resources, such as the mailing list for beginners (beginners@haskell.org), an IRC channel (wiki.haskell.org/IRC_channel), or someone you know who has some Haskell experience.

Moreover, this document does *not* explain how to operate a computer. This guide assumes some basic computer literacy, including how to use a text editor and how to install and execute programs on a computer.

Before We Can Begin

The first step is to install a Haskell interpreter on your computer. There are several ways to do this. The easiest approach is probably to go to www.haskell.org/platform/ and follow the instructions to download and install the implementation for your operation system.

This guide assumes that you have downloaded the code examples, which are grouped according to the structure of the book; that is, all the definitions for the examples used in one chapter are usually contained in one corresponding Haskell file. (Some chapters may have multiple files.) The name of the file containing relevant definitions is indicated in the margin of this text. Solutions to exercises are contained in separate files and are also mentioned in the text margin.

Using an Editor and the Haskell Interpreter

It is important to understand that the code presented in this guide occurs in two principally different places, namely (1) in *Haskell files* and (2) in the *interpreter window*.

First, definitions of values and functions are usually placed in a file (that has the file extension `.hs`). For example, we can define a number `weeks` and a function `days` for computing the days contained in a given number of weeks and put them into a file with the name `Date.hs`. Code that appears in a file is shown indented and in typewriter font as follows:

Date.hs

```
weeks :: Int
weeks = 6

days :: Int -> Int
days w = 7 * w
```

Second, once the Haskell interpreter has been started, it presents a window into which files with definitions can be loaded and where expressions can be evaluated. In the following I am using the Glasgow Haskell Compiler, which is provided as part of the Haskell platform. After starting the interpreter,[1] we are presented with a short message, followed by an input prompt.[2]

```
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /Users/erwig/.ghci
Prelude>
```

Now we can enter expressions for the interpreter to evaluate, like so:

```
Prelude> 4+5*6
34
```

We can also load files with definitions that are then available for inspection and evaluation.[3]

```
Prelude> :l Date.hs
[1 of 1] Compiling Main             ( Date.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

For example, we can use the function `days` to compute the number of days in 52 weeks.

```
*Main> days 52
364
```

[1] On Linux and Unix systems, including Mac OS, this is done by entering the command `ghci` in a terminal window.

[2] Here `Prelude>` means that a standard set of Haskell definitions have been loaded and are available for execution.

[3] The prompt changes to `*Main>`, which is the default name for Haskell modules. Every collection of definitions in a file is considered to be a module. The module name can be changed, but this is not so important here. In the rest of this guide I will mostly omit the module name in the prompt, since it is not really needed and since it makes the examples a bit easier to read.

When we change the definitions in a file and want to use them in the interpreter, we can use the command `:r` to reload the definitions. In later parts of this guide I will explain some other features of the interpreter.

The important thing to remember is this: when definitions of values, types, and functions are shown, as for `weeks` and `days`, these appear in some file that can be loaded into the Haskell interpreter. On the other hand, expressions to be evaluated must be entered into the Haskell interpreter, which is indicated by the > prompt that precedes expressions and by the result printed on the next line.

Oh, and then there will be errors. Inevitably, you will enter an erroneous expression into the interpreter or place an incorrect definition in a file. Since machines are so picky in what input they accept, talking to a machine can be a frustrating experience. Errors can be simple misspellings, which are easy to spot and correct:

```
> week

<interactive>:1:1: error:
    * Variable not in scope: week
    * Perhaps you meant weeks (line 4)
```

But some error messages can also be more difficult to understand.

```
> days

<interactive>:2:1: error:
    * No instance for (Show (Int -> Int)) arising from a use of print
        (maybe you haven't applied a function to enough arguments?)
    * In a stmt of an interactive GHCi command: print it
```

This message essentially says that function values cannot be printed, which means functions should be applied to their arguments (an `Int` value in this case). As mentioned, you should not hesitate to consult outside resources for help when you cannot make sense of an error message. In some cases, a simple Google query will help. In other situations, try the mailing list for beginners (beginners@haskell.org) or an IRC channel (wiki.haskell.org/IRC_channel).

From Humans to Machines

In the book I can use English instead of a formalized notation to express concepts and ideas of computing. Specifically, I can describe algorithms using everyday language. This works well because the computers intended for processing these descriptions are humans who understand potentially ambiguous natural language. Humans are very good at using context information to fill in gaps and guess the intent of a definition in cases when it is not completely clear.

By contrast, machines generally lack the ability to make common-sense inferences and require precise instructions about every detail. In explaining an algorithm to a machine, vagueness is not an option, and every instruction of an algorithm must be unambiguous. To formulate an algorithm in a programming language we have to be precise about every aspect, since we can't expect the machine to fill in missing details. *Programming* is the process of translating an algorithm that is given in some non-formalized description into a language, called a *programming language*, that is understood by a computing machine. The result of this process is a *program*. In other words, a program is an algorithm that can be executed by a machine.

Concept	Haskell
Algorithm	Function
Representation	Type
Instruction/Step	Expression
Execution	Function Application

Table 1: How algorithmic concepts are realized in the programming language Haskell.

Programming is not a simple activity. Many errors and problems that arise during programming result from the often non-trivial translation of a high-level, potentially vague algorithm into a precise program. This basic fact has an important consequence for the structure and content of this guide.

On the one hand, since it is fairly easy to describe Hansel and Gretel's pebble-tracing algorithm in English, it can be presented at the beginning of the book. On the other hand, since it is much harder to describe a corresponding program in Haskell (or any other programming language), we cannot start the introduction to Haskell with this example. Therefore, to have all the features of Haskell available that are required to express the pebble-tracing program, we have to precede the program by an introduction of a number of basic Haskell programming concepts.

Any introduction to programming is mostly forced to take a bottom-up approach, since programming languages lack much of the flexibility of natural languages that afford top-down descriptions. Therefore, I start in the following section with an introduction to several basic concepts of Haskell to prepare you for the code for chapter 1.

Algorithmic Concepts in Haskell

Every algorithm describes the transformation of some representation. A representation is realized in a programming language through basic values such as numbers or symbols and data structures such as lists or trees. Therefore, to define an algorithm, we have to first identify the representations that it is supposed to transform. In Haskell these representations are given by basic types such as `Int` for numbers or `String` for text, predefined data types for lists, or types defined by the programmer specifically for the algorithm. An algorithm is realized in Haskell as a function. More specifically, an algorithm that transforms a representation that is given by a type `T` into a representation of type `U` is realized in Haskell as a function of type `T -> U`. The arrow indicates that this is the type of a function that takes as input values of type `T` and produces as a result values of type `U`. To execute an algorithm for some input arguments means in Haskell to apply the corresponding function to the values that represent the input. The mapping between algorithmic concepts and corresponding Haskell concepts is summarized in table 1.

Let's consider as an example an algorithm for determining the time for setting an alarm clock. The idea is very simple: if you plan to leave the house at some time t in the morning and the duration of getting dressed, having breakfast, etc., is d, the alarm clock should be set to go off at time $t - d$.

While the instructions needed for this algorithm are trivially simple and consist only of a single subtraction, the question of what representation to choose is not as obvious. How should we represent the time and duration values to be manipulated by the algorithm? One could,

for example, represent time as the minutes since midnight, and duration also as minutes, which means that 7 a.m. would be represented by the number 420, and the duration of 1 hour and 10 minutes would be represented by the number 70.

In Haskell, (positive and negative) whole numbers are elements of the type `Int`. An algorithm to compute the wake-up time can thus be realized through the following Haskell function `wakeUpM`,[4] which takes two values of type `Int` and produces an `Int` value as a result.[5]

WakeUp.hs

```
wakeUpM :: Int -> Int -> Int
wakeUpM leave duration = leave - duration
```

The first line defines the type of the function, that is, the type of its parameters and its result. The symbol `::` can be read as "has type," and the arrow symbol `->` separates the types of parameters from each other and from the type of the result.

The second line defines what the function `wakeUpM` does when it is applied to two arguments. These arguments are represented in the definition by the two parameter names: the first parameter `leave` stands for the time to leave the house, and the second parameter `duration` stands for the time required to complete the activities from getting up to leaving the house. The two parameters are introduced directly following the function name, and the definition of the function result is given by the expression `leave - duration` that follows the = sign.[6]

If we place this function definition into a file, say `WakUp.hs`, we can load it into the Haskell interpreter. Starting the Haskell interpreter `ghci` from the command line will produce an output similar to the following.[7]

```
> ghci WakeUp.hs
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from /Users/erwig/.ghci
[1 of 1] Compiling Main             ( WakeUp.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Now that we're inside the Haskell interpreter, we can "talk Haskell" to it. We do this mainly by applying functions to arguments, adding definitions, or reloading files after changing definitions. In the example, we can compute wake-up times for different time and duration values by applying the function `wakeUpM` to corresponding integer values. For example, if we want to leave the house at 7 a.m. and we need 1 hour and 10 minutes for our daily morning routine, we can execute the algorithm by applying `wakeUpM` to the the values `420` and `70`.[8]

```
> wakeUpM 420 70
350
```

How is this result actually obtained? One can think of the arguments as being substituted for the

► *Parameters*
Chapter 2

[4] The trailing `M` stands for "**m**inutes." We will encounter several variations of this function later.

[5] In Haskell the names of functions, parameters, and other variables must start with a lowercase letter, whereas all type names must start with an uppercase letter.

[6] Having seen function definitions in mathematics, one might expect a different syntax, maybe something like `wakeUpM(leave,duration) = leave - duration`. Such a definition is actually also possible in Haskell, but it has a slightly different type that restricts how the function can be used. The shown definition is more general. In particular, it allows the partial application of functions (see section 1.1). It is the standard way of defining functions in Haskell.

[7] Starting `ghci` with a Haskell file name as argument has the same effect as starting `ghci`, followed by loading the file using the `:l` interpreter command.

[8] As mentioned earlier, I will omit module names in the interpreter prompt when showing examples.

corresponding parameters in the function definition, which then yields an expression that can be evaluated. In the example, if we substitute `420` for `leave` and `70` for duration, we obtain the following expression, which evaluates to `350`.

```
420 - 70
```

The result `350` is not surprising, but it is also only of limited use. What time do we have to set the alarm clock to now? Since few alarm clocks allow entering only minute values as alarm times, we have to convert the value back into hours and minutes. This is not hard, but it is inconvenient, which indicates that we may not have chosen the best representation for the computation. We could use floating point numbers (of type `Float`) instead of integers to represent time and duration as fractions of hours.[9] However, this doesn't really help very much. First, representing `1` hour and `10` minutes as a `Float` number requires some calculation. Fortunately, we don't have to do this ourselves: we can actually delegate the work to the Haskell interpreter.

```
> wakeUpH 7 (1+10/60)
5.8333335
```

While we didn't have to compute anything for specifying the input arguments, the result leaves us with the task of converting a `Float` number back into hours and minutes.

Maybe we need yet another representation. In particular, why not represent time and duration values by pairs of numbers? Let's try this approach next. A pair of values is written in Haskell like in mathematics. For example, we can write `(7,0)` for the pair of numbers `7` and `0` that represent `7` a.m. and `(1,10)` for `1` hour and `10` minutes. Since we don't need to represent fractional values anymore, we can use integers in these pairs. In Haskell the corresponding type is written as `(Int,Int)`, which can be read as: "this is a type of pairs whose elements are integers."

Note that we actually do not expect arbitrary integers to appear in time pairs. Specifically, for a `24`-hour representation we expect for each pair of integers `(h,m)` that $0 \leq h \leq 23$ and $0 \leq h \leq 59$. For the following definitions we simply assume that these constraints are satisfied. We could easily implement a function that checks these restrictions, see exercise 5.

To change the function definition to the new pair type, we could again replace each occurrence of `Float` or `Int` in the type by `(Int,Int)`, but the repeated use of the pair type as well as the fact that it is not clear what each `Int` component represents suggests to introduce a *type definition*, which introduces a name for a type that can be used in its place. Here is one way of doing it.

```
type Time = (Int,Int)
```

This definition allows us to use the name `Time` in the type of the function, but it doesn't explain what role the two components play. Therefore, we could add two further type definitions for hours and minutes and adapt the definition of `Time` accordingly.

```
type Minute = Int
type Hour   = Int
type Time   = (Hour,Minute)
```

► *Types* Chapter 14

Note that these definitions are not strictly needed; omitting the type definition(s) and simply using `(Int,Int)` instead of `Time` works just as well. But type definitions often make programs and

[9] To achieve this, we can either change the types of the existing function `wakeUpM` or define a second function `wakeUpH` (where the trailing `H` stands for "**h**our").

their intentions clearer. With the new pair representation of time we can now try to implement another version of the `wakeUpM` function. Since subtraction is only defined for numbers and not for pairs, we need to change the definition of the function. In a first attempt we naively try to subtract hours and minutes individually.

```
wakeUpNaive :: Time -> Time -> Time
wakeUpNaive (h,m) (h',m') = (h-h',m-m')  -- incorrect definition
```

Testing this definition with our example values immediately reveals a flaw.

```
> wakeUpNaive (7,0) (1,10)
(6,-10)
```

The result is obviously incorrect, since there are no time values with negative numbers. This "underflow" error occurs whenever the minutes to be subtracted exceed the minute value of the time from which they are to be subtracted. (An underflow can also happen for the hour values.) There are at least two ways to fix this problem. One is to identify the situations that would lead to an underflow and adjust the result accordingly. Specifically, whenever `m'` is larger than `m`, we need to subtract an additional `1` from the hours `h`. We must also adjust the minute value by adding `60`, since we are effectively trading `1` hour against `60` minutes. This is essentially the same thing that happens during the subtraction algorithm for decimal numbers where one "borrows" `10` to facilitate subtraction of a larger digit from a smaller one. The difference here is that one hour is equivalent to `60` minutes.

To express the idea of producing alternative results in Haskell we need some way of returning different results depending on a condition. Here, a condition is an expression that evaluates to either `True` or `False`, which are values of type `Bool`. An example is the expression `3<4`, which checks whether `3` is less than `4`. Since this is the case, the expression evaluates to `True`.

► *Conditional*
Chapter 10

Haskell provides two ways of selecting between two results based on a condition. One is to use an `if-then-else` construction, which leads to the following definition.

```
wakeUp :: Time -> Time -> Time
wakeUp (h,m) (h',m') = if m'>m then (h-h'-1,60+m-m')
                               else (h-h',m-m')
```

The other approach is to split the equation into two parts and prefix each result case with a condition. All cases are tried in order from top to bottom, and the first case whose condition yields `True` is selected.

```
wakeUp :: Time -> Time -> Time
wakeUp (h,m) (h',m') | m'>m      = (h-h'-1,60+m-m')
                     | otherwise = (h-h',m-m')
```

The latter style is often more readable, in particular, when multiple conditions are needed that lead to more than two different results.

Let's test the new definition with our example. Since we are using pairs of integers to represent time, we have to give the arguments to the `wakeUp` function in this form and obtain the result in the same way.

```
> wakeUp (7,0) (1,10)
(5,50)
```

One might prefer to see the result printed in a more conventional form such as `5:50 a.m.`. This is easy to achieve, but doing so here would distract from the mission of this section.

The pair representation for time has made it easier to use the function and, in particular, interpret the result, but it has made the implementation more complicated. In general, we can observe:

(a) Most problems can be solved with different representations.
(b) The choice of representation affects the implementation and use of functions.

An alternative approach to implementing `wakeUp` is to map the `Time` arguments to minutes, perform subtraction with these minute values, and then convert the result back into a `Time` value.

WakeUpS.hs

Exercise 1

Define the following two functions to convert between the hour/minute and minute representation of time.

```
timeToMinutes :: Time -> Minute
minutesToTime :: Minute -> Time
```

The functions should have the property that they are each other's inverses, that is, for any valid time value `t` (that is, any pair `(h,m)` with $0 \leq$ `h` ≤ 23 and $0 \leq$ `h` ≤ 59), the following equation should hold: `minutesToTime (timeToMinutes t)` = `t`. Likewise, for any minute `m` between 0 and 1439 the following equation should hold: `timeToMinutes (minutesToTime m)` = `m`.

The condition mentioned in the exercise also demonstrates how to apply a function to the result computed by another function. The combination of function applications is how multiple steps in an algorithm are executed in a language like Haskell: if you want to transform a value using a function `f` and then transform the returned result further by another function `g`, you can simply apply `g` to the result of `f` as in `g (f x)`. When combining multiple steps in this way, as in the example `k (h (g (f x)))`, one can make nested function applications more readable by introducing names for intermediate results. For example, we can rewrite the last example as follows:

```
let y = f x
    z = g y
    a = h z
 in k a
```

This is particularly useful in cases where an intermediate result is used more than in one place. For example, the application `g (f x) (f x)` can be conveniently be rewritten as follows:

```
let y = f x
 in g y y
```

Not only is this convenient and less error-prone in cases we have to use a large, complicated expression more than once, it also has the added benefit that the computation of the named

expression is performed only once, which saves computing time.

Exercise 2

Give an alternative implementation for the function `WakeUp` that uses the two functions `timeToMinutes` and `minutesToTime`. (**Hint**: The expression `div n m` computes how often `m` fits into `n`, and the expression `mod n m` computes the rest that remains when dividing `n` by `m`.)

For the following exercise we have to understand the type `Bool` of boolean values, which consists of the two values `True` and `False`. These values are used to represent the outcome of conditions, such as `m'>m` that is used in the definition of the function `wakeUp`. In addition to predefined test and comparison operations, we can define our own functions to represent tests. For example, here is the definition of a function that tests whether an integer is positive. Note that the type specifies that `positive` applies to integers but returns boolean values.

```
positive :: Int -> Bool
positive x = if x>=0 then True else False
```

As with the definition of `wakeUp`, we can split the equation using the condition and obtain an equivalent definition that uses two cases for the equation.

Exercise 3

Give an alternative definition of the function `positive` that uses a condition to split the equation into two cases.

We can observe something peculiar in the definition of `positive`: The definition effectively says that if the condition `x>=0` evaluates to `True`, then return `True`, and if `x>=0` evaluates to `False`, then return `False`. This seems to be redundant in the sense that the condition already provides the value that should be returned by the function. In other words, it seems that we can save the `if-then-else` construct (or the split equation) and return the expression itself as a result. We are thus tempted to define the function simply as follows, simply using the condition itself as a result.

```
positive :: Int -> Bool
positive x = x>=0
```

This definition says that `positive x` returns the same result as the expression `x>=0`, which is exactly what we want.[10]

Exercise 4

Define the following two functions that check whether an integer is a valid minute or hour value. A minute value is valid if it is in the range 0 to 59, and an hour value is valid if it is in the range 0 to 23. (**Hint**: The expression `c && c'` yields `True` if both `c` and `c'` evaluate to `True`.)

```
validMinute :: Minute -> Bool
validHour :: Hour -> Bool
```

[10] The definition can actually be further simplified by exploiting the fact that functions can be partially applied to only some of their arguments and then return functions as a result. For binary operations this is also called *sectioning* and works by enclosing the operations in parentheses and filling the left or right argument, which allows the simplified definition `positive = (>=0)`, see also section 1.1.

We can now combine the two functions to define a test for valid time values.

Exercise 5

Implement the function `validTime :: Time -> Bool` that takes a `Time` value `(h,m)` and returns `True` if `h` and `m` are valid hour and minute values, respectively.

❋ ❋ ❋

The purpose of this section is the introduction of some basic concepts of Haskell that are needed to express algorithms. The chosen example is necessarily very simple. In particular, the algorithm itself consists of only one step,[11] and the employed representation consists of numbers or pairs of numbers. In the next section I demonstrate how to implement more complicated algorithms that require a mechanism to express the repetition of steps and that use lists as a representation for collections of values.

1 A Path to Understanding Computation

Chapter 1 is centered around Hansel and Gretel's algorithm for finding their way back home from the forest. The algorithm consists of steps to move along a path of pebbles that Hansel had dropped on the way into the forest. Any algorithm works by transforming a representation, and the pebble-tracing algorithm transforms the position of Hansel and Gretel as they move from pebble to pebble. To emulate this behavior with a computer we first need to define a representation for the position of Hansel and Gretel and the positions of all the pebbles.

A position in the two-dimensional plane can be represented by a pair of floating point numbers. Such a pair is called a *coordinate*. Typically, the first component in a coordinate is used to define the horizontal position, and the second component defines the vertical position. We can assign some arbitrary position to Hansel and Gretel's home.

Chapter1.hs

```
type Pos = (Float,Float)

home :: Pos
home = (3.0,4.5)
```

Before we can define a function for Hansel and Gretel's pebble-tracing algorithm, we have to develop a number of auxiliary definitions. We start by looking into different ways to move around. Moving from a given position `n` units to the right (or east) can be achieved by the following function.

```
moveRightBy :: Float -> Pos -> Pos
moveRightBy n (x,y) = (x+n,y)
```

It is instructive to contemplate the type of `moveRightBy` in more detail. The function has two parameters, the distance to move (a `Float` value) and the position from which to move (a `Pos` value). While it makes sense to separate a function parameter type from its result type by an arrow (indicating the dependency of resulting values on the arguments supplied for the parameter), it seems peculiar that the two parameters are separated by an arrow as well.

► *Parameters* Chapter 2

[11] The algorithm in exercise 2 consists of four steps.

1.1 Multiple Parameters and Partial Function Application

The rationale for this notation is that a function of two parameters of types `A` and `B` that returns something of type `C` is actually the same as a function of one parameter of type `A` that returns a function of type `B -> C`.[12] In other words, if we apply a function with two parameters to only one of its arguments, the result is a function of the remaining argument. This is called *partial function application*, and the style of writing functions in this way is called *currying*. In the example, this means if we apply `moveRightBy` to only a float value, say `3`, we obtain a function that takes a position and transforms it by adding `3` to the horizontal component. We can test this in Haskell as follows. We can define `moveRightByThree` as the result of the application `moveRightBy 3`, and observe that it is indeed a function of type `Pos -> Pos`. Note that the `let` command of the interpreter extends the definitions currently available; it can also hide previous definitions by providing a new definition for an already existing name. And the `:t` command shows the type of any definition known to the interpreter.

```
> let moveRightByThree = moveRightBy 3
> :t moveRightByThree
moveRightByThree :: Pos -> Pos
```

We can also easily convince ourselves that applying `moveRightByThree` to a position `p` yields the same result as the application of `moveRightBy` to `3` and `p`. We can test this using concrete positions, such as `home`.

```
> moveRightByThree home
(6.0,4.5)
> moveRightBy 3 home
(6.0,4.5)
```

We can even let Haskell do the comparison (note that == is the operation for testing the equality of two values).

```
> moveRightByThree home == moveRightBy 3 home
True
```

But maybe this relationship holds only for this particular case. How can we be sure that it is true for any position and any movement? This is obvious if we substitute the definition of `moveRightByThree`.

► *Substitution*
Chapters 13, 15

```
  moveRightByThree p
= moveRightBy 3 p          -- substitute definition of moveRightByThree
```

Thus we have seen that `moveRightBy` can be viewed as a parameterized operation on positions: once it is provided with a specific distance, it turns into an operation that transforms `Pos` values.

Chapter1S.hs

Exercise 6

Define the function `moveUpBy :: Float -> Pos -> Pos`.

It is easy to see that we can use the function `moveRightBy` also to move to the left by simply providing a negative number as a first argument. Similarly, we can use `moveUpBy` to move down.

[12] In math, the two types $A \times B \to C$ and $A \to B \to C$ are said to be *isomorphic*.

1.2 Multiple Steps through Composing Function Applications

► *Control Structures* Chapter 10

Simple horizontal and vertical movements can be employed as individual steps within larger algorithms, which are obtained through composition of steps. The most simple form of composition is to execute one step after another. Since the execution of a single step corresponds in Haskell to the application of a function, such a sequential composition corresponds to the composition of functions that represent the individual steps. For example, we can express moving 4 units up followed by moving 3 units to the right by the following expression.

```
> moveRightBy 3 (moveUpBy 4 home)
(6.0,8.5)
```

This might be a bit difficult to parse. In particular, it may seem strange that the second step of moving right precedes the first one of moving up. We could untangle the expression using a `let` expression that defines an intermediate position `p`.

```
let p = moveUpBy 4 home
 in moveRightBy 3 p
```

This formulation highlights the fact that the expression `moveUp home 4` is computed first and that its result is a position that is subsequently transformed by `moveRight`.

This still doesn't look great. Specifically, it is not what you might expect if you have seen algorithmic notation or programs that operate by assuming the modification of some underlying state. The reason for the more clumsy seeming notation is that functional programming languages such as Haskell make the values that are subject to transformation explicit. Specifically, if the transformation affects an underlying state, the state representation has to be passed explicitly into functions as arguments and returned from functions as results, and this is what shows up in the notation. Other language approaches leave state arguments implicit, which leads to a simpler notation. The advantage of explicit state arguments and results is that it facilitates the reasoning about programs. There are several ways to address the notational inconvenience, but this would be a distraction from the goal of this section, and it ultimately doesn't affect the formulations of algorithms too much anyway.

Exercise 7

Define the function `moveBy :: Float -> Float -> Pos -> Pos` that takes two `Float` values and a position and returns the position obtained by moving horizontally (as indicated by the first `Float`) and vertically (as indicated by the second `Float`).

Instead of moving a certain distance from a given position, we can also envision movements directly to a new position. The following definition captures this idea.

```
moveTo :: Pos -> Pos -> Pos
moveTo p _ = p
```

The first argument `p` represents the new position to move to, which is also the result of the function `moveTo`, no matter what the second `Pos` value is (which represent the current position). The definition uses an underscore instead of a name for the second parameter to indicate that it is not used in the definition.[13]

[13] We can use a name for the second parameter, but the use of the underscore emphasizes the fact that the second parameter is not used in the definition.

With the function `moveTo` we can move to a specific position. The idea is to use this function in the pebble-tracing algorithm to repeatedly move to the next visible pebble on the path home. To realize this idea we need to answer several questions.

- How do we represent the pebbles dropped by Hansel?
- How do we identify the next visible pebble?
- How do we express the repeated application of a function?
- How do we represent the movement of Hansel and Gretel from one pebble to the next?

1.3 Representing Collections with Lists

► *Lists*
Chapter 4

To start with the first question, a collection of pebbles can be represented in several different ways. A straightforward approach is to use a list data structure. In Haskell, a list can be simply written down by listing all of its elements. For example, we can define the following list that contains three positions.

```
pebbles :: [Pos]
pebbles = [(9.0,8.0), (6.0,8.5), home]
```

Note the type declaration in the first line: it says that `pebbles` is a list of positions. For any type `T`, the type `[T]` denotes the type of lists that contain elements of type `T`. Note that lists in Haskell are homogeneous, that is, all elements in the list have to have the same type.

In addition to listing all elements at once, we can also build lists incrementally by adding elements at the front of an existing list, which is done with the so-called *cons* operator `:` as follows:

```
> (11.0,9.5):pebbles
[(11.0,9.5),(9.0,8.0),(6.0,8.5),(3.0,4.5)]
> pebbles
[(9.0,8.0),(6.0,8.5),(3.0,4.5)]
```

Note how `home` has been replaced by its definition and that the spaces after the commas have been removed: it doesn't matter how exactly we denote values; the Haskell interpreter uses an internal representation and prints values in a normalized way.

Exercise 8

Define the value `pebbles` using only the cons operator `:`. (**Hint**: You have to start with the empty list, which is written as `[]`.)

As the reevaluation of `pebbles` shows, the expression `(11.0,9.5):pebbles` constructs a new list but does not change the existing definition of `pebbles`. If we wanted to add the new coordinate to the list `pebbles` we have to add it to the definition in the program or define a new value in the interpreter using the `let` command of the interpreter.

```
> let morePebbles = (11.0,9.5):pebbles
> morePebbles
[(11.0,9.5),(9.0,8.0),(6.0,8.5),(3.0,4.5)]
```

Lists can be inspected and decomposed in many different ways. The function `head` computes a list's first element, and the fucntion `tail` computes the list obtained by removing the first element.

```
> head pebbles
(9.0,8.0)
> tail pebbles
[(6.0,8.5),(3.0,4.5)]
```

The function `reverse` reverses the order of elements in a list, and `++` appends two list.

```
> reverse pebbles
[(3.0,4.5),(6.0,8.5),(9.0,8.0)]
> pebbles ++ pebbles
[(9.0,8.0),(6.0,8.5),(3.0,4.5),(9.0,8.0),(6.0,8.5),(3.0,4.5)]
```

Of course, we can also compose list functions. For example, we can access the second element of a list by computing the head of the tail of a list.

```
>  head (tail pebbles)
(6.0,8.5)
```

But we cannot compute `tail (head pebbles)`, since `head pebbles` produces a pair of `Float` values, whereas `tail` requires a list as an argument.

Exercise 9

Define the function `last :: [Pos] -> Pos` that computes the last element in a list of positions by composing the functions `head` and `reverse`.

Having chosen lists as a representation for a collection of pebbles, we can now address how to identify the next pebble. The elements in a list are ordered but not necessarily in the way Hansel and Gretel would find them during their search. Let us assume for simplicity that Hansel and Gretel always find the pebble that is closest to their current position. How could we identify that pebble in the list? Since we do not know a priori which pebble is the closest, we have to traverse the list and compute the distance between Hansel and Gretel's position and each pebble and then pick the one for which this result is smallest.

A function for computing the distance between two positions is obtained by a straightforward encoding of the formula for the distance between two points in the plane. (Note that `x^2` computes the square of `x` and that `sqrt` is the function for computing the square root of a number.)

```
distance :: Pos -> Pos -> Float
distance (x1,y1) (x2,y2) = sqrt ((x2-x1)^2 + (y2-y1)^2)
```

Next we can apply different strategies to identify the closest pebble. Any approach has to traverse the list and compute for each pebble the distance to Hansel and Gretel's position. However, there are different ways to then determine the closest pebble. One approach is to remember the currently closest pebble and its distance and update this information while traversing the list. This approach is in the spirit of imperative programming that views computation as state manipulation. A slightly different approach is to first compute all distances, then find the smallest one of these, and finally traverse the list again to find the position whose distance is equal to the minimum distance. In the following, I will describe this approach, since it reflects the style of functional programming that emphasizes the composition of functions and values.

1.4 Different Ways to Transform Lists

Functional programming offers several different high-level operations to transform lists (and other collections). One is offered by the function `map` that applies a function to all elements in a list. For example, we can increment all numbers in a list by 1 by applying the function `succ` for computing successors of integers to all the elements in the list.

```
> map succ [3,4,5,6]
[4,5,6,7]
```

Or we can compute the distances of all pebbles to a position, say `(3,2.5)` by applying the function `distance (3,2.5)`[14] to all elements in the list `pebbles`.[15]

```
>  map (distance (3,2.5)) pebbles
[8.13941,6.708204,2.0]
```

Haskell provides a special syntax, called *list comprehensions*, that can make applications of `map` more convenient and can simplify the combination with other computations.

```
> [distance (3,2.5) p | p <- pebbles]
[8.13941,6.708204,2.0]
```

The expression can be read as follows: "For each position `p` taken from the list `ps`, compute `distance (3,2.5) p`." All the results are collected in a list that is the result of the expression. (List comprehensions offer a number of additional features, some of which we will employ soon.)

The other major approach for processing lists is called *folding*, which is repeatedly combining two elements from a list into one element. This process combines the values in a list to a single value.[16] A simple example of list folding is the definition of the function `smallest` for computing the smallest number in a list that uses the predefined Haskell function `min` to compute the smaller of two elements.

```
smallest :: [Float] -> Float
smallest [x]    = x
smallest (x:xs) = min x (smallest xs)
```

The definition considers two cases: First, when the list contains only a single element, that element is the list's smallest element. Second, in the general case, when the list is not empty, the smallest element is the smaller of the first element `x` and the smallest of the elements in the remaining list `xs`.

► *Recursion* Chapter 12

The second case illustrates an important feature of computing: the use of *recursion*. In a recursive function definition the name of the function being defined is used in its own definition. This happens here with `smallest`. The meaning in this case should be clear: in order to compute the smallest element of a list, we can compute the smallest element of the rest list and then take the smaller of that result and the first element of the list. An equation of a function definition that uses recursion is also called a *recursive* or *inductive* case of the definition, and an equation that does not use recursion is called a *base case*. Note also that the function `smallest` is undefined for empty lists, and will produce an error when applied to an empty list (see also box Error Handling.)

[14] Note that we have to put the partial function application in parentheses because otherwise `map` would try to apply the function `distance` to `(3,2.5)`, which is a pair and not a list.

[15] Remember that while `distance` applied to two arguments yields a `Float` value, `distance` applied to only its first argument `p` yields a function that takes a position and returns the distance to `p`.

[16] Folding is also often referred to as *aggregation* or *reduction*.

Error Handling

This introduction to Haskell mostly ignores the handling of errors in functions, that is, when functions are undefined for some values of their input types, they will fail with an error message or, worse, not terminate. It is good software engineering practice to handle errors explicitly to avoid unexpected behavior and surprising error messages. However, this often complicates the code and distracts from the core of algorithms.

For example, executing `smallest []` will fail with an error message. Errors can be dealt with by changing the result types of functions to account for erroneous situations, but this requires that all calls of such functions have to explicitly deal with the two different cases. Since the major purpose of this introduction is to illustrate computer science concepts, error handling code is omitted in most cases for clarity.

Exercise 10

Define the function `myMin :: Float -> Float -> Float` for computing the smaller of two `Float` values. (The function `min` is already predefined in Haskell, and the attempt to redefine it would lead to an error.)

Equipped with the function `smallest`, we can now define a function for finding the pebble that is closest to a given position. The function `closestTo` operates in basically four steps, implementing the strategy outlined before. First, we compute the distance of all pebbles from `c`, which represents the current position of Hansel and Gretel. This is done by the list comprehension `[distance c p | p <- ps]`, which produces a list of `Float` values. Second, we compute the smallest value of this list using the function `smallest`, and we bind the result to the variable `d` using the `let-in` construct. Third, we compute the list of all pebbles whose distance to `c` is exactly the smallest distance `d`. This is done in a second list comprehension `[p | p <- ps, distance c p == d]`, which filters out all those `p` taken from `ps` that satisfy the condition `distance c p == d`. This list comprehension computes a list of potentially several positions that are all equally close to `c`. Since we need one position, we select the first of these using the function `head`.

```
closestTo :: Pos -> [Pos] -> Pos
closestTo c ps = let d = smallest [distance c p | p <- ps]
                 in head [p | p <- ps, distance c p == d]
```

Now we can find the closest position from a list of pebbles that Hansel and Gretel can move to. It is not so obvious, however, how to combine the list of pebbles, the function `closestTo`, and the function `moveTo` to describe the movement of Hansel and Gretel along the pebbles in the list.

1.5 Repeating Steps through Recursion

For implementing the pebble-tracing algorithm, it is again instructive to think about the representation that the algorithm has to transform. In the following we will consider several attempts that ultimately lead to the solution.

The most obvious transformation is the movement of Hansel and Gretel from one position to another. Since the next position is found in the list of pebbles, it seems the input to the function implementing the algorithm should be a position and a list of pebbles, and the output should be

a new position. If we look closely at the type of this function `Pos -> [Pos] -> Pos` and think of what it should do, we notice that we already have defined it: it is, in fact, the function `closestTo`.

Next we have to figure out how to apply this function repeatedly. There are several ways to do this. In the style of functional programming, we can define a function using recursion, which we have already encountered in the definition of the function `smallest`. This definition has to consider two cases. First, if Hansel and Gretel are already home, the task is finished, and no further transformation of their position is required. Otherwise, the function should continue using the position from the pebbles that is closest to the current one.

The following function definition realizes these two cases using a split equation based on the condition `c==home`.[17] The parameter `c` represents the current position of Hansel and Gretel, `home` is the position defined earlier, representing the location of their home, and == tests whether the two are equal.[18]

```
findHome1 :: Pos -> [Pos] -> Pos
findHome1 c ps | c==home   = home
               | otherwise = findHome1 (closestTo c ps) ps
```

The interesting case is the second part of the equation where `findHome1` is recursively applied to the position that is closest to the current position `c`. The function application `closestTo c ps` represents the first step in the algorithm, namely going to the position of the next pebble, and the recursive application of `findHome1` represents the continuation of the algorithm, that is, all the remaining steps until Hansel and Gretel reach the position represented by `home`.

The shown definition looks reasonable and seems to make sense. Without reading ahead, can you spot what's wrong with it? To see what the problem is, apply the function to a location, say `(10,10)`.

```
> findHome1 (10,10) pebbles
```

Surprisingly, this computation does not terminate. (You can abort a computation by pressing `Control+C` in the interpreter.)

1.6 When Computations Don't Terminate

Instead of just telling you why the definition doesn't terminate, let's change the definition of the function `findHome1` so that it reveals more about its behavior and gives us a clue about what's going on. The function's behavior is reflected by the the position it traverses. Therefore, we can change the definition such that it doesn't just produce the final position reached, but the list of all traversed positions. To do this we first have to change the function's result type to return a list of positions `[Pos]`. Then we have to adapt the two return expressions of the split equation to produce the appropriate lists. In the base case, when we're already home, the list consists of just the final position, `home`, reflected by the return result `[home]` that represents a list of just that one position. Otherwise, the list of traversed positions is given by the current position `c`, followed by the list of positions from the next position to `home`, which is computed recursively as before. The resulting function definition looks as follows:[19]

[17] You may notice that the function definition relies on the position `home` to be part of the list `ps`. We will relax this assumption in exercise 13.

[18] The chosen name for the function `findHome1` indicates that this is only the first version; we encounter several alternative definitions in the following.

[19] The `T` appended to the name stands for "trace," indicating that the function computes a trace of positions.

```
findHome1T :: Pos -> [Pos] -> [Pos]
findHome1T c ps | c==home   = [home]
                | otherwise = c:findHome1T (closestTo c ps) ps
```

We can apply `findHome1T` to compute a list of positions.

```
> findHome1T (10,10) pebbles
[(10.0,10.0),(9.0,8.0),(9.0,8.0),(9.0,8.0),(9.0,8.0), ...
```

Like `findHome1`, this function will not terminate and quickly fill the screen. To see only the beginning of the list, we can select its first seven elements using the built-in Haskell function `take` as follows:

```
> take 7 (findHome1T (10,10) pebbles)
[(10.0,10.0),(9.0,8.0),(9.0,8.0),(9.0,8.0),(9.0,8.0),(9.0,8.0),(9.0,8.0)]
```

► *Termination*
Chapter 11

This reveals what the problem is. The function is stuck on the first position in `pebbles`.[20]

Exercise 11

Termination of an algorithm or program generally depends on the input on which it is run.

(a) Is there a list of pebbles that can we use as a second argument so that `findHome1` (and `findHome1T`) will always terminate?
(b) Is there a situation in which the function application of `findHome1` (or `findHome1T`) using the list `pebbles` does, in fact, terminate?

Why is the program stuck on the first position it finds in `pebbles`? The answer can be found by looking at the definition of `closestTo`, which computes the closest position in the list `pebbles` to the current position. Suppose the closest position found in `pebbles` is `p`. The function then computes in the first recursive call `closestTo p pebbles`. What is the result? Of course, it is `p` since the distance from `p` to itself is always `0`.

To fix this problem, we can try to amend the definition of `findHome1` so that when we compute the closest position to `c` we disregard the current position. This can be achieved by using a list comprehension that selects all positions from `ps` that are different from `c`.[21]

```
findHome2 :: Pos -> [Pos] -> Pos
findHome2 c ps | c==home   = home
               | otherwise = findHome2 (closestTo c [p | p <- ps, p/=c]) ps
```

Unfortunately, this definition still does not terminate, which can be easily verified by trying to compute `findHome2 (10,10) pebbles`. Again, to see what's going on, we can define a tracing version of this function using the same strategy employed in `findHome1T`.

```
findHome2T :: Pos -> [Pos] -> [Pos]
findHome2T c ps | c==home   = [home]
                | otherwise = c:findHome2T (closestTo c [p | p <- ps, p/=c]) ps
```

If we compute the first few elements of the sequence, we can observe a back-and-forth between two positions.

[20] If you are skeptical and suspect that this is not the case, try to compute longer lists with `take 20` or `take 100`.
[21] The condition `c/=p` is `True` if `c` is not equal to `p`.

```
> take 7 (findHome2T (10,10) pebbles)
[(10.0,10.0),(9.0,8.0),(6.0,8.5),(9.0,8.0),(6.0,8.5),(9.0,8.0),(6.0,8.5)]
```

The problem arises once the distance to the next pebble is larger than the one from the previous one, in which case `closestTo` identifies the previous position as the next.

The solution to this problem is to remove any position moved to from the list of pebbles so that it can't be reached again. This implementation amounts to Hansel and Gretel's picking up the pebbles as they encounter them.

Removing the elements in `ys` from `xs` amounts to computing the difference between two lists `ys` and `xs`, which can be computed using the built-in Haskell function `\\`. With this function we can simply remove the current position from the currently remaining list of pebbles by the expression `ps \\ [c]`. Note that we have to enclose `c` in square brackets to turn it into a one-element list, since `\\` requires two lists as argument.

```
findHome3 :: Pos -> [Pos] -> Pos
findHome3 c ps | c==home   = home
               | otherwise = findHome3 (closestTo c ps) (ps \\ [c])
```

The function `findHome3` finally accomplishes the goal of computing a path along a set of positions given by a list based on distance, which can be easily verified by applying `findHome3` to different inputs. And to see the positions traversed by `findHome3`, one can easily define a tracing version as done for `findHome2` and `findHome1`.

Exercise 12

Does a computation `findHome3 c ps` always traverse all the positions in `ps`? Justify your answer.

The program `findHome3` assumes the the position of Hansel and Gretel's home is contained in the list `pebbles`, because otherwise it would not be clear how to formulate the condition for the function to terminate. Such a dependency of an algorithm on its input is bad programming style. We get rid of this dependency in the following exercise.

Exercise 13

First define the function `findPath` that takes two position parameters, the start and the end position, plus the parameter for the pebbles and finds the path from the start to the end position.

```
findPath :: Pos -> Pos -> [Pos] -> Pos
findPath from to ps = ...
```

Using the function `findPath`, define the function `findHome` that finds a path from a given start position to the position `home`.

```
findHome :: Pos -> [Pos] -> Pos
findHome c ps = findPath ...
```

While the function `findHome3` correctly computes the path home, its definition contains a minor

flaw, which may not be easy to spot.

Exercise 14

Define the function `findHome3T` analogous to the other two tracing functions, and then execute the following:

```
> findHome3T (1,1) [(2,2),(3,3),home]
```

Identify the problem, and then define a function `findHome4T` that doesn't suffer from it.

For simplicity the repeated steps from pebble to pebble that are expressed as a loop in the algorithm are represented in the `findHome` functions using recursion. The algorithm can also be expressed as a program that uses a loop, which reflects the original algorithm more closely but is also a bit more complicated. The relationship between loops and recursion is discussed in sections ?? and ??.

❋ ❋ ❋

In this section we have seen how the implementation of an algorithm requires the careful design of the underlying representation that it is transforming. This is done through the definition of types, which guide the function definitions. I discuss representations in more detail in sections 3 and 4. First, however, we consider in the next section the question of runtime and space requirements of computation.

2 Walk the Walk: When Computation Really Happens

The previous section is focused on how to implement an algorithm in Haskell, which involves the identification of the appropriate representation and the definition of a function that correctly maps any possible input to the desired output. We have seen that this is not always an easy task. A particular challenge is to not make any unwarranted assumptions.

In addition to correctness, another critical aspect of computation is its cost. Any algorithm or program requires time to execute and space to store the manipulated representations. While a program that computes incorrect results is of limited use (or might be even dangerous), a program that is correct but takes too long to execute or requires resources beyond what is available is also effectively useless.

Chapter 2 of the book illustrates how the use of parameters facilitate the description of algorithms that work for different inputs. This aspect has already been covered and thus doesn't need to be explained again. In addition, Chapter 2 discusses the resource requirements of algorithms, and this section provides code examples to illustrate this aspect.

2.1 How to Measure Runtime

Typically, the runtime of an algorithm is measured as a function of the size of its input. Abstracting from different inputs and expressing the runtime as dependent on the input ensures that the runtime information characterizes the algorithm performance for arbitrary inputs and not for a particular one.

Determining the runtime of an algorithm is not a programming but an analytical activity, which means that we apply mat to figure out the runtime of an algorithm instead of writing

a program. Still we can support this activity by transforming programs that work on values representing the problem domain so that they count the number of steps they take. This doesn't mean that we can compute the runtime of programs, it only means that we can compute sample values that provide hints about a program's complexity.

To illustrate this idea, let's consider again the definition of the function `smallest` (see also page 15).

```
smallest :: [Float] -> Float
smallest [x]    = x
smallest (x:xs) = min x (smallest xs)
```

There are two possibilities to change this definition so that it counts the steps that the function performs. First, we could try to add the computation of steps to the computation of values. This means that the return type of the function becomes a pair of the result and the number of steps. If we represent the number of steps with integer values, this approach leads to the following type and base case:

```
smallestS :: [Float] -> (Float,Int)  -- unnecessarily complicated
smallestS [x] = (x,1)
...
```

We call the function `smallestS` to indicate that it (also) computes the number of steps. While this works well so far, the definition for the inductive case is complicated by the fact that the recursive call to `smallestS` now produces a pair of values, which have to be extracted from the pair to facilitate the two computations of values and steps to proceed.

A simpler approach is to ignore the computation of the original values altogether and only compute the number of steps, which is appropriate because we already have the function `smallest` to compute the values. This means that the result type of `smallestS` should be simply `Int`. But what about the type of the input? While `smallest` works on lists, the goal of `smallestS` is to report the number of steps the algorithm takes dependent on the *size* of the input, that is, the length of the list. Therefore, the input to `smallestS` should be also an integer representing the length of the input list. To express the purpose of `smallestS` more clearly in its type, we use the following two type definitions.

Chapter2.hs

```
type Size  = Int
type Steps = Int
```

The definition of the function `smallestS` must now compute the number of steps the function `smallest` takes for a list of a given length. We have already seen that the smallest element of a list containing just one element can be computed in one step, since we can simply return the list element as a result. For a list of n elements the definition of `smallest` first recursively computes the smallest element of the tail of the list, which contains $n - 1$ elements, and then determines the smaller of that element and the first element of the list. The number of steps required by the recursive computation can be computed, also recursively, by `smallestS`, since this is a function that determines the number of steps `smallest` takes for a length of some size. The computation of the minimum takes one step. Altogether, this leads to the following definition.

```
smallestS :: Size -> Steps
smallestS 1 = 1
smallestS n = 1 + smallestS (n-1)
```

With `smallestS` we can now compute the number of steps for different lengths as follows.

```
> smallestS 7
7
> smallestS 99
99
```

Since the runtime of an algorithm is a function relating input size to number of steps, applying `smallestS` to individual values is not very useful. To get a more comprehensive picture of the runtime, we should apply the function to large range of values. We can use the `map` function to do this together with a special Haskell syntax that allows us to construct a list of values by giving a range.

```
> map smallestS [1..25]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25]
```

This indicates that `smallestS` behaves like the identity function and that the number of steps that `smallest` needs is proportional to the length of the input list. This behavior is called *linear*, and `smallest` is said to have *linear runtime*. We sometimes also say that `smallest` is a *linear algorithm*. Of course, looking at a limited number of examples is not a proof that the runtime of `smallest` is linear; it only suggests this fact.

Note that ignoring the values of lists and not distinguishing between lists of the same length, effectively considering them the same by representing them by their length, is a form of *abstraction*. This means that functions that share the same structure have the same runtime (if the individual steps have the same runtime). Therefore, if we consider a function `largest` that can be obtained from `smallest` by replacing `min` with `max`, we can see that it also has linear runtime, since the definition of a step counting function `largestS` would be identical to `smallestS`.

► *Abstraction* Chapter 15

Exercise 15

Consider the following definition of the function `snoc` that add an integer at the end of an integer list:

```
snoc :: Int -> [Int] -> [Int]
snoc x []     = [x]
snoc x (y:ys) = y:snoc x ys
```

It consists of two cases: the base case says that adding an element to the empty list is a list of just the element to be added. Adding an element to a non-empty list works by adding the element to the tail of the list and putting the head of the list in front of the result. By recursively applying `snoc` to smaller and smaller lists, we eventually reach the base case of the empty list.
Define the function `snocS` to count the number of steps that `snoc` takes for adding an element to a list of n elements. Note that `snocS` should have the following type:

```
snocS :: Size -> Steps
```

The reason for ignoring the (size of the) first argument is that it does not affect the result. Note that adding an element at the front of a list with `:` takes one step.

Linear runtime is only one possible behavior. In the remainder of this section we look at a few other code examples and their runtime behavior.

2.2 Constant Runtime

An algorithm has *constant runtime* if it always requires a constant number of steps, irrespective of the input. In other words, the number of steps the algorithm takes does not depend on the size of the input. In such cases we also talk of a *constant-time* algorithm.

Examples are all the functions discussed in the introductory section, for example, `wakeUp` and `positive`. Also, the functions `moveRightBy`, `moveTo`, and `min` take only a constant number of steps. Whether an operation has constant runtime depends on the abilities of the computer to execute algorithms. For example, it is a fair assumption that today's PCs and laptops can multiply integers in constant time. However, that assumption is generally not true for humans: the larger the two numbers, the longer it usually takes to multiply them.

In the case of Haskell, we can assume for simplicity in the following that any function that does not employ recursion (either directly or indirectly by calling other recursively defined functions) has constant runtime. Thus we consider functions that perform computations with numbers and also simple list functions, such as `head` and `tail`, to run in constant time. Moreover, any (non-recursive) combination of constant-time functions results in a constant-time function.

2.3 Linear Runtime

An algorithm has *linear runtime* if the number of steps it takes grows proportionally with the input. For example, an algorithm will take roughly twice as long to process input that is twice as large. Many algorithms on lists have linear runtime (or worse). As one example we have already seen the function `smallest`. As a more complex example, let's consider the definition of `closestTo` (see also page 16).

```
closestTo :: Pos -> [Pos] -> Pos
closestTo c ps = let d = smallest [distance c p | p <- ps]
                 in head [p | p <- ps, distance c p == d]
```

We observe that the computation happens in three phases: first, a list comprehension maps the argument list of positions into a list of `Float` values that represent the distance from the parameter `c`. This takes as many steps as the list contains elements, that is, linear time with respect to the size of the list (assuming that the function `distance` is a constant-time function). Then the function `smallest` is applied to this list. As we have seen, this also takes linear time. Finally, another list comprehension filters out of the argument list all those positions whose distance is equal to the smallest distance, which again takes linear time. The application of the constant-time function `head` does not change the linear-time behavior.

We can capture this analysis in the definition of a function `closestToS` that works similar to `smallestS`. We ignore the first argument, since it doesn't contribute to the runtime and otherwise replace expressions with expressions that reflect their runtime. For example, we replace both list comprehensions by `n`, since `n` is the size of the input list and the list comprehensions are linear in that size, and the application of `head` is replaced by adding one step. We also pass `n` as an argument to `smallestS` to count the steps for this part of the computation, and we have to add the number of steps `s` to the final result to make sure that all the steps in the `let` expression are properly counted.[22]

```
closestToS :: Size -> Steps
closestToS n = let s = smallestS n + n
               in 1 + n + s
```

[22] Note that `smallestS n + n = (smallestS n) + n` because function application has a higher precedence than binary operations.

We can now compute the result of `closestToS` yields for different input sizes.

```
> map closestToS [1..25]
[4,7,10,13,16,19,22,25,28,31,34,37,40,43,46,49,52,55,58,61,64,67,70,73,76]
```

Here the relationship is not as obvious as in the case for `smallestS`. We could create a graphical plot of the values, which would show that the relationship between input and output is indeed linear.

Instead of graphing the result or otherwise speculating about the exact relationship, we can also simplify the function definition, which will reveal the linear relationship.[23] First, since we have observed that `smallestS` is the identity function, we can substitute `smallestS n` by `n`. Second, we can substitute the variable `s` by its definition, which after the simplification of `smallestS n` has become `n + n`. We obtain the following intermediate result.

```
closestToS :: Size -> Steps
closestToS n = let s = n + n
               in 1 + n + (n + n)
```

Finally, we can remove the `let` expression, since the variable `s` is not used in the result, and we can simplify the addition of the three `n`s.

```
closestToS :: Size -> Steps
closestToS n = 3*n + 1
```

This definition clearly reveals the linear relationship between the input size and number of steps.

Exercise 16

Consider the following definition of the function `sndSmallest` that works by computing the smallest element of a list from which the smallest element has been removed. (Note that the function definition works only for list of two or more elements.)

```
sndSmallest :: [Float] -> Float
sndSmallest xs = smallest (xs \\ [smallest xs])
```

Define the function `sndSmallestS` for counting the number of steps that `sndSmallest` takes for a list of n elements and simplify it so that it reveals the functional relationship between input size and number of steps.

2.4 Quadratic Runtime

If finding the next pebble is a constant-time operation, it takes Hansel and Gretel linear time in the number of pebbles to find their way back home. The assumption that the next pebble can always be found in constant time may or may not be warranted—it all depends on how densely Hansel has dropped the pebbles and thus how easy it is for Hansel and Gretel to find the next pebble in each step.

The different version of the `findHome` function do not make any assumption about the order of the pebble locations in their list argument. Therefore, the closest pebble always has to be identified with the function `closestTo`. Since `closestTo` is a linear algorithm that is executed n

[23] Here we make use of the nice property of functional programming that we can always substitute equals for equals.

times for a list containing n pebbles, we suspect that `findHome` is a quadratic algorithm.[24] In most cases this level of analysis is sufficient to understand the runtime behavior of a function. But we can also derive a function for counting the steps as we did for other functions. Concretely, let's take the final definition of the function `findHome`, developed as a solution for exercise 14.

```
findHome :: Pos -> [Pos] -> [Pos]
findHome c ps | c==home   = [home]
              | otherwise = let qs=ps \\ [c]
                            in c:findHome (closestTo c qs) qs
```

In developing the corresponding step-counting function `findHomeS` we again first ignore the `Pos` argument (since it doesn't affect the runtime), change the argument list to a `Size` value, and the result type to `Steps`. Without the first `Pos` argument, we have to change the condition that distinguishes the base case and the inductive case. Specifically, we have to base the condition on the newly introduced `Size` value. For this application, it is reasonable to assume that the position that equals `home` will be found last, which means it will be found if the argument list is of size 1. In that case, the resulting list of `findHome` can be constructed in one step. Otherwise, we have to remove an element from a list, which takes linear time (`n`), compute `closestTo` for a list of $n-1$ elements, which takes time `closestToS (n-1)`, then recursively compute `findHome` on the same list of $n-1$ elements, which takes time `findHomeS (n-1)`, and finally add `c` at the front of the list for one additional step. Altogether this leads to the following definition:

```
findHomeS :: Size -> Steps
findHomeS 1 = 1
findHomeS n = n + closestToS (n-1) + findHomeS (n-1) + 1
```

Computing example values confirms that the runtime of the function is not linear, but the exact relationship is not easy to see.

```
> map findHomeS [1..20]
[1,8,19,34,53,76,103,134,169,208,251,298,349,404,463,526,593,664,739,818]
```

It turns out, however, that the recursive definition of `findHomeS` is equivalent to the following definition:

```
findHomeS :: Size -> Steps
findHomeS n = 2*n^2 + n - 2
```

This confirms that the `findHome` is a quadratic algorithm.

More information on runtime (quadratic and others), including exercises, follows in sections 5, ??, and ??.

3 The Mystery of Signs

Whenever an algorithm is executed, the resulting computation solves a problem if the representation is chosen properly, that is, if the symbols that are manipulated by an algorithm stand for something meaningful in the world. This means a computation becomes meaningful only if the

[24] Even though `closestTo` is applied to shorter and shorter lists, it is on average applied to a list of length $n/2$, and $n/2 \times n$ is still quadratic. More precisely, `closestTo` is applied to lists of lengths n, $n-1$, ..., 1, which means the total number of steps is proportional to the sum of the first n numbers, which is quadratic in n.

results are interpreted by someone. It therefore follows that the same computation can have different meanings under different interpretations and that the question of whether the computation solves a problem is dependent on such an interpretation.

This aspect can be nicely demonstrated by different number systems. In everyday life we are used to working with *decimal numbers*, which are sequences of one or more digits in the range 0 to 9. In contrast, a *binary number* is a sequence of digits that are all either 0 or 1. In Haskell the type `Int` consists of decimal numbers. We can also create our own representation by using lists to reveal the structure of numbers. Specifically, we can define the following type for positive decimal numbers:

Chapter3.hs

```
type Digit   = Int
type Decimal = [Digit]
```

The idea is that each digit of a number is represented by a separate list element. With this type we could represent the number 237, for example, by the list `[2,3,7]`, which preserves the order of digits. This representation is not without problems, because the type `Dec` contains values, such as `[5,31]` or `[-3,-5]`, that do not follow the rules of using only single digits and thus do not represent a decimal number according to the intended representation. We can, of course, implement functions to check whether a `Dec` value represents a proper decimal number and use those functions to ensure the correctness of the representation wherever needed.

Chapter3S.hs

Exercise 17

Define the following two functions that test whether an integer is a digit and whether a `Decimal` value is a proper representation of a decimal number. (See also exercises 4 and 5.)

```
isDigit   :: Digit   -> Bool
isDecimal :: Decimal -> Bool
```

Note that the definition of the function `isDecimal` should contain *two* base cases: one for the empty list and one for a list of one digit.

Capturing a representation in Haskell means to identify *two* types: one type `T` of values that are to be represented, and one type `R` of values that do the representing. In other words, `R` values represent `T` values. In the terminology of semiotics the represented type `T` contains the signifieds, and the representing type `R` contains the signifiers. For the decimal number representation, `R = Decimal` and `T = Int`, that is, `Decimal` is the type for representing `Int`. For example, the value `[2,3,7]` is a signifier for the signified 237.

One way to capture the relationship between the two types of a representation is to define by two functions `rFromT` and `rToT`, which have the following types.

```
rFromT :: T -> R
rToT   :: R -> T
```

These two functions should satisfy the following equation for all values `t` of type `T`:

```
rToT (rFromT t) = t
```

This means that converting a value to its representation and then converting it back should yield the original value.[25] We may also want to have a similar relationship for `R` values. However,

[25] In Haskell we can form the composition of two functions using the operator "`.`" so that we can express this requirement also concisely as an equation between functions: `rToT . rFromT = id` where `id` is the identity function.

as we have seen, the representation type is sometimes "too big" and contains values that do not represent any `T` value. We could restrict a corresponding equation to only valid `R` values.

```
validR r  =>  rFromT (rToT r) = r
```

Alternatively, we could rely on the implementation of the `fromT` function to only produce valid representations and write instead:

```
rFromT (rToT (rFromT t)) = rFromT t
```

To illustrate these ideas, let us define the two functions for the decimal representation of integers. First, here is the function definition for converting a positive integer to a `Decimal` value. The base case applies to integer values that are single digits and results in a list of that integer. We can use the function `isDigit` defined in exercise 17 to implement this test. The inductive case splits the integer argument by dividing it by 10 into a front part and last digit, extracting the last digit using the modulus function `mod` and the number without its last digit using integer division `div`. For example, `div 237 10 = 23` and `mod 237 10 = 7`. (Note that the backquotes allow us to write any binary operation as an infix operator, that is, we can write `div 237 10` as `237 ‘div‘ 10`.) The front part can be transformed into a list by recursively calling `fromInt`, and the last digit is then appended to this list.

```
decFromInt :: Int -> Decimal
decFromInt i | isDigit i = [i]
             | i>9       = decFromInt (i ‘div‘ 10) ++ [i ‘mod‘ 10]
```

This function behaves as intended.

```
> decFromInt 237
[2,3,7]
```

The idea for conversion in the other direction, from `Decimal` back to `Int`, is to multiply each digit with a power of 10, corresponding to its position in the list and then add all these numbers. For example, `[2,3,7]` should result in the computation of $10^2 \times 2 + 10^1 \times 3 + 10^0 \times 7 = 200 + 30 + 7$. In the definition of the function `decToInt` the base case for a single-digit list needs no computation at all. For longer lists, the exponent is determined by number of remaining digits, which is the same as the length of the tail of the digit list.

```
decToInt :: Decimal -> Int
decToInt [d]    = d
decToInt (d:ds) = 10^(length ds)*d + decToInt ds
```

We can now test the two functions on a range of examples using the above-mentioned equation that requires the two functions be inverses of each other.

```
> map (decToInt.decFromInt) [0,1,9,13,99,100,101,9999]
[0,1,9,13,99,100,101,9999]
```

Exercise 18

Following the example of `Decimal`, define the types `Bit` and `Binary` for representing integers using list of `0`s and `1`s. Then define functions to test whether an integer is a bit and whether a `Binary` value is a proper representation of a binary number.

```
isBit    :: Bit -> Bool
isBinary :: Binary -> Bool
```

Then define the following representation functions.

```
binFromInt :: Int -> Binary
binToInt   :: Binary -> Int
```

Note that the definition of these two functions can be mechanically obtained by exchanging the base 10 of the decimal representation by a base 2 for the binary representation.

What have we achieved by defining a (decimal or binary) number representation? We can now implement operations on numbers based on that representation, that is, operation on the type of represented values `T` can be realized using values of the `T` representation. To illustrate, consider the case of multiplying a decimal number n by 10. We all know that this is achieved by appending a `0` at the end of n's digits. Adding a `0` at the end of a list of numbers can be done using the following function `addZero`.[26]

```
addZero :: Decimal -> Decimal
addZero is = is ++ [0]
```

We now expect that applying `addZero` to a decimal representation results in a representation of an integer 10 times as big. In other words, given an integer `n`, obtaining its decimal representation using `fromInt`, applying `addZero` to it, and turning it back into an integer with `toInt` should result in the integer `10*n`. We can test this proposition using a number of examples.[27]

```
> map (decToInt.addZero.decFromInt) [0,1,9,13,99,100,101,9999]
[0,10,90,130,990,1000,1010,99990]
```

Similarly, adding a zero to the binary representation means to multiply the represented number by 2, a fact easily confirmed by using the binary representation functions.

```
> map (binToInt.addZero.binFromInt) [0,1,9,13,99,100,101,9999]
[0,2,18,26,198,200,202,19998]
```

These examples demonstrate how we can compute with numbers by manipulating the values of the representing type. (The same principles apply to the geometric computation of $\sqrt{2}$ that is explained in the Introduction of the book.)

[26] Actually, we could also simply use the expression `(++ [0])`, which fixes the second argument of the `++` function and turns it into a function to add the list `[0]` to any list of numbers it is applied to.

[27] Note that an application of a composed function `(f.g.h) x` is the same as applying the functions individually as in `(f (g (h x))`. Function composition allows us to pass the function directly as an argument to `map`. Without it we had to first define an auxiliary function.

We can now implement all kinds of arithmetic functions on decimal and binary representations. Here is one simple example as an exercise.

Exercise 19

Define a function for division by 10 (without rest) for the decimal representation.

```
div10 :: Decimal -> Decimal
```

The function should behave as follows.

```
> map (decToInt.div10.decFromInt) [0,1,9,13,99,100,101,9999]
[0,0,0,1,9,10,10,999]
```

4 Detective's Notebook: Accessory after the Fact

This section explores the relationship between data structures and data types. We have already seen how lists provide a concrete representation for maintaining collections of elements. Lists are an example of a *data structure*. In section 4.1 I discuss more examples of lists and list functions, which allows the comparison of lists to two other important data structures, *arrays* and *trees*, covered in sections 4.2 and 4.3, respectively.

Different data structures represent information differently and therefore have different strengths and weakness with respect to different computation tasks. Such tasks are defined by an application that employs data structures to represent the information needed by the application. Specific requirements for data transformations are captured in a *data type*. A data type (at minimum) lists the operations together with their types. Only few languages allow the specification of the properties of the operations, but like most other programming languages, Haskell doesn't support this aspect of data types. In Haskell a data type is specified using a so-called *type class*. The name "type class" suggests that a type class usually stands for a collection of types, namely for all types that implement the operations of that class.[28]

We will explore the set, stack, and queue data types in sections 4.4 through 4.7. Specifically, I will illustrate in section 4.4 how we can implement one data type with different data structures.

4.1 More on Lists

We have already used lists extensively. The standard lists provided by Haskell are so-called *singly-linked lists*, in which each element has a pointer to the next element in the list,[29] and a list is identified by a pointer to its first element. We have seen that we can build lists by adding elements at the beginning of a list using the cons operator `:` or by listing all elements enclosed in square brackets and separated by commas. Here is a list of three suspects from the Sherlock Holmes adventure *The Hound of the Baskervilles*.

Chapter4.hs

```
suspects :: [String]
suspects = ["Mortimer","Beryl","Selden"]
```

The Haskell list type as well as the type `Bool` are two examples of types that can be introduced by a `data` definition.[30] A data definition introduces constructors that have a unique name and

[28] In other languages, the concept of a type class is often called an *interface*.

[29] Doubly-linked lists also have pointers to preceding elements and can be traversed in both directions.

[30] In Haskell, these types are actually called *data types*, which should not be confused with the concept of data type used in the book and in this document, which is often called *abstract data type* to distinguish the two.

that may carry additional information. One of the simplest data definitions is the type for representing boolean values.

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

It consists of two constructors that don't carry any additional information. The type `Bool` has just two values, `False` and `True`. Constructors that carry additional information are used to build data structures. The list type is a good example. The Haskell built-in lists offers a special syntax and are also polymorphic, which means that we can create and manipulate lists of integers, people, pairs, and so on Ignoring these two aspects, we can define a simple type of integer lists as follows.

```
data List = Empty | Cons Int List
```

The type `List` has two constructors. The first constructor, `Empty`, represents an empty list. Just as `True`, this is a constructor that is a constant value that stands for itself and carries no further information. The second constructor, `Cons`, has two arguments, an integer `x` and a list `xs`, and represents a list whose first element is `x` and whose tail is `xs`. The list `[1,2,3]` (which is just a more convenient way of writing `1:2:3:[]`) can be represented as a value of type `List` as follows:

```
Cons 2 (Cons 2 (Cons 3 Empty))
```

We could also define a list type for storing strings by simply changing the first argument type of the `Cons` constructor.

```
data List = Empty | Cons String List
```

Unfortunately, since function, type, and constructor names must be unambiguous, we cannot use both definitions at the same time. By parameterizing the data definition we can make the list data structure *polymorphic*, that is, we can have one type of lists that can store integers or strings or any other type of information. The list type name is extended by a type parameter `a`, which is substituted by the type of elements when a list is built.

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

With this definition of lists, we can represent integer lists as well as string lists or any other lists. But note that within one list, all elements must be of the same type type. The purpose of the previous example is to illustrate how data structures can be defined using Haskell's `data` definitions. In the following we continue to use the built-in lists of Haskell.

The elements of the list `suspects` are so-called *strings*, which are sequences of characters enclosed in double quotes. String values stand for themselves, unlike names, which are sequences of characters without quotes that are defined in Haskell programs and denote values, functions, or types. Using strings is a good idea whenever the names or text to be manipulated is not clear in advance. In this case, strings provide the flexibility that is needed to form arbitrary sequences of characters. On the other hand, using strings prevents the use of some data structures. In particular, strings cannot be used as indexes for an array. Since I want to illustrate in section 4.4 two implementations of the set data type by lists and arrays, I introduce the names of suspects here using a an enumeration type.

```
data People = Mortimer | Desmond | Jack | Beryl | Selden
              deriving (Eq,Ord,Show,Ix)
```

The shown form of a `data` definition is similar to the type `Bool`. The `deriving` clause makes sure these values can be compared for equality (`Eq`), can be ordered, that is, compared with `<` (`Ord`), printed (`Show`), and used as index values for an array (`Ix`).

With these newly introduced constructors, we can define a list of suspects as a list of `People` values as follows.[31]

```
suspectsL :: [People]
suspectsL = [Mortimer,Beryl,Selden]
```

Note that we can only use names that have been declared as constructors of the `People` type, whereas with strings we can use any name we like.

A consequence of the singly-linked list representation is that a list is always accessed via its first element and has to be traversed element by element, front to back, which causes many list operations to have linear runtime. Consider, for example, the task to find an element in a list, implemented by the following function `find`.

```
find :: People -> [People] -> Bool
find _ []                 = False
find y (x:xs) | x==y      = True
              | otherwise = find y xs
```

We can use the `find` function to determine if a specific name is a suspect.

```
> find Selden suspectsL
True
> find Jack suspectsL
False
```

The runtime of `find` depends on the position of the sought element in the list. In the worst case, when the element appears at the end of the list or isn't contained in the list at all, the list has to be traversed completely. Similarly, looking up an element that is stored at a particular position in the list takes time proportional to the position and thus is also linear in the worst case. A widely used convention is that list positions are numbered starting with 0.[32] Since this is rather counter-intuitive, I start numbering lists element with 1.

```
index :: Int -> [String] -> String
index 1 (x:xs) = x
index n (x:xs) = index (n-1) xs
```

In the book I have compared a list to a ring binder into which pages can be inserted and from which pages can be removed. Correspondingly, we can define functions to insert and remove elements into/from a list. Inserting at the beginning of a list can easily be done using the cons operator `:` that we have used before. Inserting an element at a particular position requires traversing the list to find the position.

```
insertAt :: String -> Int -> [String] -> [String]
insertAt y 1 xs     = y:xs
insertAt y n (x:xs) = x:insertAt y (n-1) xs
```

[31] The suffix L after the name indicates that this is the list representation of suspects.

[32] Haskell is no exception. It has a predefined infix operation `!!` that works like `index` but starts numbering lists elements at 0.

If Sherlock Holmes wants to add Jack as the second member to his list (or ring binder) of suspects, he can do this as follows.

```
insert Jack 2 suspects
["Mortimer","Jack","Beryl","Selden"]
```

Chapter4S.hs

Exercise 20

Define a function for removing an element from a specific position in a list.

```
removeAt :: Int -> [People] -> [People]
```

4.2 Arrays

Unlike lists, arrays have a fixed size and thus cannot grow or shrink, that is, we cannot define functions `insert` and `remove` for arrays. An array is more like a notebook that has a fixed number of pages. Although the information on each page can be changed, no new pages can be added, and it is usually assumed that no pages are torn out. Therefore, it does not make much sense to define a three-element array with the names who are currently suspects if the suspects are expected to change, since we could not update the array to reflect such a change. Instead, we can define an array that contains space for each potential suspect, that is, for each element of the type `People` and store with it information on whether the person is currently a suspect.

In the following definition, the type says that `suspectA` is an array that is indexed by values of type `People` and that stores in each cell a boolean value. The first argument of the `array` function that builds a new array defines the size of the array and the range of its index values. In this case, the size is 5, and the range includes all element from the type `People`, from the first to the last mentioned constructor.[33] The second argument is a list of (index, value) pairs that define the content of the array. Note that the `True` and `False` values have been chose so that this array represents the same people as the list `suspectL`.

```
suspectsA :: Array People Bool
suspectsA = array (Mortimer,Selden)
   [(Mortimer,True),(Desmond,False),(Jack,False),(Beryl,True),(Selden,True)]
```

We can "find" people in the array by directly looking up the boolean value stored in the cell indexed by the corresponding name, which is done using the infix operation `!`.

```
> suspectsA ! Selden
True
> suspectsA ! Jack
False
```

Adding Jack as a suspect now requires changing the value of its array cell to `True`. This is done using the infix operation `//`, which takes a list of (index, value) pairs and updates all mentioned cells accordingly.

[33] To construct a three-element array with indexes spanning from `Desmond` to `Beryl`, you would use the range `(Desmond,Beryl)`. Note that ranges taken from the index type must be consecutive, thus it is not possible to construct, say, a two-cell array indexed by `Jack` and `Selden`.

Referential Transparency

Referential transparency requires that the definition of variable cannot be changed. For example, a list on which operations are performed still keeps its original value.

```
> let xs = [1,2,3]
> 99:tail xs
[99,2,3]
> xs
[1,2,3]
```

Consequently, if we update a data structure, Haskell has to construct a representation that leaves the old value unchanged if it is bound to a variable such as `xs` in the example.

For arrays this creates a problem, since no array implementation can update an array cell in constant time and at the same time leave an old version of the array with constant time access to its elements. This means that we either have to give up referential transparency to achieve constant-time array access and updates, or use a less efficient array implementation that does support referential transparency. Haskell has chosen the latter path.

```
> suspectsA // [(Jack,True)]
array (Mortimer,Selden) [(Mortimer,True),(Desmond,False),(Jack,True),
(Beryl,True),(Selden,True)]
```

Note that the position of people in the array is fixed when an array is built. Specifically, the ordering is given by the type definition of the index value, here `People`.

Although Haskell has ample support for arrays, dealing with arrays is complicated by Haskell's steadfast commitment to *referential transparency* (see also box Referential Transparency), which means that we can always substitute a variable in an expression by its definition without changing the meaning of the expression. However, this property cannot be supported by array implementations that provide constant-time access and update of individual array cells.[34] Thus, you should be aware that some of the array update operations that are mentioned in the following do not have the theoretically possible optimal runtime.

Exercise 21

Define a function for removing a person as a suspect in the array representation of suspects.

```
clear :: People -> Array People Bool -> Array People Bool
```

4.3 Trees

Trees come in many different shapes and forms. In general, a tree consists of nodes that are hierarchically arranged. Specifically, a node can have zero or more child nodes, and all but one node have exactly one parent. The node without a parent is called the tree's *root*, and it provides the entry point to the elements in a tree, just like the first element in a list. A node without any children is called a *leaf* of the tree. The following data definition captures this idea. For simplicity, I have fixed the type of elements to be strings. As in the example of lists, we could generalize this definition by using a type parameter if it were needed. Also, since we don't plan

[34] It actually can be achieved by using monads, but these are beyond the scope of this document.

on using the strings stored in a tree as index values for an array, we don't need a `data` definition and can stick with strings instead.

```
data Tree = Node String [Tree]
          deriving Show
```

This definition looks peculiar, since it is on the one hand recursive and on the other hand, unlike the list type, doesn't have a base case. But after considering a few examples it becomes clear that the base case for the employed list type provides the base case for trees as well. For instance, here is a tree that consists of only one node that has no children.

```
Node "Hugo" []
```

The node is root and leaf at the same time. And here is a tree that has one root and two children.

```
Node "Hugo" [Node "Charles" [], Node "John" []]
```

Since it is inconvenient having to write the empty list for each leaf in a tree, one might want to introduce a function for constructing leaves that does that automatically and thus makes expressions for denoting trees easier to write and read.

```
leaf :: String -> Tree
leaf s = Node s []
```

Using this auxiliary function `leaf` the Baskerville family tree can be represented as follows.

```
baskervilles :: Tree
baskervilles = Node "Hugo" [leaf "Charles",
                            Node "John" [leaf "Henry"],
                            Node "Rodger" [leaf "Stapleton"],
                            leaf "Elizabeth"]
```

Exercise 22

Define a function `single` that takes two strings `p` and `c` and constructs a tree in which `p` is the parent of `c`. (Note: You may want to reuse the function `leaf` in your solution.)

```
single :: String -> String -> Tree
```

Give a definition of the `Tree` value `baskervilles` that uses the function `single`.

The inheritance order of a family tree is the list of names obtained by traversing the tree from top to bottom and left to right. Specifically, to compute the inheritance list for a node with parent `p` and a list of children `cs`, we want to (a) compute the inheritance lists for all the children in `cs` from left to right, (b) concatenate them into one list, and (c) add `p` at the beginning of this list. Step (a) can be accomplished by a list comprehension that applies the function `inheritance` to all elements `c` of `cs`. For step (b) we employ the function `concat` that takes a list of lists and concatenates them all into one list, and step (c) means to cons `p` in front of the result of (b). This leads to the following implementation.

```
inheritance :: Tree -> [String]
inheritance (Node p cs) = p:concat [inheritance c | c <- cs]
```

With `inheritance` we can compute the inheritance list for the Bskerville family as follows.

```
> inheritance baskervilles
["Hugo","Charles","John","Henry","Rodger","Stapleton","Elizabeth"]
```

4.4 The Set Data Type

A very basic data type is that of a *set*. A set needs operations to create an empty set, add and remove elements from a set, and an operation to test whether a value is an element of a set. A description of the set data type as a Haskell type class looks as follows:

```
class PeopleSet s where
  empty   :: s
  isEmpty :: s -> Bool
  member  :: People -> s -> Bool
  insert  :: People -> s -> s
  remove  :: People -> s -> s
```

This first line of this definition introduces the name of the data type, `PeopleSet`, and a variable, `s`, that stands for the data structures that implement the data type. Each of the following lines then presents the name of an operation plus its type to create, manipulate, and inspect the data type. For example, `empty` is a value that represents an empty set, and `member` tests whether a given value of type `People` is contained in a set of type `s`.

This set data type is specialized to representing only sets of `People`, but of course sets can be used to represent many other kinds of data. Instead of defining a separate data type for each new element type, we could generalize the definition by using another type variable instead of the type `People` in the definition to make it more general. However, this would raise some technical issues that would distract from the major point of this section, and having a set for representing sets of `People` values is sufficient to illustrate the major points about data types and their relationship to data structures.

A data type also has a set of equations that specify the required behavior of the operations. For example, we would expect the set operations to have the following properties.[35]

```
isEmpty empty         == True
isEmpty (insert p s)  == False
member p (insert p s) == True
member p (remove p s) == False
remove p (insert p s) == s
...
```

However, only few languages support the specification and enforcement of such properties. In Haskell, one can express such equations as functions that can then be used for automatic testing by tools such as QuickCheck.

The important aspect of a data type definition is that it does *not* prescribe any particular implementation. In fact, most data types can be implemented in a variety of ways. The implementation of a data type requires a concrete data structure to represent values of the data type and an implementation for all operations of the data type.

In Haskell the implementation of a data type through a data structure is accomplished by using a so-called *instance declaration*. Just as the name "type class" indicates that a data type can be implemented by many types, the name "instance declaration" indicates that a data structure that implements the data type is one of these types.

[35] The variables `p` and `s` stand for arbitrary people and set values, respectively.

4.4.1 A List Implementation of Sets

In the following definition, the first line says that the type `[People]` is the data structure hat implements the data type `PeopleSet`. The following lines give the definitions for the functions based on this list representation: an empty set is represented by an empty list, and a set is empty when its representation is the empty list. The `member` function requires behavior that is realized by the function `find` that we have defined already, and thus we can use it to implement `member`. Inserting an element into a set can be done by simply adding it at the front of the list. With this implementation it might happen that one element is represented multiple times in a list. This is the reason why the implementation of the function `remove` must continue to remove elements from the list even if it has found and removed the element (second to last line). This illustrates that assumptions about the representation (here that a list might contain multiple copies of a set element) affects how the functions need to be implemented.

```
instance PeopleSet [People] where
  empty = []
  isEmpty ps = ps==[]
  member p ps = find p ps
  insert p ps = p:ps
  remove q []                 = []
  remove q (p:ps) | p==q      = remove q ps
                  | otherwise = p:remove q ps
```

Since we have provided an implementation for the data type, we can now use the data type operations to build and manipulate sets. However, since the operations are elements of a type *class*, they can have multiple different implementations. In fact, we will provide another implementation based on an array representation shortly. Thus, since one name can refer to different implementations, we have to indicate which implementation we want to use. This can be done by adding a type annotation of the form "`:: T`" to an expression. Since we want to select the implementation based on lists, the annotation must be "`:: [People]`".

```
> insert Beryl (insert Jack empty) :: [People]
[Beryl,Jack]
```

Without such an annotation Haskell will report an error because it is not clear, which implementation it should pick.

The shown implementation allows multiple copies of the same element in a list representing a set, as can be seen in the following example.

```
> insert Jack (insert Jack empty) :: [People]
[Jack,Jack]
```

This is not a problem as far as correctness is concerned, since the only queries on sets, `isEmpty` and `member`, are not affected by it and the implementation of `remove` ensures to remove all copies. The only problem with this approach is that the list representation may become too large. The alternative, to store each element at most once, is easy to implement and requires only a change

of the definitions for `insert` and `remove`.

Exercise 23

Give definitions for functions `insert2` and `remove2` that maintain the invariant that set elements are represented only once in the list representation and otherwise work like `insert` and `remove`.

```
insert2 :: People -> [People] -> [People]
remove2 :: People -> [People] -> [People]
```

Specifically, the `insert2` should show the following behavior.

```
> insert2 Jack (insert2 Jack empty)
[Jack]
```

4.4.2 An Array Implementation of Sets

The array implementation of sets first specifies the type for representing sets, namely `Array People Bool`, an array whose index values are the elements of the `People` type and that stores boolean values. The following five lines implement the set operations based on the array representation: an empty set is given by an array that has a cell for each element of its index type, `People`, which is specified by giving the range `(Mortimer,Selden)` that spans all elements from the first to the last. Each cell is defined to be `False`, which means that none of the index values is an element of the set, exactly what is required of an empty set. The list of (index, value) pairs is given by a list comprehension (see page 15). The definition of `isEmpty` is similar to the list case, `member` works by looking up the boolean in the element's cell, and `insert` and `remove` change an element's cell to `True` and `False`, respectively.

```
instance PeopleSet (Array People Bool) where
  empty = array (Mortimer,Selden) [(p,False) | p <- range (Mortimer,Selden)]
  isEmpty s = s==empty
  member x s = s ! x
  insert x s = s // [(x,True)]
  remove x s = s // [(x,False)]
```

We can observe how this implementation works by executing the same examples as for the list representation.

```
> insert Beryl (insert Jack empty) :: Array People Bool
array (Mortimer,Selden) [(Mortimer,False),(Desmond,False),(Jack,True),
(Beryl,True),(Selden,False)]

> insert Jack (insert Jack empty) :: Array People Bool
array (Mortimer,Selden) [(Mortimer,False),(Desmond,False),(Jack,True),
(Beryl,False),(Selden,False)]
```

The array representation is much harder to read than the list representation, since it shows all potential set elements. One can easily define a function to produce a list representation from an array representation.

```
asList :: Array People Bool -> [People]
asList a = [p | (p,b) <- assocs a, b]
```

This definition again uses a list comprehension and employs the predefined Haskell function `assocs` that extracts from an array the list of all of its (index, value) pairs. Note that the condition to filter the returned list elements is the variable `b`, which is a boolean. Thus, only those pairs are kept whose boolean value is `True`. Also, using only `p` (instead of the pair `(p,b)`) to the left of the bar means to only include the `People` index values in the resulting list and not the boolean values.

Exercise 24

Define a function `asArray` that transforms a list representation of a set into a corresponding array representation of the same set. (Hint: Use a list comprehension and the `//` operation to update the empty set.)

```
asArray :: [People] -> Array People Bool
```

The function definition should, for example, satisfy the following condition.

```
> asArray suspectsL == suspectsA
True
```

Even though we can convert complete set representations from list to array and back, we cannot mix the different representation, which means that it is impossible to construct a set as a list and then modify it using the array operations or vice versa.

4.5 The Dictionary Data Type

The dictionary data type is an extension of the set data type that associates additional information with the elements stored (which are called *keys*). Dually, a set can be considered a dictionary whose entries have no information associate with them; the only information about them is whether or not they are contained in the set.

Whereas fixing the element type for the set data type is sufficient for the examples covered in the book, the dictionary data type is used with different key types. Since a formalization as a Haskell type class would require the introduction of too many type system concepts, I will not do that here. Instead, in section 5, I will present data structures directly to represent a dictionaries.

4.6 The Stack Data Type

A stack realizes the LIFO (last in, first out) behavior for collections, which means that the element inserted last, will be retrieved first. In Haskell we define the stack data type, like in the set example, as a type class. Since the stack data type behaves like a stack of plates in a cafeteria, the operation for putting a new element onto the stack is called `push`. The corresponding operation for removing an element is called `pop`. And since we can inspect only the topmost element, the operation to access that element is called `top`. In the following definition, we use the suffix `S` in `emptyS` and `isEmptyS` to distinguish the functions from those of the set data type.

```
class PeopleStack s where
  emptyS   :: s
  isEmptyS :: s -> Bool
  push     :: People -> s -> s
  top      :: s -> People
  pop      :: s -> s
```

The implementation of the stack data type with lists is straightforward.

```
instance PeopleStack [People] where
  emptyS = []
  isEmptyS ps = ps==[]
  push p ps  = p:ps
  top (p:ps) = p
  pop (p:ps) = ps
```

It is striking that `push`, `top`, and `pop` are essentially the basic list operations cons (`:`), `head`, and `tail`. We could emphasize this relationship by writing the implementation in a slightly different but equivalent way.

```
  push = (:)
  top = head
  pop = tail
```

We can use the stack to simulate the seating arrangement in an airplane. Suppose Jack, Beryl, and Selden sit in the same row so that Selden has a window seat, Beryl has the middle seat, and Jack has the aisle seat. While they have to take their seats in the order Selden, Beryl, Jack, when Selden needs to go to the bathroom, first Jack and then Beryl have to get off the row. Only then can Selden go.

```
> let row = push Jack (push Beryl (push Selden emptyS)) :: [People]
> row
[Jack,Beryl,Selden]
> top (pop (pop row))
Selden
```

4.7 The Queue Data Type

A queue is very similar to a stack, the only difference is that elements are added at one end and removed at the other. Queues realize the FIFO (first in, first out) behavior for collections. The operations are named differently from the stack operations to reflect the different behavior.

```
class PeopleQueue q where
  emptyQ   :: q
  isEmptyQ :: q -> Bool
  enqueue  :: People -> q -> q
  front    :: q -> People
  dequeue  :: q -> q
```

As with stacks, the implementation of the queue data type with lists is straightforward. In fact, the only operation that is different is the function `enqueue`, which adds element at the end of a list.

```
instance PeopleQueue [People] where
  emptyQ = []
  isEmptyQ ps = ps==[]
  enqueue p ps = ps ++ [p]
  front = head
  dequeue = tail
```

We can use a queue to simulate what happens when people stand in line. Suppose after their long flight Selden, Beryl, and Jack want to get a coffee. Now they will be served in the order in which they have entered the line, represented by a queue data structure.

```
> let queue = enqueue Jack (enqueue Beryl (enqueue Selden emptyQ)) :: [People]
> queue
[Selden,Beryl,Jack]
> front queue
Selden
> front (dequeue queue)
Beryl
> front (dequeue (dequeue queue))
Jack
```

Exercise 25

Suppose two queues have formed at the coffee shop. The barista wants to serve the people in the two queues by alternating between the two queues. This can be also achieved by merging two queues into one. Define a function `mergeQ` which does that. To avoid the need for dealing with type-class constraints in the type, define the function for the list implementation of queues.

```
mergeQ :: [People] -> [People] -> [People]
```

(**Hint**: You need two base cases to deal with the case that either queue is empty, and the inductive case applies when both queues are not empty and can contribute an element to the merged queue.)

5 The Search for the Perfect Data Structure

A general pattern for any search is the repeated reduction of the set of locations that can contain the sought item until it is found. The linear search through a list does this not very efficiently since in every step the set of locations is reduced only by one, the first element of the list, which leads to linear runtime for th search algorithm.

... binary search

Before Indiana Jones crosses the tile floor over an abyss, he has to decide which tiles to jump onto. Since the code word that indicates the safe tiles exists in different languages, it is not immediately obvious by which one he should be guided. He can improve his chances of hitting a safe tile by computing the frequency of the letters that occur in all the possible words and then pick letters that occur in most words.

A histogram maps elements that occur in some document or other data source to numbers that indicate their frequency. It is a data structure that is of direct use for Indiana Jones in this situation. A histogram can be represented by a dictionary (see section 4.5) that has the elements as keys and their frequencies as the information stored with the keys.

Section 5.1 demonstrates a simple implementation of the histogram using lists. A more efficient implementation using binary search trees follows in section 5.2. Finally, the trie data structure that underlies the tile floor is explained in section 5.3.

5.1 Histograms as Lists

A simple implementation represents a histogram as a list of (letter, counter) pairs. Single letters, or characters, are represented in Haskell by the type `Char`. We can use the following implementation of a histogram.

Chapter5.hs

```
type Histogram = [(Char,Int)]
```

To implement a function for increasing the count of a particular character in a histogram, we have to distinguish two cases. If the character is not contained in the histogram, we add the character with a count of 1. Otherwise, we increase the current count of the character by 1. To find the character to be updated in the histogram we traverse the list until we have found it (second equation) or until we have reached the end of the list (first equation).

```
update :: Histogram -> Char -> Histogram
update []        c = [(c,1)]
update ((k,n):h) c | k==c      = (k,n+1):h
                   | otherwise = (k,n):update h c
```

To extend a histogram by the count for all the letters in a word we need a function that applies the function `update` repeatedly to a histogram using each letter from that word. A word is represented as a string. In Haskell a string is the same as a list of characters, a fact we can verify by inquiring the definition of the type `String` using the `:i` command of the interpreter.

```
> :i String
type String = [Char]         -- Defined in GHC.Base
```

This means that the function to update a histogram by all the letters in a word can process the string as a list of characters. Like for so many list functions, the definition requires the consideration of two cases. When the string is empty, the histogram remains unchanged. When the string contains at least one character, we update its count in the histogram and then update the resulting histogram by the remaining characters of the word.

```
extendBy :: Histogram -> String -> Histogram
extendBy h []     = h
extendBy h (c:cs) = update h c `extendBy` cs
```

Note that I have used infix notation for the recursive function call of the `extendBy` function. The expression is equivalent to `extendBy (update h c) cs`. If we expand the application of `extendBy` to a string `"aba"`, we obtain the following expression.

```
  extendBy [] "aba"
= update [] 'a' `extendBy` "ba"
= update (update [] 'a') 'b' `extendBy` "a"
= update (update (update [] 'a') 'b') 'a' `extendBy` ""
= update (update (update [] 'a') 'b') 'a'
```

This recursion pattern of updating a value `x` repeatedly by a function `f` with the elements of a list `[x1,x2,...,xk]` is called *folding* and is captured in an operation `foldl`, where the `l` stands for "left" because the list elements are consumed from left to right. The behavior of `foldl` is captured by the following equation.

```
foldl f x [x1,x2,...,xk] = f ... (f (f x x1) x2) ... xk
```

By using `extendBy` for `f`, the histogram to be extended for `x`, and the word (that is, list of characters) for the list `[x1,x2,...,xk]`, we can define the `extendBy` function using the `foldl` operation as follows.

```
extendBy :: Histogram -> String -> Histogram
extendBy h s = foldl update h s
```

We can also observe that a function definition of the form `f x = e x`, which defines a function with a parameter `x` by applying an expression `e` to `x`, is identical to saying that `f` is the same as `e`. This means the definition can be simplified to `f = e`. This line of reasoning can be applied repeatedly, and a definition `f x y = e x y` can be abbreviated to `f = e`. Applying this idea to the definition of `extendBy` leads to the following, short definition:

```
extendBy :: Histogram -> String -> Histogram
extendBy = foldl update
```

Once one has some experience with how folding operations work, one can understand the latter definition much faster than the recursive one; it simply says that `extendBy` repeatedly updates a histogram with the letters taken from a word from left to right.

Indiana Jones can now use `extendBy` repeatedly to build a histogram of all the words that could be code words. In a first attempt, this can be done in a series of explicit definitions, naming intermediate histograms.

```
h1 = [] `extendBy` "God"
h2 = h1 `extendBy` "Iehova"
h3 = h2 `extendBy` "Jehova"
h4 = h3 `extendBy` (nub "Yahweh")
h5 = h4 `extendBy` "Yehova"
```

In computing `h4` the function `nub` gets rid of the second `h` because counting the `h` twice for one word would distort the letter count: it would lead to the incorrect conclusion that `h` occurs in all five words, which it doesn't. In general, the function `nub` removes all duplicate elements from a list.

Chapter5S.hs

Exercise 26

Define a function `remDup` that removes all duplicate characters from a string but leaves the characters and their order otherwise unchanged.

```
remDup :: String -> String
```

The function should behave as follows:

```
> remDup "Mississippi"
"Misp"
```

By inspecting the resulting histogram `h5` we can observe that letters `o`, `e`, `h`, and `a` all occur in four of the five words and are therefore most likely to identify safe tiles.

```
> h5
[('G',1),('o',4),('d',1),('I',1),('e',4),('h',4),('v',3),('a',4),('J',1),
('Y',2),('w',1)]
```

Instead of scanning the histogram manually, we can define a function that computes the letters with the highest frequency.

Exercise 27

Define a function `maxCount` that computes the list of all characters that have the highest count in a histogram. The function should also return the count.

```
maxCount :: Histogram -> (Int,String)
```

(**Hint**: Take another look at the function `closestTo` defined on page 16. The definition of `maxCount` can be defined using a very similar structure.)

With `maxCount` we can determine the safe tile characters automatically.

```
> maxCount h5
(4,"oeha")
```

Another aspect of the example that should be automated is the repeated call to `extendBy`, which will also eliminate the intermediate histograms. To this end, we can define a function `addWords` that repeatedly adds words to a histogram using the function `extendBy`.

```
addWords :: Histogram -> [String] -> Histogram
addWords h []     = h
addWords h (w:ws) = h `extendBy` (nub w) `addWords` ws
```

In the definition the function `nub` is systematically applied to every word that is being used to extend the histogram because we don't know whether a word contains duplicate characters. We can observe that the definition of `addWords` employs the same recursion schema as `extendBy`.

Exercise 28

Give an alternative definition for the function `addWords` using `foldl`.

With `addWords` we can compute the final histogram in one step.

```
h = addWords [] ["God","Iehova","Jehova","Yahweh","Yehova"]
```

And we can easily verify that the result is the same as before by comparing `h` with `h5`.

```
> h == h5
True
```

As mentioned, a histogram is essentially a dictionary data type with characters as keys and numbers as entries. An important operation on dictionaries is to find entries by keys. This functionality is actually part of the `update` function that traverses the list and compares the characters stored in it. Still, if we want to know about the frequency of a specific character, we do not have a function that can accomplish that. Such a function is not difficult to define, but it raises the issue of what to return when we are looking for a character that is not contained in the histogram. In this particular application, returning `0` for non-existing characters works well, but in general, we want to distinguish elements that are contained in a dictionary from those that are not. This can be achieved by using the `Maybe` type, which contains two constructors, one for returning regular values of some type `a` and the other for indicating the absence of a value.

```
data Maybe a = Just a | Nothing
```

When we find a key in a dictionary, we return the information stored with it using the `Just` constructor, otherwise we return `Nothing`. For the list implementation of dictionaries, the predefined `lookup` function can be used to find information of type `b` stored with keys of type `a`.

```
> :i lookup
lookup :: Eq a => a -> [(a, b)] -> Maybe b          -- Defined in GHC.List
```

The annotation `Eq a =>` means that the function works only for those types `a` whose elements can be compared for equality. This is required to allow `lookup` to compare the key value to be found with the keys stored in the list. With `lookup` we can inquire about the frequencies of individual characters in the histogram.

```
> lookup 'a' h
Just 4
> lookup 'z' h
Nothing
```

Using the `Maybe` type is the right thing to do in many situations, but in the case of histograms it is appropriate to simply return plain integer values and produce `0` for characters not contained in the histogram.

Exercise 29

Define the function `frequency` that computes the frequency of a letter in a histogram.

```
frequency :: Char -> Histogram -> Int
```

5.2 Histograms as Binary Search Trees

Using lists to implement dictionaries is inefficient since one always has to search for keys starting at the beginning of the list. A *binary search tree* can improve the runtime of operations for keys that are ordered, such as numbers characters, strings, enumeration types, or a combination of those. The efficiency gains are due to the fact that a binary search tree can effectively divide the search space using its keys.

In the type definition for `Tree` in section 4.3 a node can have an arbitrary number of children. By contrast, in a binary tree each node has exactly two children. This is reflected in the following definition in which the `Node` constructor takes exactly two tree arguments instead of a list of trees. The concepts of parent, children, and root also apply to binary trees.

The following definition is parameterized by a type parameter `a`, which means that we can use this tree type to store information of different types, just like with the built-in list type or the type `List` defined in section 4.1.

```
data Tree a = Node a (Tree a) (Tree a)
            | Empty
            deriving Show
```

The `deriving Show` clause produces an automatically defined function for printing trees, but this function might not be adequate, because printing a tree of several levels as a linear sequence

of nested `Node` constructors gets quickly unreadable; we will later have to define a customized function for that.

A binary *search* tree is a binary tree that has the additional property that all keys in a left subtree are smaller than the key in the root and all keys in a right subtree are greater than the key in the root. The type definition does not guarantee this property in any way; it has to be established through the functions that build or manipulate binary search trees.

Similar to the list representation, a histogram can be represented as a tree of (letter, counter) pairs. The trailing `T` in this and the following definitions indicates that they are based on the binary search tree representation.

```
type HistogramT = Tree (Char,Int)
```

Next we need to adapt the histogram-building functions to work with binary search trees. We need new functions for updating the count for a single character, all the characters of one word, and the characters for a list of words. As it turns out, the definitions for the last two are almost identical to their list counterparts. The most interesting function definition is the one for updating a histogram-representing binary search tree for a single character. Like its list counter part it has two cases for an empty and non-empty tree, but the case for non-empty trees consists of three different parts. The case when the character is found in the node is identical to the list case, but if the character is not in the current node, the function distinguishes between the two cases when the character is smaller or larger than the one in the node. Correspondingly, the function continues to update the left or right subtree, respectively.

```
updateT :: HistogramT -> Char -> HistogramT
updateT Empty            c             = Node (c,1) Empty Empty
updateT (Node (k,n) l r) c | c==k      = Node (k,n+1) l r
                           | c<k       = Node (k,n) (updateT l c) r
                           | otherwise = Node (k,n) l (updateT r c)
```

This function definition is fine and does what it's supposed to. However, the definition distinguishes between upper- and lowercase letters. In particular, any uppercase letter is considered to be smaller than any lowercase letter, a fact we can easily verify.

```
> 'Z' < 'a'
True
```

This letter ordering distorts the arrangements of the entries in the binary search tree. For example, the binary search tree we expect to get (following the example in the book) for the word `"God"` is balanced, and the root `('G',1)` has both a left and right subtree. However, the above implementation and the predefined letter ordering produces an unbalanced tree. While this is not a serious problem, it makes it more difficult to follow the example in the book using the code. This issue can be addressed in different ways. On the one hand, we could use only lowercase letters in the example. Then the code doesn't have to be changed at all. On the other hand, we could change the code so that it ignores the distinction between uppercase and lowercase letters. We can achieve this by changing a character to lowercase in all comparisons using the predefined function `toLower`.

```
updateT :: HistogramT -> Char -> HistogramT
updateT Empty            c                        = Node (c,1) Empty Empty
updateT (Node (k,n) l r) c | toLower c==toLower k = Node (k,n+1) l r
                           | toLower c<toLower k  = Node (k,n) (updateT l c) r
                           | otherwise            = Node (k,n) l (updateT r c)
```

The `extendBy` function repeatedly calls `update` for the characters from its string argument. The corresponding `extendByT` function has to do the same thing, except that it has to use the `updateT` function. Thus, we can reuse the previous definition almost verbatim.

```
extendByT :: HistogramT -> String -> HistogramT
extendByT = foldl updateT
```

At this point let's construct histograms for the words `"God"` and `"Iehova"`.

```
t1 = Empty `extendByT` "God"
t2 = t1 `extendByT` "Iehova"
```

Printing the two histograms illustrates the need for an improved printing strategy.

```
> t1
Node ('G',1) (Node ('d',1) Empty Empty) (Node ('o',1) Empty Empty)
> t2
Node ('G',1) (Node ('d',1) (Node ('a',1) Empty Empty) (Node ('e',1) Empty
Empty)) (Node ('o',2) (Node ('I',1) (Node ('h',1) Empty Empty) Empty)
(Node ('v',1) Empty Empty))
```

The structure of `t2` is very difficult to discern. Producing a nice graphical rendering of a tree is not a trivial task. In the interest of getting a relatively simple implementation quickly, I will present an implementation that produces a left-to-right rendering. The idea is to first define a function that takes an additional string parameter that is used to produce indentations for subtrees so that all nodes on each level are effectively lined up in columns. Here is the rendering for the tree `t1`.

```
        ('o',1)
('G',1)
        ('d',1)
```

And this is the output that is produced for `t2`.

```
                ('v',1)
        ('o',2)
                        -
                ('I',1)
                        ('h',1)
('G',1)
                ('e',1)
        ('d',1)
                ('a',1)
```

If you rotate the displayed text clockwise by 90°, you obtain the figures used in the book (minus the lines connecting the nodes). Omitting empty subtrees from all leaf nodes and the indentation makes the tree structure easier to discern. Note also how all entries are sorted alphabetically from bottom to top.

This output can be produced by the following function `showTS`, which prints each node on a separate line.[36] It uses the first argument string as a prefix to indent the node by the right amount

[36] The suffix `T` stands for "**t**ree", and the suffix `S` indicates that the function uses a parameter for printing additional space at the beginning of each line.

of empty space. The first two cases for empty trees and leaves are obvious: they simply concatenate the indentation `s`, the node value, which is turned into a string using the `show` function defined for that value's type,[37] and a newline character `\n`, which has to be turned into a list of characters using the string quotes. In the case of a node with at least one non-empty subtree, this function first computes the printable representation of the node value plus its indentation and a newline character at the end. It then computes with the function `length` the number of characters in it, and produces a string of as many space characters `' '` using the function `replicate`, which is the indentation for the subtrees. The function then produces the lines for the right subtree, followed by the current line `c`, followed by the lines for the left subtree.

```
showTS :: Show a => String -> Tree a -> String
showTS s Empty = s++"-\n"
showTS s (Node x Empty Empty) = s++show x++"\n"
showTS s (Node x l r) = let c=s++show x++"\n"
                            s'=replicate (length c) ' '
                        in showTS s' r++c++showTS s' l
```

The function `showTS` will be called with an initially empty indentation `""` and the tree to be printed. This is handled by the function `showT`. The result of the function is a `String` value. Since we don't want to look at the `String` representation, but rather have it be printed on the terminal, we have to use another function `putStr`.[38] The function `pT` combines `putStr` and `showT` and finally provides us with a way of rendering tree output within the GHC interpreter.

```
showT :: Show a => Tree a -> String
showT = showTS ""

pT :: Show a => Tree a -> IO ()
pT t = putStr (showT t)
```

Coming back to the histogram application, building a histogram through the repeated use of `extendByT` can be captured by a function `addWordsT`, again, just what we did for the list representation. And as with `extendByT`, we can reuse the previous definition and simply rename the functions that affect the representation.

```
addWordsT :: HistogramT -> [String] -> HistogramT
addWordsT h ws = foldl extendByT h (map nub ws)
```

The histogram of character frequencies can now be built using the `addWordsT` function.

```
t = addWordsT Empty ["god","iehova","jehova","yahweh","yehova"]
```

And we can finally also look at the tree representation of the histogram by applying the function `pT`.

[37] The `Show a =>` annotation in the type restricts the type of `showTS` to only those types `a` for which a `show` function is defined. This function is defined for all predefined types and can in most cases be automatically derived for user-defined types.

[38] The type `IO ()` says that `putStr` is a function that does not compute a value but only has an input/output effect.

```
> pT t
                                -
                        ('Y',2)
                                ('w',1)
                ('v',3)
                        -
        ('o',4)
                        ('J',1)
                ('I',1)
                        ('h',4)
('G',1)
                ('e',4)
        ('d',1)
                ('a',4)
```

It is interesting to observe that the structure of the binary search tree depends on the order in which the elements to be inserted are encountered. This fact is already observable with small examples.

```
> pT (Empty `extendByT` "God")
        ('o',1)
('G',1)
        ('d',1)
> pT (Empty `extendByT` "doG")
                -
        ('o',1)
                ('G',1)
('d',1)
        -
```

Finally, we want to compute letter frequencies for tree-represented histograms as we did for the list representation. There are several ways of doing this. A quick solution is obtained by converting a tree into a list and then using the function we have defined for lists.

Exercise 30

Define a function `inorder` that computes the list of nodes in a tree from left to right, that is, the nodes in a left subtree should become before the root, and the nodes in the right subtree should follow the root.

```
inorder :: Tree a -> [a]
```

The function should produce the following output.

```
> inorder t1
[('d',1),('G',1),('o',1)]
> inorder t2
[('a',1),('d',1),('e',1),('G',1),('h',1),('I',1),('o',2),('v',1)]
```

► *Sorting* Chapter 6

You may have noticed that the output is sorted by characters. This may suggest the use of a binary search tree with an `insert` (see section 5.3) and `inorder` function to sort data. However, the efficiency of the resulting sorting algorithm depends on the order in which elements appear in the input, since the order in which the elements are inserted into a tree determines its structure.

And as we have seen, in some cases the tree structure might degenerate to a list, which negatively affects the efficiency of the search tree.

Exercise 31

Define the function `maxCountT` that computes the frequencies of letters in the tree representation of a histogram.

```
maxCountT :: HistogramT -> (Int,String)
```

(**Hint**: simply compose the functions `inorder` and `maxCount`.)

Similarly to the list implementation, we may want to have a function for computing frequencies of individual letters for the trie representation.

Exercise 32

Define the function `frequencyT` that computes the frequency of a letter in a histogram represented as a binary search tree.

```
frequencyT :: Char -> HistogramT -> Int
```

5.3 Tries

A *trie* is a data structure that is specialized for storing strings or (similarly structured types). It works particularly well when the strings to be stored have many prefixes in common. For example, whereas a binary search tree represents the strings `"mat"`, `"map"`, and `"max"` separately, a trie will share their common prefix `"ma"` and represent it only once.

A trie can be realized as a binary search tree that stores individual characters in nodes and places continuations of strings in left subtrees and alternatives to characters in right subtrees.

```
type Trie = Tree Char
```

As an example, this is how the two words `"map"` and `"mat"` are stored with the described representation using left and right subtrees.

```
    -
'm'
        -
    'a'
            't'
        'p'
            -
```

Both words start at the root with the character `'m'` and are continued in the left subtree with the character `'a'`. The third character of the first word, `'p'`, continues the word in the left subtree of `'a'`. Since the third character of the second word, `'t'`, is different it occurs in the right subtree of `'p'`, indicating an alternative continuation of `"ma"`. If we add the word `"max"`, the first two characters will again be shared, and the last character `'x'` will appear as an alternative continuation to both `'p'` and `'t'` in the right subtree of `'t'`.

```
    -
'm'
        -
    'a'
                'x'
            't'
                -
        'p'
            -
```

Instead of the plain visualization of binary search trees, tries can be a more compactly printed by aligning chains of left subtrees horizontally (without separating space) and right subtrees vertically, starting in the column where its parent is located. This makes words (and word suffixes) much more easy to read. For example, the trie for the words `"map"` and `"mat"` looks as follows.

```
map
..t
```

The dots indicate that the closest character above should be substituted to complete the word represented. Adding the word `"max"` leads to the following output:

```
map
..t
..x
```

A function for printing tries in the shown way can be defined as follows. In addition to the tree being printed, the function uses one argument `s` to track the indentation of the current subtree and a boolean flag `b` that indicates whether the indentation should be printed or whether it should be ignored. The indentation should be ignored when the root of the current subtree follows the last printed character on the current line (in the example, when we print the `'a'` after the `'m'`). For all non-empty trees, we print the root using the auxiliary function `indent`, followed by the left and right subtrees if they exist. For the left subtree the indentation is increased and the flag is set to `False` since left subtrees continue the current word. By contrast, since the right subtree indicates an alternative word continuation, it has to be printed on a new line, which is why the newline character is added and the indentation has to be printed to align the subtree horizontally with the current root.

```
indent :: String -> Bool -> Char -> String
indent s True  c = s++[c]
indent s False c = [c]

showTrS :: String -> Bool -> Trie -> String
showTrS _ _ Empty = ""
showTrS s b (Node c Empty Empty) = indent s b c
showTrS s b (Node c l Empty)     = indent s b c ++ showTrS ('.':s) False l
showTrS s b (Node c l r)         = indent s b c ++ showTrS ('.':s) False l
                                       ++ "\n" ++ showTrS s True r
```

The function `showTrS` will be called with an initially empty indentation `""`, the value `False`, and the tree to be printed. This is done by the function `showTr`. As in the case of `pT`, we don't want to look at the `String` representation, but have it be printed on the terminal. To this end, we define another function `pTr` that combines `putStr` and `showTr`.

```
showTr :: Trie -> String
showTr = showTrS "" False

pTr :: Trie -> IO ()
pTr = putStrLn . showTr
```

We can now print tries in the format shown, but how do we insert words into a trie? To this end, we define a function `insertT` that takes a string (that is, a list of characters) and uses it to navigate along the trie, inserting only characters that are not yet covered by prefixes in the trie. In the base case when the string is empty, the trie is not changed. Otherwise, when a word with first character `c` is to be inserted into an empty trie, we create a new node with `c` as its root and continue inserting the rest of the word `cs` into its left subtree while the right subtree remains empty. If the trie is not empty, we have to compare the character to be inserted with the character in the root. If they are equal, we have to continue to insert the remaining word into the left subtree, since the current prefix of the word is already represented in the trie. Otherwise, we have to insert the whole word `c:cs` as an alternative to the current character into the right subtree.

```
insertT :: String -> Trie -> Trie
insertT []     t                        = t
insertT (c:cs) Empty                    = Node c (insertT cs Empty) Empty
insertT (c:cs) (Node x l r) | x==c      = Node x (insertT cs l) r
                            | otherwise = Node x l (insertT (c:cs) r)
```

To build a trie from a list of words, we can borrow the definition of the function `addWords` almost verbatim to define the function `buildT`, since the structure is the same: a function for changing the data structure is applied repeatedly for all elements of a list. However, the arguments of the function `insertT` appear in the wrong order to apply the function `foldl`. If only `insertT` had the type `Trie -> String -> Trie` instead, we could use `foldl`. We could, of course, redefine `insertT` to make the type match, but a simpler solution is to use the built-in function `flip`, which takes a function and changes the order of its arguments.

```
build :: [String] -> Tree String
build = foldl (flip insert) Empty
```

With `build` we can construct the trie of words `"bag"`, `"bat"`, `"beg"`, and `"bet"` and check whether binary tree structure and the improved trie printing match the specifications.

```
> let tr3 = buildT ["bag","bat","beg","bet"]
> pT tr3
    -
'b'
          -
        'e'
              't'
          'g'
              -
    'a'
          't'
        'g'
          -
```

```
> pTr tr3
bag
..t
.eg
..t
```

Note that we have used a simplified version of tries that cannot distinguish between prefixes of a word that are elements of a trie and those that are not. For example, inserting the word `"mate"` into an empty trie results in a chain of left subtrees, which represent that word, but what about the word `"mat"`? Is it also an element of this trie? Since we haven't inserted it yet, the answer should be "no." However, since inserting the word `"mat"` does not change the trie, we can't distinguish these two cases. It is not difficult to extend the trie data structure to allow the representation of word prefixes as elements of tries. This is left as an exercise.

❋ ❋ ❋

Chapters 6-15 coming soon ...