Learning Haskell by Solving Problems

A Practical Handbook on Functional Programming

Gonçalo Leão

September 30, 2024

Contents

Chapter 1
Introduction

Haskell is a functional programming language, or a language that follows the *functional programming paradigm*. A programming paradigm is a way of thinking, a means of reasoning about the way one writes computer programs.

Functional programming languages put a lot of emphasis on functions. Unlike imperative languages, where the execution model is running an ordered list of instructions, in functional languages all computations are the result of a call to a function. In addition, functions are *first-class citizens*, which means that they can be used as if there were any other sort of "conventional variables" (such as integers or strings). For instance, they can be passed as an argument to another function (the latter are called *higher-order functions*) or be the return value of a function.

To be more accurate, Haskell is a purely functional language, since all entities are immutable (the value of a variable cannot be updated), and functions calls are free of side effects (such as printing something in the screen or using global variables) and deterministic (two calls to the same function using the same arguments guarantees it yields the same result). Working with a pure language have several benefits since it is easier to perform parallel computing, programs are clearer and it is easier to prove the correctness of an algorithm. This last advantage is associated with the ease of providing mathematical formalism to Haskell code (a = 1 in Haskell means actual equality rather than assignment like in C). Functional programming has a very close connection to *lambda calculus*.

Haskell has three relevant properties:

- It is *statically-typed* (just like C or Java), which means that the type of all the variables is known at compile-time, rather than on run-time (like in dynamically typed languages, such as Python or JavaScript). This has the advantages of allowing many errors to be caught during earlier stages of code development and enabling the compiler to perform more optimizations (and thus allow the code to run faster).
- It is *strongly typed* (just like Java and Python), which means that variables are tightly bound to a data type and type errors are raised if the types used in an expression do not match with the expected ones.

- It is *lazy*, which means that expressions are not computed until they are actually needed (*lazy evaluation*). This enables the programmer to work with infinite data structures without necessarily running into endless loops and allows for increases in performance (by avoiding some unnecessary computations), though it makes it harder to reason about the temporal and spatial complexity of a function.

This introductory chapter aims to help readers get acquainted with the programming language by providing examples of how to write expressions and functions. Basic conditional structures and recursion are introduced to allow readers to develop basic functions in Haskell.

1.1 Getting started

To start your journey into functional programming and Haskell, you will need a Haskell interpreter (which processes and executes your code) and a text editor (where you can develop your programs).

One of the widely used compilers is GHC (Glasgow Haskell Compiler). This book recommends using the Haskell Platform, which includes this compiler and many other useful tools (for debugging, profiling, documentation ...). This platform is portable to multiple operating systems, including Windows, Linux and OS X.

Let us write a simple program to illustrate the software development process in Haskell. Haskell code files typically have the .hs extension. Create a file called test.hs and add the following line of code:

```
main = putStrLn "Hello, world!"
```

To use the GHC to compile a program and then execute it, the following sequence of commands can be used:

```
$ ghc test.hs
$ ./test
```

The first line produces (among other files) an object file, test.o. The second line runs the program, which should print "Hello, world!" in the screen. Congratulations, you wrote your first program in Haskell! For more information about how to use GHC, please consult the official manual.

GHC comes with an interactive environment called GHCi, which can be used to compute expressions and interpret Haskell programs in an interactive manner. To run the "Hello, world" example via GHCi, the following sequence of commands can be used:

```
$ ghci
Prelude> :load test.hs
*Main> main
```

The first line accesses the interactive environment. The second line loads the Haskell file without executing the code within it. Instead of writing `:load` , the shorthand version `:l` can also be used. The third line runs the function "main" which prints the message on the screen. The official manual can be consulted for more information about how to use GHCi.

1.2 Simple expressions

In this section, we will use the GHCi environment to evaluate some simple numeric expressions. The interpreter reads expressions, computes their values and prints them in the console, in a loop.

Haskell uses functions to perform any operation. It can also use operators, which are simply functions written in a more user-friendly manner (usually using infix notation). Operators can be used as functions by surrounding their symbol with parenthesis, whereas functions can be used as operators by surrounding them with backticks. Here are some examples:

```
Prelude> (+) 3 4
7
Prelude> 5 `mod` 2
1
Prelude> (+)(5 `mod` 2 + 2)(mod 5 2)
4
```

Table 1.1 presents some common basic operators and functions of Prelude, a standard module in Haskell with useful functions that is imported by default.

Table 1.1 Some basic numeric and logical operators in Haskell

Operator	Comment
`+`, `-`	Addition, subtraction
`*`, `/`	Multiplication, division (floating-point)
`^`, `^^`, `**`	Exponentiation (non-negative integer, integer, floating-point)
`div`	Integer division quotient
`mod`	Integer division remainder
`==`	Equality
`/=`	Inequality
`<`, `<=`, `>`, `>=`	Comparison
`not`	Negation
`&&`, `||`	Logical conjunction and disjunction

Special attention should be taken when writing expressions with negative numbers, as sometimes it is required to surround a negative number with parenthesis to avoid syntactic errors (due to ambiguities between the minus sign and the subtraction operator). For instance, to add 3 and -3 one must write:

```
Prelude> 3 + (-3)
0
```

As a rule of thumb, if one is unsure about the precedence of the operations in an expression, one should surround a sub-expression with parenthesis to ensure it is computed first.

Using GHCi, it is possible to "cheat" and determine the precedence of operators by running the ghci command with some special flags and then writing the expression to test in the format `\$([| expression |])`, as shown in the following example:

```
$ ghci -ddump-splices -XTemplateHaskell
Prelude> $([|1+1*2|])
```

```
<interactive>:6:3-11: Splicing expression
[| 1 + 1 * 2 |] ======> (1 + (1 * 2))
3
```

It should be noted that the above method does not always return what one would expect at first. For instance, the minus sign is replaced by the negate function, which inverts the sign of a number:

```
Prelude> $([|  (-1) |])
<interactive>:53:3-13: Splicing expression
[| (- 1) |] ======> negate 1
-1
```

Sample exercises

Sample exercise IN-1

Problem statement

Add all the implicit parentheses to the following expressions to clarify which operations are performed first.

a) `3 + (-3)* 2`

b) `sqrt 4 + 4`

c) `sqrt (4 + 4)`

d) `1 + 2 + 3 + 4`

e) `2^3^3`

f) `100 `div` 4 `div` 3`

g) `(+) 3 (-3)* 2`

Solution

a) `3 + ((-3)* 2)`

Parenthesis are added to the multiplication operator since it is performed before addition.

b) `(sqrt 4)+ 4`

Function application (i.e. writing a function's name followed by its arguments) has precedence over all other operators.

c) `sqrt (4 + 4)`

This expression already has all the parenthesis.

d) `((1 + 2)+ 3)+ 4`

Addition is left-associative, similarly to other basic operators, including subtraction, multiplication and division (for floating-point numbers and integers).

e) `2^(3^3)`

Exponentiation is right-associative.

f) `(100 `div` 4)`div` 3`

Binary functions converted to operators using backticks are left-associative by definition.

g) `(3 + (-3))* 2`

This is a tricky one! The addition comes before the multiplication because the addition is used as a function application.

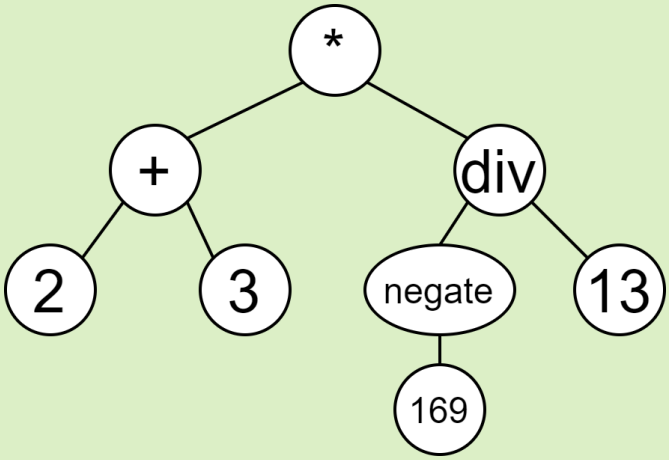
Sample exercise IN-2

Problem statement

Draw a syntax tree for the following expression:

```
(+) 2 3 * (negate 169 `div` 13)
```

Solution



These syntax trees allow one to understand the structure of an expression in Haskell more easily. The leaves represent the most elemental operands, while the nodes with children correspond to functions/operators.

Practice exercises

Practice exercise IN-3

Problem statement

Add all the implicit parentheses to the following expressions to clarify which operations are performed first.

a) `3 - (-2)+ 1`

b) `4 / (-2)- 3 * 6`

c) `(-)2 3 * 6`

d) `100 `div` 4 `div` 3`

e) `100 `div` div 4 3`

f) `(+)1 2 + 3 * 4`

g) `(+)(5 `mod` 2 + 2)(mod 5 2)`

Practice exercise IN-4

Problem statement

Draw a syntax tree for the following expression:

`72 / 9 + (-)5 4 * ((-)14 7 / 7)`

Solutions to the practice exercises

Practice exercise IN-3

Solution

a) `(3 - (-2))+ 1`

Addition and subtraction have the same level of precedence, so the operations are simply performed from left to right.

Parentheses are required for the negative number to avoid confusions with the (binary) subtraction operator.

b) `(4 / (-2))- (3 * 6)`

Division and multiplication come before subtraction.

c) `(2 - 3)* 6`

The subtraction comes before since the `-` operator is written in prefix notation.

d) `(100 `div` 4)`div` 3`

The `div` operations are executed from left to right. The result is 8.

e) `100 `div` (div 4 3)`

"Prefix" `div` comes before "infix" `div` . The result is 100.

f) `((+)1 2)+ (3 * 4)`

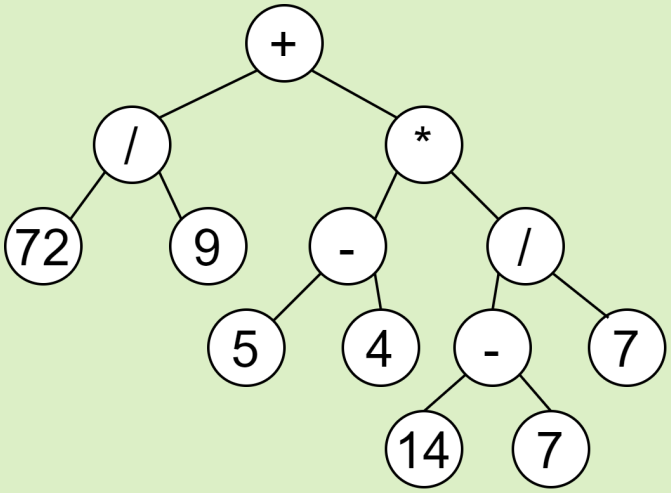
"Prefix" addition and multiplication come before "infix" addition.

g) `(+)((5 `mod` 2)+ 2)(mod 5 2)`

The outermost addition is performed once the expression on both its operands is computed.

Practice exercise IN-4

Solution



The outermost operator is `+`. In the expression on the right operand of `+` ↪ , both "prefix" subtractions are performed first, then the division (due to parentheses) and finally the multiplication.

1.3 Simple functions

In Haskell, since functions behave like any other entity, they are defined using the same syntax as variables. In fact, variables (in the sense as those used in imperative languages) are a particular case of a function with no arguments.

The general syntax to define a function is:

```
<function name> <argument 1> <argument 2> ... = <expression>
```

Valid function and argument names begin with a lowercase letter, followed by letters, numbers, underscores (example: `func_2`) and apostrophes (`func'`). The names

used cannot be any of the following reserved keywords:

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

Note: In this chapter, exceptionally, the type of the functions to be implemented will be presented in the problem statement (example: `double :: Num a => a -> a`). These types are explained at the end of Chapter 2, where they will no longer be provided in the problem statement for the remainder of the book. From that section onwards, the reader is expected to define the functional types themselves.

Sample exercises

Sample exercise IN-5

Problem statement

Define the following functions:

a) double, which returns the double of number.

```
double :: Num a => a -> a
```

b) nand, which returns the negation of the logical conjunction of two booleans.

```
nand :: Bool -> Bool -> Bool
```

c) funcX, which corresponds to the following equation:
$funcX(a, b, c, x) = aT^2 + bT + c$, where $T = cos(x) + sin(x)$

```
funcX :: Floating a => a -> a -> a -> a -> a
```

Usage examples:

```
*Main> double 4.5
9.0
*Main> nand True False
True
*Main> funcX 1 2 3 4
2.168466014280302
```

Solution

a)

```
double x = 2*x
```

b)

```
nand x y = not(x && y)
```

c)

```
-- version with let
```

```
funcX a b c x =
  let t = cos x + sin x in
  a*t^2 + b*t + c
-- version with where
funcX' a b c x =
  a*t^2 + b*t + c
  where t = cos x + sin x
```

Alternative implementations of functions will end with an apostrophe in this book.

This function definition introduces auxiliary expressions. This can be accomplished using the `let` or `where` keywords.

Haskell is indentation-sensitive: definitions on the same scope should be aligned (by adjusting the number of spaces at the start of each line of code).

Single-line comments use `--`, while multi-line comments use `{-` and `-}`.

Practice exercises

Practice exercise IN-6

Problem statement

Define the following functions:

a) half, which returns half of a number number.

```
half :: Fractional a => a -> a
```

b) xor, which returns the exclusive-or of two booleans.

```
xor :: Bool -> Bool -> Bool
```

c) cbrt, which returns the cubic root of a number.

```
cbrt :: Floating a => a -> a
```

d) heron, which computes the area of a triangle using the length of its three side using Heron's formula:
$A = \sqrt{s(s-a)(s-b)(s-c)}$, where $s = \frac{a+b+c}{2}$

```
heron :: Floating a => a -> a -> a -> a
```

Usage examples:

```
*Main> half (-5)
-2.5
*Main> xor True False
True
*Main> cbrt 8
2.0
*Main> heron 4 13 15
24.0
```

Practice exercise IN-7

Problem statement

a) Implement isTriangular which returns a boolean indicating whether the values of its three arguments could correspond to the side lengths of a triangle. In a triangle, the length of any of its sides is smaller or equal to the sum of the other two lengths (triangle inequality).

```
isTriangular :: (Ord a, Num a) => a -> a -> a -> Bool
```

b) Implement isPythagorean which returns a boolean indicating whether its three arguments form a Pythagorean triple: $A^2 + B^2 = C^2$, where A, B and C can be the function's arguments a, b and c in any order.

```
isPythagorean :: (Num a, Eq a) => a -> a -> a -> Bool
```

Usage examples:

```
*Main> isTriangular 1 5 6
False
*Main> isPythagorean 3 4 5
True
```

Solutions to the practice exercises

Practice exercise IN-6

Solution

a)

```
half x = x/2
```

b)

```
xor a b = (a && (not b)) || ((not a) && b)
```

c)

```
cbrt x = x**(1/3)
```

The `**` operator must be used for exponentiation since the exponent, $\frac{1}{3}$, is non-integer.

d)

```
-- version with let
heron a b c =
  let s = (a+b+c)/2 in sqrt(s*(s-a)*(s-b)*(s-c))
-- version with where
heron' a b c =
```

```
  sqrt(s*(s-a)*(s-b)*(s-c))
  where s = (a+b+c)/2
```

Practice exercise IN-7

Solution

a)

```
isTriangular a b c =
    (a + b <= c) && (a + c <= b) && (b + c <= a)
```

b)

```
isPythagorean a b c =
    (a^2 + b^2 == c^2) || (a^2 + c^2 == b^2) || (b^2 + c^2 == a^2)
```

1.4 Conditional structures

There are various ways of writing conditional structures in Haskell, which, given a condition, determine which expression will be computed. The four main conditional structures are:

- if-then-else expressions.
- guards.
- pattern matching.
- case expressions.

Sample exercises

Sample exercise IN-8

Problem statement

Implement five, which returns the string "five" if its sole argument is equal to 5, and "not five" otherwise.

```
five :: (Eq a, Num a) => a -> [Char]
```

Usage examples:

```
*Main> five 5
"five"
*Main> five 4
```

```
"not␣five"
```

Solution

```
-- version with if-then-else
five n = if n == 5 then "five" else "not␣five"

-- version with guards
five' n
    | n == 5 = "five"
    | otherwise = "not␣five"

-- version with pattern matching
five'' 5 = "five"
five'' _ = "not␣five"

-- version with case expression
five''' n = case n of 5 -> "five"
                      _ -> "not␣five"
```

This solution presents the four main alternatives of writing conditional structures.

if-then-else structures are expressions, and thus result in a value, and not statements (like in imperative languages). The `else` cannot be omitted, since a result must always be returned by the function.

The second solution uses guards, which return the expression of the first function definition (read from top to bottom) that meets the corresponding condition. A common mistake when working with guards is adding an equal sign (`=`) after the function's arguments. `otherwise` is a special keywords used to designed cases that are not captured by the guards.

In the third solution, with pattern matching, each pattern is tested against the function's arguments until a matching pattern is found. The function body of the matching definition is then executed.

In the last solution, with a case statement, the underscore (`_`) on the second pattern corresponds to a "catch-all" definition where we do not care about the value of n (i.e. do not use it in any computation).

Sample exercise IN-9

Problem statement

Implement the min3 function, that returns the smallest element of the three arguments.

```
min3 :: Ord a => a -> a -> a -> a
```

Usage examples:

```
*Main> min3 3 1 2
1
*Main> min3 4.0 4.5 (-4.9)
```

```
-4.9
```

Solution

```
-- version with if-then-else
min3 x y z =
    if x < y && x < z
        then x
        else if y < z
            then y
            else z
-- version with guards
min3' x y z
  | x < y && x < z  = x
  | y < z           = y
  | otherwise       = z
```

Sample exercise IN-10

Problem statement

The pH of a solution indicates how acid or alkaline it is. This metric is linked to the concentration of H^+ ions in the solution.

Implement the testPh function that receives as input the hydrogen ion activity ah and returns a string indicating the type of the solution: "acid" ($pH < 7$), "neutral" ($pH = 7$) or "basic" ($pH > 7$). The pH is linked to the ion activity ah by the equation: $pH = -log_{10}(ah)$.

```
testPh :: (Ord a, Floating a) => a -> [Char]
```

Usage examples:

```
*Main> testPh 5e-6
"acid"
```

Solution

```
testPh ah
  | ph <= 7.0 = "acid"
  | ph == 7.0 = "neutral"
  | ph >= 7.0 = "basic"
  where ph = -logBase 10 ah
-- version with the magic number defined in a variable
testPh' ah
  | ph <= neutral_ph = "acid"
  | ph == neutral_ph = "neutral"
  | ph >= neutral_ph = "basic"
  where ph = -logBase 10 ah
        neutral_ph = 7.0
```

In the second solution, the constant is stored in a variable (within a `where` block) to avoid its repetition.

Practice exercises

Practice exercise IN-11

Problem statement

Implement the max3 function, that returns the largest element of the three arguments.

```
max3 :: Ord a => a -> a -> a -> a
```

Practice exercise IN-12

Problem statement

The BMI (body mass index) is measured using a person's weight and height.

Implement the testBMI function that receives as inputs a weight (in kg) and height (in m) and returns a string with person's category: "underweight" ($bmi < 18.5$), "healthy weight" ($18.5 <= bmi < 25.0$), "overweight" ($25.0 <= bmi < 30.0$) and "obese" ($bmi >= 30.0$). The bmi ratio is computed using the equation $bmi = weight/height^2$.

```
testBMI :: (Ord a, Fractional a) => a -> a -> [Char]
```

Usage examples:

```
*Main> testBMI 64.0 1.64
"healthy weight"
```

Practice exercise IN-13

Problem statement

Consider the definition of function f below:

```
f :: (Ord a, Num a, Integral b) => a -> b
f x
    | x > 0 = 1
    | x < 0 = -1
    | x == 0 = 0
```

a) Explain concisely what function f computes.

b) Implement function f without using guards and using at most one if-then-else expression.

Solutions to the practice exercises

Practice exercise IN-11

Solution

```
-- version with if-then-else
max3 x y z =
    if x > y && x > z
        then x
        else if y > z
            then y
            else z
-- version with guards
max3' x y z
  | x > y && x > z  = x
  | y > z           = y
  | otherwise       = z
```

Practice exercise IN-12

Solution

```
testBMI weight height
  | bmi < bmi_underweight = "underweight"
  | bmi >= bmi_underweight && bmi < bmi_overweight = "healthy weight"
  | bmi >= bmi_overweight && bmi < bmi_obese = "overweight"
  | bmi >= bmi_obese = "obese"
  where bmi = weight / height^2
        bmi_underweight = 18.5
        bmi_overweight = 25.0
        bmi_obese = 30.0
```

Practice exercise IN-13

Solution

a) Function f computes the sign of a number x, and returns 1, -1 or 0, respectively, if x is positive, negative or null.

b)

```
f' :: (Ord a, Num a, Integral b) => a -> b
f' 0 = 0
f' x = if x > 0 then 1 else -1
```

A pattern is used to cover the case where x is equal to 0. This avoids the need to use a nested if-then-else expression in the second definition.

1.5 Recursion

In Haskell, there is not iteration, namely “for” and “while” cycles. To execute a fragment of code a certain number of times until a condition is met, one must use *recursion*, where a function’s expression contains a call to itself.

A recursive function f has two types of clauses:

- a base case, which corresponds to a “simpler"/“smaller"/trivial input value for which the output expression corresponds directly to the result.
- a recursive step, where f’s output is defined as a function of f for a “slighlty simpler"/“smaller"/more trivial input value. The value of f with a “simpler" input value is obtained using a *recursive call*.

Functions can have multiple base cases and recursive steps.

Sample exercises

Sample exercise IN-14

Problem statement

Implement the factorial function that computes the product of an integer n by all the numbers between 1 and n.

```
factorial :: (Ord p, Num p) => p -> p
```

Usage examples:

```
*Main> factorial 6
720
```

Solution

```
-- version with pattern matching and guards
factorial 0 = 1
factorial n
    | n > 0 = n * factorial(n-1)
    | n < 0 = error "negative argument"
-- version with if-then-else
factorial' n =
    if n == 0 then 1
    else if n > 0
        then n * factorial'(n-1)
        else error "negative argument"
```

In both alternatives, the base case corresponds to when the argument is equal to 0. The recursive step is executed when the argument is positive.

If the argument is negative, then a special function, error, is run which interrupts the program execution with an run-time error accompanied by a message.

As shown in the first example, one can have nested if-then-else expressions.

Sample exercise IN-15

Problem statement

Implement the myGcd function, which computes the greatest common divisor of two integers. This value can be computed using the following recursive definition which assumes $|a| >= |b|$:

$$gcd(a,b) = \begin{cases} |a| & b = 0 \\ gcd(b, a \mod b) & |b| > 0 \end{cases}$$

```
myGcd :: Integral a => a -> a -> a
```

Note: the function is called myGcd and not simply gcd, since the latter already exists in the Prelude.

Usage examples:

```
*Main> myGcd 12 8
4
*Main> myGcd 1280 (-860)
20
```

Solution

```
myGcd a b = myGcdAux (max posA posB) (min posA posB)
    where posA = abs a
          posB = abs b

myGcdAux a 0 = a
myGcdAux a b = myGcdAux b (mod a b)
```

The main function, myGcd, resorts to an auxiliary function, myGcdAux, to perform the actual computations. The auxiliary method assumes that both its arguments are positive and that the first argument is not smaller than the second one. The main function ensures these assumptions are met.

Practice exercises

Practice exercise IN-16

Problem statement

Without using the exponentiation operators, implement the mPower function that computes the nth power of m, m^n, where m is a real number and n is an integer. Note: n can be negative.

```
mPower :: (Fractional a, Integral t) => a -> t -> a
```

Usage examples:

```
*Main> mPower 3 6
729.0
*Main> mPower 2 (-4)
6.25e-2
```

Practice exercise IN-17

Problem statement

Implement fib which computes the nth number of the Fibonacci sequence ($n >= 0$). This sequence starts with a 0 followed by a 1. The following numbers are the sum of the two preceding elements in the sequence.

```
fib :: (Num a, Ord a, Num p) => a -> p
```

Usage examples:

```
*Main> fib 0
0
*Main> fib 1
1
*Main> fib 2
1
*Main> fib 3
2
*Main> fib 10
55
```

Practice exercise IN-18

Problem statement

Implement the ackermann function, which implements the Ackermann function. This function is defined as follows for non-negative integers m and n:

$$ack(m,n) = \begin{cases} n+1 & m = 0 \\ ack(m-1,1) & m > 0, n = 0 \\ ack(m-1, ack(m,n-1)) & m > 0, n > 0 \end{cases}$$

```
ackermann :: (Num a, Ord a, Num t, Ord t) => a -> t -> t
```

Usage examples:

```
*Main> ackermann 3 0
5
*Main> ackermann 0 3
4
*Main> ackermann 3 3
61
```

Practice exercise IN-19

Problem statement

Implement pascal, which computes the value on the nth row and kth column of the Pascal triangle, which is denoted by $\binom{n}{k}$. It is assumed that $k <= n$.The values are computed using the following equation:

$$\binom{n}{k} = \begin{cases} 1 & k = 1 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 1 < k < n \end{cases}$$

```
pascal :: (Num a, Ord a, Num p) => a -> a -> p
```

Usage examples:

```
*Main> pascal 5 10
126
*Main> pascal 6 10
126
*Main> pascal 7 10
84
```

Practice exercise IN-20

Problem statement

Implement isPrime which returns a boolean indicating whether a positive integer is a prime number or not. A prime number p contains exactly two positive divisors: 1 and p. Thus, 1 is not a prime number.

```
isPrime :: Integral t => t -> Bool
```

Usage examples:

```
*Main> is_prime 37
True
*Main> is_prime 457
True
*Main> is_prime 459
False
```

Solutions to the practice exercises

Practice exercise IN-16

Solution

```
mPower m n
  | n == 0 = 1
  | n > 0 = m * mPower m (n-1)
  | otherwise = 1 / mPower m (-n)
```

The function has one definition for the base case, and two for the recursive cases (for positive and negative exponents). To compute negative exponents, the following relation was used: $m^(- n) = \frac{1}{m^n}$.

Practice exercise IN-17

Solution

```
fib 0 = 0
fib 1 = 1
fib n
    | n > 0 = fib(n-2) + fib(n-1)
    | otherwise = error "negative argument"
```

This problem is interesting as it has two base cases rather than one. Also, this implementation makes two recursive calls in the recursive case.

Practice exercise IN-18

Solution

```
ackermann 0 n = n+1
ackermann m 0 = ackermann (m-1) 1
ackermann m n
    | (m > 0) && (n > 0) = ackermann (m-1) (ackermann m (n-1))
    | otherwise = error "negative argument"
```

This function performs a recursive call in one of the arguments of the recursive call.

Practice exercise IN-19

Solution

```
pascal 1 _ = 1
pascal k n
```

```
  | k > 1 && k == n = 1
  | k > 1 && n > k = pascal (k-1) (n-1) + pascal k (n-1)
  | k > 1 && n < k = error "n must be greater or equal than k"
  | otherwise = error "k must be greater than 0"
```

This function has two base cases and performs two recursive calls in the recursive definition.

Practice exercise IN-20

Solution

```
isPrime n
  | (n == 2) = True
  | (n <= 1) = False
  | (mod n 2 == 0) = False
  | otherwise = not(hasPrimeFactor 3 n)

hasPrimeFactor i n
  | (i*i > n) = False
  | (mod n i == 0) = True
  | otherwise = hasPrimeFactor (i + 2) n
```

It can be proven that all composite (i.e. non-prime) numbers n contain a prime number divisor p such that $p <= sqrt(n)$.

An auxiliary function, hasPrimeFactor, is defined to check if a number has a prime divisor. The first argument of this function is an iterator i that checks if i divides n. The function does not test if i is prime to make the function more efficient. Instead, it excludes some non-prime divisors by only checking odd numbers (notice that the recursive call increments i by 2 and not by 1), since all prime numbers are odd, with the exception of 2.

The auxiliary function returns `False` if the iterator surpasses $sqrt(n)$. To make the function more efficient, the $i^2 > n$ inequality is used rather than $i > sqrt(n)$, to avoid computing square roots.

Chapter 2
Fundamentals on types

Since Haskell is a strongly-typed language, it performs strict checks on the type of every expression and function. Being statically-typed, these checks are performed during compilation. These verifications allow various sorts of programming mistakes to be caught on earlier stages of software development, where they are easier to be found and handled. Therefore, knowing how to work well with types in Haskell is critical to develop good programming practices.

This chapter introduces Haskell's type system. It begins by presenting the language's main elementary and composite types (tuples and lists). Then, it presents typeclasses, which group several types, and type variables. Finally, it describes how the types of functions can be defined in the source code, a good practice that will be followed during the remainder of this book.

2.1 Elementary types

In Haskell, the $e :: T$ notation is used to denote that expression e has type T.

The type of any expression or function can be checked in GHCI using the `:type` command (or `:t` , for short). Examples:

```
Prelude> :type True
True :: Bool
Prelude> :type 'a'
'a' :: Char
Prelude> :type "a"
"a" :: [Char]
Prelude> :type (+)
(+) :: Num a => a -> a -> a
Prelude> :type odd
odd :: Integral a => a -> Bool
Prelude> :type head
head :: [a] -> a
```

Table 2.1 presents some of Haskell's most common basic types (i.e. non-composite).

Table 2.1 Most common elementary types in Haskell

Type	Comment
Bool	Boolean. Two values: True and False.
Char	Character. Denoted by single quotes.
Int	Integer. Fixed-precision (i.e. has a maximum size) depending on the system's architecture.
Integer	Integer. Arbitrary precision.
Float	Floating-point. Single precision (32 bits).
Double	Floating-point. Double precision (64 bits).

The `Int` type has the advantage over `Integral` of being more efficient in terms of time and space, since the variable's size is fixed. However, if not handled properly the former may lead to silent overflow issues (where the value stored in a variable exceeds the maximum value that is representable by that data type).

2.2 Tuples

A tuple is a sequence of elements with a fixed size. The elements do not have to be all of the same type. They are denoted by parenthesis.

Tuples with one element do not exist as they have the type of the actual element. There is a single type for a tuple with zero elements, the unit type `()`.

The lines below present examples of the types of some tuples:

```
Prelude> :type (True,'a')
(True,'a') :: (Bool, Char)
Prelude> :type ('x',(True,False))
('x',(True,False)) :: (Char, (Bool, Bool))
```

Table 2.2 presents some of the functions in Prelude to work with tuples.

Table 2.2 Some Prelude functions for tuples

Function	Comment	Example
fst	Returns the first element of a pair (binary tuple).	`fst (3,8)-> 3`
snd	Returns the second element of a pair (binary tuple).	`snd (3,8)-> 8`

Sample exercises

Sample exercise FT-1

Problem statement

Implement myFst, which, similarly to Prelude's fst, given a pair (binary tuple), returns its first element.

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

Usage examples:

```
*Main> myFst (3,8)
3
*Main> myFst ((3,8),8)
(3,8)
*Main> myFst (myFst ((3,4),8))
3
```

Solution

```
myFst (x,_) = x
```

This solution uses a pattern to retrieve an element of the tuple. This sort of pattern can be used with tuples of any size. The underscore is used on the second element of the tuple as a wildcard: it matches anything but does not define a new variable.

Practice exercises

Practice exercise FT-2

Problem statement

Implement mySnd, which, similarly to Prelude's snd, given a pair (binary tuple), returns its second element.

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

Usage examples:

```
*Main> mySnd (3,8)
8
```

Practice exercise FT-3

Problem statement

Implement mySwap, which, similarly to Data.List's swap, given a pair (binary tuple), swaps its elements.

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

Usage examples:

```
*Main> mySwap (3,8)
(8,3)
```

Practice exercise FT-4

Problem statement

a) Implement distance2, which, given two pairs of coordinates representing 2D points, computes the Euclidean distance between them, using the formula:

$$D_{\text{Euclidean}}(A, B) := \sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}$$

```
distance2 :: Floating a => (a, a) -> (a, a) -> a
```

b) Implement distanceInf, which, given two pairs of coordinates representing 2D points, computes the Chebyshev distance between them, using the formula:

$$D_{\text{Chebyshev}}(A, B) := \max(|A_x - B_x|, |A_y - B_y|).$$

```
distanceInf :: (Num a, Ord a) => (a, a) -> (a, a) -> a
```

Usage examples:

```
*Main> distance2 (0,0) (4,5)
6.4031242374328485
*Main> distance2 (1,1) (4,5)
5.0
*Main> distance2 (1,1.5) (4,5.5)
5.0
*Main> distanceInf (0,0) (4,5)
5
*Main> distanceInf (1,1) (4,5)
4
*Main> distanceInf (1,1.5) (4,5.5)
4.0
```

Solutions to the practice exercises

Practice exercise FT-2

Solution

```
mySnd (_,x) = x
```

Practice exercise FT-3

Solution

```
mySwap (x,y) = (y,x)
```

Practice exercise FT-4

Solution

a)

```
distance2 (x1,y1) (x2,y2) = sqrt((x1-x2)^2 + (y1-y2)^2)
```

b)

```
distanceInf (x1,y1) (x2,y2) = max (abs(x1-x2)) (abs(y1-y2))
```

2.3 Lists

A list is a variable-sized sequence of elements of the same type. It has several key differences from tuples:

- Lists have a variable size, which means that a function that receives a list as input can handle lists with 2 or 1000 elements.
- They are homogeneous: all of the elements in a list must have the same type.
- A list can have one element (singleton list).

Empty lists are represented as `[]`. A String is a particular case of a list. It corresponds to an array of characters, `[Char]`.

Lists can be defined in various ways:

- Using square-brackets.
  Example: `[1,4,7,10,13]`. This is used to represent lists of fixed size.
- Using the "cons" operator (`:`).
  Example: `1:4:7:10:13:[]`. This is used in non-empty lists to separate a list's head (first element) from its tail (a list with the remaining elements, which can be `[]`). It is very useful when defining recursive functions to handle lists, with patterns to distinguish the empty list and non-empty lists, where the head is used in some computations and the tail is passed as argument on the recursive call.
- Using ranges.
  Example: `[1,4..13]`. This allows one to define large lists (possibly even infinite lists) in a compact manner. Ranges will be further presented in section 3.1.

- Using list comprehensions.
  Example: `[x | x <- [1..15], mod x 3 == 1]`. This also allows one to define arbitrarily large lists. List comprehensions will be further presented in section 3.3.

The lines below present examples of the types of some lists:

```
Prelude> :type "abc"
"abc" :: [Char]
Prelude> :type [True,True,False]
[True,True,False] :: [Bool]
Prelude> :type [[True],[True],[False]]
[[True],[True],[False]] :: [[Bool]]
Prelude> :type [("ab",True),("c",False)]
[("ab",True),("c",False)] :: [([Char], Bool)]
```

Table 2.3 presents some of the functions in Prelude to work with lists.

Table 2.3 Some Prelude functions for lists

Function	Comment	Example
`(++)`	Appends two lists.	`[1,2] ++ [3,4] -> [1,2,3,4]`
`head`	Extracts the first element on non-empty lists.	`head [1,2,3] -> 1`
`tail`	Removes the first element on non-empty lists (output can be `[]` if input is a singleton).	`tail [1,2,3] -> [2,3]`
`last`	Extracts the last element on non-empty lists.	`last [1,2,3] -> 3`
`init`	Removes the last element on non-empty lists.	`init [1,2,3] -> [1,2]`
`elem`	Checks if a value is contained in a list.	`elem 3 [1,2,3] -> True`
`(!!)`	Returns the n-th element of a list (with indices starting at 0).	`[1,2,3] !! 1 -> 2`
`length`	Returns the number of elements of a list.	`length [1,2,3] -> 3`
`reverse`	Inverts the order of the elements in a list.	`reverse [1,2,3] -> [3,2,1]`
`take`	Extracts the first n elements of a list.	`take 2 [1,2,3] -> [1,2]`
`drop`	Removes the first n elements of a list.	`drop 2 [1,2,3] -> [3]`
`repeat`	Creates an infinite list with x as the value of all elements.	`repeat 1 -> [1,1,1,1,1,1...]`
`cycle`	Creates an infinite repetition of a list.	`cycle [1,2,3] -> [1,2,3,1,2,3...]`
`zip`	Creates a list with the corresponding pairs of two lists.	`zip [1,2,3] "abc"-> [(1,'a'),(2,'b'),(3,'c')]`
`sum`	Adds all of the numbers in the list.	`sum [1,2,3,4] -> 10`
`product`	Multiplies all of the numbers in the list.	`product [1,2,3,4] -> 24`

There is also a module with more useful functions to work with lists, Data.List. To import it in the command line or use it in a source code file, one can use the line:

```
import Data.List
```

Sample exercises

Sample exercise FT-5

Problem statement

Indicate the result of the following expressions or "error" if the expression is invalid:

a) `1:[2]`
b) `1:2:[]`
c) `1:2`
d) `[1,2,3] ++ [] ++ [[4]]`
e) `[1,2,3] ++ [] ++ [4]`
f) `head (tail [1,2,3])`
g) `"abcdef"!! 3`
h) `length (init "abcd")`
i) `reverse ([1]:[[2]])`
j) `take 3 (repeat 'a')`
k) `cycle "abcd"`
l) `zip (repeat 'a')[1,2,3]`

Solution

a) `[1,2]`
b) `[1,2]`
The expression is valid since the cons operator is right-associative. The parentheses are not necessary.
c) error
The right operand of cons must always be a list.
d) error
The expression is invalid since the first operand is a list of integers and the last one is a list of lists of integers.
e) `[1,2,3,4]`
The expression is valid since all the operands are lists of integers.
f) `2`
This expression retrieves the second element of the list.
g) `'d'`
h) `3`
i) `[[2],[1]]`
j) `"aaa"`
take converts infinite lists into finite ones.
k) `"abcdabcdabcd..."`
l) `[('a',1),('a',2),('a',3)]`
zip also works when the input lists have different lengths. The output will have the length of the shortest list.

Sample exercise FT-6

Problem statement

a) Implement myHead, which, similarly to Prelude's head, returns the first element of a list.

```
myHead :: [a] -> a
```

b) Implement hasLengthTwo, which returns a boolean indicating whether the input list has exactly two elements.

```
hasLengthTwo :: [a] -> Bool
```

Usage examples:

```
*Main> myHead [1,2,3,4]
1
*Main> hasLengthTwo [1]
False
*Main> hasLengthTwo [1,2]
True
```

Solution

a)

```
myHead (h:_) = h
myHead [] = error "Empty list"
```

This solution presents a basic example of pattern-matching with lists. To retrieve a list's first element, a pattern with the cons operator (:) is used.

The second definition uses the empty list pattern. As an alternative, an underscore could have been used for the input argument.

One very common error when using the cons pattern is forgetting the parentheses, which are not optional.

b)

```
-- Alternative with a fixed-size list pattern
hasLengthTwo [_,_] = True
hasLengthTwo _ = False

-- Alternative with the cons operator in the pattern
hasLengthTwo' (_:_:[]) = True
hasLengthTwo' _ = False
```

The first alternative illustrates a second kind of pattern that can be used in lists of fixed size.

The second alternative presents a slightly more complex use of cons operator patterns.

Sample exercise FT-7

Problem statement

Implement myLength, which, similarly to Prelude's length, returns the number of elements in a list.

```
myLength :: Num b => [a] -> b
```

Usage examples:

```
*Main> myLength [1,2,-1,3]
4
*Main> myLength "abc"
3
*Main> myLength ""
0
```

Solution

```
myLength [] = 0
myLength (_:xs) = 1 + myLength xs
```

This solution defines a typical strategy to implement functions with lists: a base case is defined for the empty list, and a recursive case, for lists with at least one element. The recursive case performs some sort of computation and contains a recursive call where the list's tail is passed as argument.

The underscore is used on the second definition since the value of the head of a list is not useful for the computation of its length.

Sample exercise FT-8

Problem statement

Implement myMinimum, which, similarly to Prelude's minimum, returns the smallest element of a list.

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

Usage examples:

```
*Main> myMinimum [1,2,-1,3]
-1
*Main> myMinimum "abc"
'a'
```

Solution

```
myMinimum (x:xs) = myMinimumAux xs x
myMinimum [] = error "Empty list"

myMinimumAux [] m = m
myMinimumAux (x:xs) m
  | x < m = myMinimumAux xs x
  | otherwise = myMinimumAux xs m
```

The solution uses an auxiliary method which has an argument to store the minimum element found at a given point. This argument is initialized with the

first element of the list, which is not passed to the first argument of the auxiliary function as it has already been processed.

The function will not work on empty lists since `[]` does not match with the pattern of the main definition of myMinimum. An additional (optional) definition was used to provide a user-friendly error message when the empty list is given as input.

Practice exercises

Practice exercise FT-9

Problem statement

Indicate the result of the following expressions or "error" if the expression is invalid:

a) `reverse []`
b) `[[1,2]]++[[]]++[[3],[4,5]]`
c) `[1,2]:[]:[3]:[[4,5]]`
d) `([1,2]:[]:[3]:[[4,5]])!! 3`
e) `length ([]:[]:[]:[])`
f) `take 2 ([1,2]:[3,4,5]:[6,7]:[8])`
g) `take 2 ([1,2]:[3,4,5]:[6,7]:[])`
h) `[]:[]:[] ++ []:[]`
i) `"abc":[[]] ++ "dce":[]`
j) `tail ([1]:[]:[2]:[3]:[])`
k) `[[1,2,3,4],[5,6,7,8],[9,10,11,12]] !! 2 !! 3`
l) `[5,6,7,8] ‘elem‘ [[1,2,3,4],[5,6,7,8],[9,10,11,12]]`

Practice exercise FT-10

Problem statement

Consider the definition of function f below:

```
f :: [a] -> (a,[a])
f (_:_:x:y) = (x,y)
```

a) Explain concisely what function f computes.

b) Implement function f using functions from the Prelude, and without using pattern matching nor recursion.

Practice exercise FT-11

Problem statement

Consider evaluateLength, which returns a string judging the input list's length according to the following rules:

- A list with 0 or 1 elements is considered "short".
- A list with 2 or 3 elements is considered "medium-sized".
- A list with 4 or more elements is considered "long".

a) Implement evaluateLength using Prelude's length function and without patterns.

```
evaluateLength :: [a] -> String
```

b) Implement evaluateLength using pattern matching instead. Use as few definitions as possible.

Usage examples:

```
*Main> evaluateLength "abc"
"Medium-sized"
*Main> evaluateLength "abcde"
"Long"
*Main> evaluateLength ""
"Short"
*Main> evaluateLength [1,2]
"Medium-sized"
```

Practice exercise FT-12

Problem statement

a) Implement mySum, which, similarly to Prelude's sum, compute the sum of the elements in a list of integers.

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

b) Implement myProduct, which, similarly to Prelude's product, compute the product of the elements in a list of integers.

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

Usage examples:

```
*Main> mySum [1,2,3,4]
10
*Main> myProduct [1,2,3,4]
24
```

Practice exercise FT-13

Problem statement

a) Implement myAnd, which, similarly to Prelude's and, computes the logical conjunction of a list of booleans.

```
myAnd :: [Bool] -> Bool
```

b) Implement myOr, which, similarly to Prelude's or, computes the logical disjunction of a list of booleans.

```
myOr :: [Bool] -> Bool
```

Usage examples:

```
*Main> myAnd [True, False]
False
*Main> myOr [False, True, False]
True
*Main> myOr [length [] == 0, 5 > 3]
True
```

Practice exercise FT-14

Problem statement

Implement the scalarProduct, which computes the scalar product between two lists X and Y with the same length n, using the formula:

$$X \cdot Y = \sum_{k=1}^{n} x_i * y_i$$

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

Usage examples:

```
*Main> scalarProduct [1,-1,5] [4,3,4]
21
```

Practice exercise FT-15

Problem statement

Implement myMaximum, which, similarly to Prelude's minimum, given a list, returns the largest element of a list.

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

Usage examples:

```
*Main> myMaximum [1,2,-1,3,1]
3
```

```
*Main> myMaximum "abc"
'c'
```

Practice exercise FT-16

Problem statement

Implement myElem, which, similarly to Prelude's elem, returns a boolean indicating if an element belongs to a list.

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

Usage examples:

```
*Main> myElem 'e' "4drwe897y"
True
*Main> myElem 'k' "4drwe897y"
False
```

Practice exercise FT-17

Problem statement

Implement nth1, which, given an integer n and a list l, returns the n-th element of l (with indices starting at 1).

If the index is out of range, an error with the text "Index out of range" should be issued.

Note: This function is slightly different to Prelude's indexation operator (!!), which considers indices starting at 0.

```
nth1 :: (Eq a, Num a) => a -> [b] -> b
```

Usage examples:

```
*Main> nth1 3 "abcd"
'c'
*Main> nth1 5 "abcd"
*** Exception: Index out of range
CallStack (from HasCallStack):
error, called at nth1.hs:1:13 in main:Main
```

Practice exercise FT-18

Problem statement

a) Implement seq22, which, given an integer n, creates a sequence of length n with the structure [1,2,2,...,2,1] (1 followed by a series of 2's and ending with a 1).

```
seq22 :: Num a => Int -> [a]
```

b) Implement seq42, which, given an integer n, creates a sequence of length n with the structure [1,4,2,4,2,...,4,2,1] (1 followed by a series of 4 and 2's and ending with a 1).

```
seq42 :: Num a => Int -> [a]
```

Usage examples:

```
*Main> seq22 6
[1,2,2,2,2,1]
*Main> seq22 9
[1,2,2,2,2,2,2,2,1]
*Main> seq42 6
[1,4,2,4,2,1]
*Main> seq42 9
[1,4,2,4,2,4,2,4,1]
```

Practice exercise FT-19

Problem statement

Consider the definition of function f below:

```
f :: (Ord a) => [a] -> a
f [] = error "Empty list"
f [_] = error "Singleton"
f (x:y:zs)
    | (x > y) = g zs x y
    | otherwise = g zs y x

g :: (Ord a) => [a] -> a -> a -> a
g [] _ y = y
g (z:zs) x y
  | (z >= x) = g zs z x
  | (z >= y) = g zs x z
  | otherwise = g zs x y
```

Explain concisely what function f computes.

Solutions to the practice exercises

Practice exercise FT-9

Solution

a) `[]`

b) `[[1,2],[],[3],[4,5]]`

c) `[[1,2],[],[3],[4,5]]`

The expression is equivalent to the previous one.

d) `[4,5]`

This expression is tricky. The result from the previous line should help reaching the correct result.

e) `3`

By reading the expression from right to left, one can see that the rightmost empty list is not converted to an element of the output's list.

f) error

The expression is invalid since the rightmost operand indicates the list is composed of integers, while the other operands suggest the output is a list of lists of integers.

g) `[[1,2],[3,4,5]]`

h) `[[],[],[]]`

i) `["abc","","dce"]`

j) `[[],[2],[3]]`

k) `12`

l) `True`

Practice exercise FT-10

Solution

a) Function f returns a pair with the third element of the input list l and the sublist of l starting at the fourth element.

b)

```
f' :: [a] -> (a,[a])
f' l = (l !! 2, drop 3 l)
```

For the first element of the pair, `2` is passed to `(!!)` rather than `3` since this function considers that list indices start at 0.

Practice exercise FT-11

Solution

a)

```
evaluateLength l
    | len <= 1 = "Short"
    | len <= 3 = "Medium-sized"
    | otherwise = "Long"
    where len = length l
```

A `where` block is used to shorten the code.

b)

```
evaluateLength' [] = "Short"
```

```
evaluateLength' [_] = "Short"
evaluateLength' [_,_] = "Medium-sized"
evaluateLength' [_,_,_] = "Medium-sized"
evaluateLength' _ = "Long"
```

A pattern is needed for each list length between 0 and 3. The final, "catch-all" pattern is used to cover all the other cases.

Practice exercise FT-12

Solution

a)

```
mySum [] = 0
mySum (x:xs) = x + (mySum xs)
```

b)

```
myProduct [] = 1
myProduct (x:xs) = x * (myProduct xs)
```

The neutral element of multiplication is 1, since multiplying any number n by 1 results in n. Therefore, the base case of product (with the empty list) returns 1.

Practice exercise FT-13

Solution

a)

```
myAnd [] = True
myAnd (x:xs) = x && myAnd xs
```

The neutral element for the logical conjunction is `True`, which corresponds to the result for the base case.

b)

```
myOr [] = False
myOr (x:xs) = x || myOr xs
```

The neutral element for the logical disjunction is `False`.

Practice exercise FT-14

Solution

```
scalarProduct [] [] = 0
scalarProduct (x:xs) (y:ys) = x*y + (scalarProduct xs ys)
```

Practice exercise FT-15

Solution

```
myMaximum (x:xs) = myMaximumAux xs x
myMaximum [] = error "Empty list"

myMaximumAux [] m = m
myMaximumAux (x:xs) m
  | x > m = myMaximumAux xs x
  | otherwise = myMaximumAux xs m
```

This function is very similar to the one from exercise FT-8.

Practice exercise FT-16

Solution

```
myElem _ [] = False
myElem y (x:xs)
    | (x==y) = True
    | otherwise = myElem y xs
```

This solution uses patterns with guards.

Practice exercise FT-17

Solution

```
nth1 _ [] = error "Index out of range"
nth1 1 (x:_) = x
nth1 n (_:xs) = nth1 (n-1) xs
```

Practice exercise FT-18

Solution

a)

```
seq22 n = 1:(take (n-2) (repeat 2)) ++ [1]
```

The subsequence with variable length is created using repeat to create an infinite list of 2's, to which we only take the first (n-2) elements.

b)

```
seq42 n = 1:(take (n-2) (cycle [4,2])) ++ [1]
```

The subsequence with variable length is created using cycle to create an infinite list of 4-2's, to which we only take the first (n-2) elements.

Practice exercise FT-19

Solution

Function f computes the second largest element of a list by scanning it only once.

Additional info:
An auxiliary function g contains two extra arguments to store the largest and second largest elements found so far. In the base case of function g, the second largest element (i.e. the third argument) is returned. The recursive cases of g and the definition of f use guards to ensure g's extra arguments are always correctly updated.

2.4 Typeclasses

Certain functions operate over a certain group of types but not over every single type. To allow this behavior, Haskell defines typeclasses which group a set of types by a common property. Types are instances of typeclasses, just like classes are implementations of interfaces in object-oriented programming languages (like Java).

Table 2.4 presents some of the most common typeclasses.

Table 2.4 Some relevant typeclasses

Typeclass	Comment
Num	Numeric types.
Integral	Integer types.
Fractional	Floating-point types. Supports real number division with `(/)`.
Floating	Floating-point types. Includes a type for complex numbers, Complex. Defines certain functions with irrational numbers, such as `sqrt`, `log`, `sin`, `asin` and `sinh`.
RealFloat	Another floating-point typeclass that does not include complex numbers.
Eq	Types for which the equality and inequality operators (`==` , `/=`) are defined.
Ord	Types for which the comparison operators (`>` , `<` , `>=` , `<=`) are defined.
Enum	Types that can be enumerated.

Elements of the Enum class implement the successor and predecessor functions, succ and pred:

```
Prelude> succ 'a'
'b'
Prelude> pred 'Z'
'Y'
Prelude> succ False
True
Prelude> succ 42
43
```

```
Prelude> succ (-42)
-41
```

Certain typeclasses from Table 2.4 inherit properties of other typeclasses. For example, the equality operator is defined for numbers, so `Num` is a derived typeclass of `Eq`.

Figure 2.1 presents a simplified hierarchy of the most common typeclasses and types in Haskell. It should be noted that this figure presents a very simplified view of typeclasses in Haskell. For example, there is a typeclass between `Num` and `Integral` which was omitted for clarity purposes.

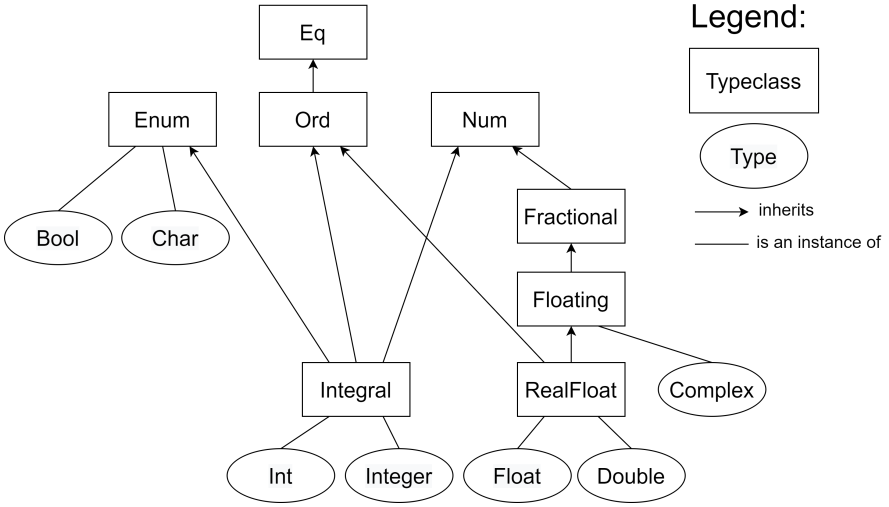


Fig. 2.1 Hierarchy of some relevant typeclasses and types

2.5 Type variables

What is the type of 1? One could say that it is an Int, but it could also be an Integer. One could also want to use 1 as an argument to a trigonometric function for example, which works on floating-point numbers.

When documenting the type of a variable, instead of making the commitment of assigning a variable to a certain type, one could instead associate a variable to a typeclass. This can be achieved using the notation:

```
e :: TC a =>a .
```

This line denotes that e is of type a, which is an instance of typeclass TC. a is a *type variable*: e belongs to any data type a that is an instance of typeclass TC. a is intentionally left undefined. The arrow `=>` denotes a *class constraint*.

Type variables can also be used without an associated type class. For example, the empty list `[]` can contain any type of variable, so one can document it using a type variable: `[] :: [a]` .

When only partial information about a variable is provided, one should document it with the most general type, to avoid unnecessary assumptions.

When asked about the type of numbers (using the `:type` command), GHCI typically responds with a type variable to link it to a type class (usually `Num`) rather than a specific type:

```
Prelude> :type 1
1 :: Num p => p
Prelude> :type sin 1
sin 1 :: Floating a => a
Prelude> :type (True,3.0)
(True,3.0) :: Fractional b => (Bool, b)
Prelude> :type [1,2]
[1,2] :: Num a => [a]
Prelude> :type (1.4,1)
(1.4,1) :: (Fractional a, Num b) => (a, b)
```

Sample exercises

Sample exercise FT-20

Problem statement

a) Write a valid type declaration for the following expressions without using type variables.

i) `['a','b','c']`
ii) `"abc"`
iii) `42`
iv) `42.5`
v) `[(False,1),(True,3.5)]`
vi) `reverse [(False,1),(True,3.5)]`
vii) `([1],[1.5,2.5])`
viii) `tail [1]`

b) Write the most general type declaration for the previous expressions.

Solution

a) To test if the answers to these questions are correct, one can write type declarations on GHCI. If they are incorrect, an error is issued.

i) `['a','b','c'] :: [Char]`
ii) `"abc":: [Char]`
iii) `42 :: Integer`
iv) `42.5 :: Double`

v) `[(False,1),(True,3.5)] :: [(Bool,Float)]`

vi) `reverse [(False,1),(True,3.5)] :: [(Bool,Float)]`

vii) `([1],[1.5,2.5]):: ([Int],[Float])`

Another valid result would be `([Float],[Float])`, but `([Int],[Int])` would be incorrect.

viii) `tail [1] :: [Int]`

b) These answers correspond to what is returned by the `:type` command.

i) `['a','b','c'] :: [Char]`

In this example, `[Char]` is already the most general type. `Enum a =>[a]` would be incorrect, as that would indicate that 'a' is a Bool.

ii) `"abc":: [Char]`

iii) `42 :: Num a =>a`

iv) `42.5 :: Fractional a =>a`

v) `[(False,1),(True,3.5)] :: Fractional b =>[(Bool, b)]`

vi) `reverse [(False,1),(True,3.5)] :: Fractional b =>[(Bool, b)]`

vii) `([1],[1.5,2.5]):: (Num a, Fractional b)=>([a], [b])`

viii) `tail [1] :: Num a =>[a]`

Despite `tail [1]` returning the empty list, the output of tail has the same type as its input. Therefore, the output list can only contain elements compatible with 1.

Practice exercises

Practice exercise FT-21

Problem statement

a) Write a valid type declaration for the following expressions without using type variables.

i) `zip [1,2] "abc"`

ii) `[[1],[2]]`

iii) `[succ 'a']`

iv) `[1,2,3,4,5.5]`

v) `[1,2] == [1,2]`

vi) `zip (zip "abc""abc")"abc"`

b) Write the most general type declaration for the previous expressions.

Solutions to the practice exercises

Practice exercise FT-21

Solution

a)

i) `zip [1,2] "abc":: [(Integer, Char)]`

ii) `[[1],[2]] :: [[Int]]`

iii) `[succ 'a'] :: [Char]`

iv) `[1,2,3,4,5.5] :: [Float]`

v) `[1,2] == [1,2] :: Bool`

vi) `zip (zip "abc""abc")"abc":: [((Char,Char),Char)]`

b)

i) `zip [1,2] "abc":: Num a =>[(a, Char)]`

ii) `[[1],[2]] :: Num a =>[[a]]`

iii) `[succ 'a'] :: [Char]`

iv) `[1,2,3,4,5.5] :: Fractional a =>[a]`

v) `[1,2] == [1,2] :: Bool`

vi) `zip (zip "abc""abc")"abc":: [((Char,Char),Char)]`

2.6 Functional types

Functions also have a type. It is good practice to declare a function's type by writing its type declaration right above its definition in a source file. These declarations have several advantages:

- It helps the programmer to reason about the function they are writing, reducing the likelihood and amount of programming errors.
- It provides useful documentation for the function. So that it is better understood by those who will use it.
- When the function is used with the wrong type of arguments, it helps to provide more clear error messages.

It should be noted however that Haskell compilers are capable of performing *type inference*, as they can deduce the type of most Haskell expressions and function.

If a function f has n arguments a_i with types `ti` and an output with type `T`, then its type declaration is:

```
f :: t1 -> t2 -> ... -> tn -> T .
```

If the type declaration contains any class constraints, then they are written before the f.

It is interesting to notice that a variable is simply a function without any arguments. Therefore, the type declaration in the last paragraph is a generalization of the `e :: T` expression presented in the beginning of the chapter.

A function whose type declaration contains type variables is known as a *polymorphic function*.

Sample exercises

Sample exercise FT-22

Problem statement

Write the most general type declaration for the following functions:

a) `(+)`

b) `(**)`

c) `[(+),(/)]`

d) `[(+),(**)]`

e) `length` Note: The output type of length is actually Int, rather than any type with typeclass Integral.

f) `fst`

g) `tail`

h) `f` , where `f x y = x /= y`

Solution

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

The "name" of a function that is usually used as an operator must be enclosed with parentheses.

b) `(**) :: Floating a =>a -> a -> a`

The (**) operator is used for exponentiation with floating-point numbers, as presented in chapter 1.

c) `[(+),(/)] :: Floating a =>[a -> a -> a]`

This question is very interesting: in a list, all elements must have the same type, so lists of functions are only valid if all their functions can be converted to the same type. In this case, the conversion is possible since Fractional derives Num. By converting a's typeclass in `(+)` to a Fractional, both functions have matching types.

When trying to find out the most general type that two or more functions share, it is useful to first define the most general type of each function individually.

d) `[(+),(**)] :: Floating a =>[a -> a -> a]`

This reasoning for this expression is similar to the previous one. The `(**)` forces the inputs/output of both function to belong to the class Floating.

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

The `:type` command actually returns `length :: Foldable t =>t a -> Int` . The Foldable typeclass was not presented in this chapter. It corresponds to data structures (such as lists) where folding functions (section 4.6) may be called upon. The declaration using a list is acceptable here.

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

g) `tail :: [a] -> [a]`

h) `f :: Eq a =>a -> a -> Bool`

Both arguments must belong to a type which is an instance of typeclass Eq since they are tested for (in)equality.

Practice exercises

Practice exercise FT-23

Problem statement

Write the most general type declaration for the following functions:

a) `mod`

b) `snd`

c) `[head,length]`

d) `drop`

Note: One of the input arguments is an Int.

e) `(!!)`

Note: One of the input arguments is an Int.

f) `zip`

g) `f1` , where:

```
f1 [] _ = 0
f1 (x:xs) n = n*sin(x) + f1 xs n
```

h) `[f2,f3]` , where:

```
f2 [] = True
f2 [x] = True
f2 (x1:x2:xs) = x1 <= x2 && f2 (x2:xs)

f3 [] = True
f3 [x] = True
f3 (x1:x2:xs) = x1 /= x2 && f3 (x2:xs)
```

i) `f4` , where:

```
f4 x y = f4 y x
```

j) `f5` , where:

```
f5 x y z = f5 z y x
```

k) `[f4,zip]`

Practice exercise FT-24

Problem statement

Write a function with type `(a,b,a)-> (b,a,b)` .

Practice exercise FT-25

Problem statement

When implementing a function to compute the average value of a list, a programmer tried to define it as follows: `average l = sum l / length l`

a) Explain briefly why this definition produces a type error.

b) Write a correct definition of `average` , including an appropriate type declaration for the function.

Suggestion: use `fromIntegral` , which returns a floating-point equivalent of an integer:

```
fromIntegral :: (Integral a, Num b)=> a -> b
```

Solutions to the practice exercises

Practice exercise FT-23

Solution

a) `mod :: Integral a =>a -> a -> a`

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

c) `[head, length] :: [[Int] -> Int]`

d) `drop :: Int -> [a] -> [a]`

e) `(!!):: [a] -> Int -> a`

The Int argument is the second one, and not the first like in drop. Knowledge on the order of the arguments comes with practice.

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

g) `f1 :: (Floating a)=>[a] -> a -> a`

h) `[f2,f3] :: (Ord a)=>[[a] -> Bool]`

Ord is more specific than Eq, so there is no need to define the type as: `[f2,f3] :: (Ord a, Eq a)=>[[a] -> Bool] .`

i) `f4 :: a -> a -> b`

Both arguments must be of the same type, since they are switched in the recursive call.

j) `f5 :: a -> b -> a -> c`

k) `[f4,zip] :: [[a] -> [a] -> [(a, a)]]`

Practice exercise FT-24

Solution

```
-- Solution with the two clauses
f (x,y,_) = (y,x,y)
f (_,y,x) = (y,x,y)
-- Almost correct solution in a single clause, enforces a class constraint
f' (x,y,z) = (y,x+z,y)
```

The first solution yields a warning in GHCI stating that the clause is redundant, since the first clause of f catches all the possible cases that the second clauses would catch.

The second solution does not produce any warning, but forces the function to have a class constraint: `f'' :: Num a =>(a, b, a)-> (b, a, b) .`

Practice exercise FT-25

Solution

a) The definition produces a type error since `length` returns an Int, while `(/)` expects both its arguments to have a type which is an instance of typeclass `Fractional` . This is not possible since `Int` is not an instance of this typeclass.

b)

```
average :: (Fractional a) => [a] -> a
average l = sum l / fromIntegral(length l)
```

Using `fromIntegral` on the denominator, both operands of `(/)` can be interpreted as real numbers.

An alternative correct solution is to call a user-defined version of `length` that returns a value whose type is an instance of typeclass `Num` , as presented in exercise FT-7.

Chapter 3
Lists

The previous chapter provided an overview of types in Haskell and provided a first glimpse on lists. This chapter is dedicated to lists since they are an essential data structure that can store data of arbitrary size. Moreover, Haskell provides a very rich array of means to build them.

This chapter aims to deepen the reader's knowledge on lists by presenting three ways to build them: using ranges, recursion and list comprehensions.

Note: since functional types were introduced in the previous chapter, starting from this chapter, rather than providing a function's signature in the problem statement, the reader must include them in all of the exercise answers that involve defining a function.

3.1 Lists by range

Lists can be defined using ranges, with one of the following formats:

- `[<value1>, <value2> .. <valueN>]` : produces a list with all the values starting at `<initial value>`, where each element is the sum of the previous element and `<value2> - <value1>`, until a value greater than `<valueN>` is reached (which is not included in the resulting list). `<valueN>` may not be included in the resulting list, depending on the step value. The step can be negative (in this case, we should have `<value1> > <valueN>`). If the step has the opposite sign of `<valueN> - <value1>`, the empty list is returned.
- `[<value1> .. <final value>]` : same as above, but assumes `<value2> = <value1> + 1`.
- `[<value1>, <value2> ..]` : produces an infinite list starting at `<initial value>`, with a step of `<value2> - <value1>`.
- `[<value1>, ..]` : same as above, but assumes `<value2> = <value1> + 1`.

Here are some examples with ranges:

```
Prelude> [1..]
```

```
[1,2,3,4,5,6,7,8,9,10...]
Prelude> [1..5]
[1,2,3,4,5]
Prelude> [1,3 .. 10]
[1,3,5,7,9]
Prelude> [1,0..]
[1,0,-1,-2,-3,-4,-5,-6,-7,-8...]
Prelude> [1,3 ..]
[1,3,5,7,9,11,13,15,17,19...]
Prelude> ['a'..]
"abcdefghijklmnopqrstuvwxyz{|}~\DEL\128\129..."
Prelude> [0.1, 0.2 .. 1]
[0.1,0.2,0.30000000000000004,0.4,0.5,0.6,0.7000000000000001,0.8,0.9,1.0]
```

As seen in the last example, special care needs to be taken when using ranges with floating-point values due to numeric imprecision.

Elements of a list range must belong to the type which is an instance of typeclass `Enum`:

```
Prelude> f a b = [a .. b]
Prelude> :t f
f :: Enum a => a -> a -> [a]
```

Haskell is able to handle computations with infinite lists due to its *lazy evaluation* mechanic which makes it so that the value of any expression is only computed when it is actually needed. For instance, using the `take` function to remove the first N elements of an infinite list results in a finite computation:

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

Working with infinite lists has the advantage of separating the logic of generating a list from the logic to process it, which allows for certain functions to be implemented more easily and in a more readable manner.

Sample exercises

Sample exercise LI-1

Problem statement

Write the following list ranges by extension (i.e. by indicating each of their elements, in order). If they consist of infinite lists, write the first 10 elements. Use an ellipsis (...) after the tenth element to denote lists with more than 10 elements.

a) `[1,-3 .. -10]`

b) `[1,-3 .. 10]`

c) `['z','y'..]`

d) `cycle [1..5]`

Solution

a) `[1,-3,-7]`
This range has a step of $-3 - 1 = -4$.
b) `[]`
The range has a step of -4 (negative), but $10 - 1 = 9$ is positive.
c) `"zyxwvutsrq..."`
This range has a step of -1.
d) `[1,2,3,4,5,1,2,3,4,5...]`

Practice exercises

Practice exercise LI-2

Problem statement

Write the following list ranges by extension (i.e. by indicating each of their elements, in order). If they consist of infinite lists, write the first 10 elements. Use an ellipsis (...) after the tenth element to denote lists with more than 10 elements.

a) `[5,-2.. -20]`
b) `['a','d'..'z']`
c) `zip [1,2..] [1,3..]`
d) `drop 5 [0,2..]`
e) `cycle (cycle [1..5])`

Solutions to the practice exercises

Practice exercise LI-2

Solution

a) `[5,-2,-9,-16]`
b) `"adgjmpsvy"`
c) `[(1,1),(2,3),(3,5),(4,7),(5,9),(6,11),(7,13),(8,15),(9,17),(10,19)...]`
d) `[10,12,14,16,18,20,22,24,26,28...]`
e) `[1,2,3,4,5,1,2,3,4,5...]`
Calling cycle twice produces the same result as calling it only once.

3.2 Lists by recursion

The previous chapter introduced some examples of recursive functions with lists. This section contains exercises to implement recursive functions with lists that return new lists.

Sample exercises

Sample exercise LI-3

Problem statement

Implement append, which, similarly to Prelude's concatenation operator (++), receives two lists l1 and l2 and outputs a list with the elements of l1 followed by those in l2 (in the same order).

Usage examples:

```
*Main> append [1,2] [3,4]
[1,2,3,4]
```

Solution

```
append :: [a] -> [a] -> [a]
append [] l = l
append (x:xs) l = x:(append xs l)
```

The most common solution for this problem has a linear temporal complexity, $O(n)$, where n is the number of elements in the first argument. It is not worth it to test which of the two list arguments is shorter, since this would require using a linear-time algorithm to compute the lengths.

Sample exercise LI-4

Problem statement

Implement myReverse, which, similarly to Prelude's reverse, given a list, returns a new list with its elements in reverse order.

Usage examples:

```
*Main> myReverse [1,2,3,4,5]
[5,4,3,2,1]
```

Solution

```
-- pure recursive version
myReverse :: [a] -> [a]
myReverse [] = []
myReverse (x:xs) = myReverse xs ++ [x]

-- version with an accumulator (more efficient)
myReverse' :: [a] -> [a]
myReverse' l = myReverse'_aux l []

myReverse'_aux :: [a] -> [a] -> [a]
myReverse'_aux [] l = l
myReverse'_aux (x:xs) acc = myReverse'_aux xs (x:acc)
```

The first solution has a temporal complexity of $O(n^2)$, where n is the length of the list, since a concatenation is performed for each element in the list.

The second solution has a complexity of $O(n)$. This solution resorts to an accumulator, which stores a partial solution of the problem. In this case, the accumulator is a list with the reverse of the input up to a certain element.

Sample exercise LI-5

Problem statement

Implement myRepeat, which, similarly to Prelude's repeat, creates an infinite list where all the items are equal to its argument.

Usage examples:

```
*Main> myRepeat 1
[1,1,1,1,1,1,1,1,1,1...]
```

Solution

```
myRepeat :: a -> [a]
myRepeat x = x:(myRepeat x)
```

Notice how an infinite recursion in this case leads to the intended infinite list, due to lazy evaluation. Infinite recursion is created by intentionally not defining a base case for the function.

Sample exercise LI-6

Problem statement

Implement conseqPairs that receives a list l and returns a list of pairs of consecutive elements in l.

Usage examples:

```
*Main> conseqPairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

Solution

```
conseqPairs :: [a] -> [(a,a)]
conseqPairs [] = []
conseqPairs [x] = []
conseqPairs (x:y:xs) = (x,y):(conseqPairs (y:xs))
```

This function definition has two base cases for lists with 0 and 1 elements. The recursive definition is used for lists with at least two elements.

Practice exercises

Note: in the following exercises, lists should be defined by recursion, rather than by comprehension (which will be introduced in the next section). Do not use any of Prelude's or other modules' functions for list processing, unless it is specifically indicated otherwise.

Practice exercise LI-7

Problem statement

a) Implement deleteOne, which, similarly to Data.List's delete, given a value v and a list l, removes the first occurrence of v in l.

b) Implement deleteAll, which, given a value v and a list l, removes all occurrences of v in l.

Usage examples:

```
*Main> deleteOne 'a' "banana"
"bnana"
*Main> deleteAll 'a' "banana"
"bnn"
```

Practice exercise LI-8

Problem statement

Implement myConcat, which, similarly to Prelude's concat, concatenates all of the sublists of a list of lists.

Usage examples:

```
*Main> myConcat [[1,2],[3,4]]
[1,2,3,4]
*Main> myConcat [[1,2],[],[3,4,5],[6]]
[1,2,3,4,5,6]
```

Practice exercise LI-9

Problem statement

Implement myReplicate, which, similarly to Prelude's replicate, creates a list with the length given by the first argument and the items having the value of the second argument.

Usage examples:

```
*Main> myReplicate 5 1
[1,1,1,1,1]
```

Practice exercise LI-10

Problem statement

Implement myCycle, which, similarly to Prelude's cycle, creates an infinite list where the input list is concatenated to itself an infinite number of times.

Usage examples:

```
*Main> cycle [1,2,3,4,5]
[1,2,3,4,5,1,2,3,4,5...]
```

Practice exercise LI-11

Problem statement

Implement myIntersperse, which, similarly to Data.List's intersperse, takes an element v and a list l and adds v between consecutive elements of l.

Usage examples:

```
*Main> myIntersperse ',' "abcde"
"a,b,c,d,e"
```

Practice exercise LI-12

Problem statement

a) Implement myTake, which, similarly to Prelude's take, given a positive integer n and a list l, returns the first n elements of l. If there are fewer than n elements, then it returns l.

b) Implement myDrop, which, similarly to Prelude's drop, given a positive integer n and a list l, removes the first n elements from l. If there are fewer than n elements, then it returns the empty list.

Usage examples:

```
*Main> myTake 5 [1,3..]
[1,3,5,7,9]
*Main> myDrop 5 [1,3..20]
[11,13,15,17,19]
```

Practice exercise LI-13

Problem statement

Implement mySplitAt, which, similarly to Prelude's splitAt, given an integer n and a list l, returns a pair where the first element contains the first n elements of l and the second one contains the remaining elements of l (in the same order as in l). If there are less than n elements in l, then the pair `(l,[])` is returned.

Usage examples:

```
*Main> splitAt 5 [1..8]
([1,2,3,4,5],[6,7,8])
*Main> splitAt 9 [1..8]
([1,2,3,4,5,6,7,8],[])
```

Practice exercise LI-14

Problem statement

Implement myGroup, which, similarly to Data.List's group, splits its list argument into a list of lists of equal, adjacent elements.

Usage examples:

```
*Main> myGroup [1,1,0,0,0,1,1,2,1,1]
[[1,1],[0,0,0],[1,1],[2],[1,1]]
*Main> myGroup "aaabbcdddaaa"
["aaa","bb","c","ddd","aaa"]
```

Practice exercise LI-15

Problem statement

a) Implement myInits, which, similarly to Data.List's inits, returns the list of prefixes of its argument list, shortest first.

b) Implement myTails, which, similarly to Data.List's tails, returns the list of suffixes of its argument list, shortest last.

Usage examples:

```
*Main> myInits "abc"
["","a","ab","abc"]
*Main> myTails "abc"
```

```
["abc","bc","c",""]
```

Practice exercise LI-16

Problem statement

a) Implement myZip, which, similarly to Prelude's zip, returns a list of pairs containing the elements of both input lists occurring at the same positions. The length of the output is equal to the length of the shortest of the input lists.

b) Implement myZip3, which, similarly to Prelude's zip3, returns a list of triples containing the elements of the three input lists occurring at the same positions. The length of the output is equal to the length of the shortest of the input lists.

Usage examples:

```
*Main> myZip [1,2,3,4] "abc"
[(1,'a'),(2,'b'),(3,'c')]
*Main> myZip3 [1,2,3,4] "abc" [1,0,0,1]
[(1,'a',1),(2,'b',0),(3,'c',0)]
```

Practice exercise LI-17

Problem statement

Implement differentFromNext which receives a list and returns all the elements that are different to the one that comes next in the list.

Usage examples:

```
*Main> differentFromNext "aabbaacca"
"abac"
*Main> differentFromNext "aab"
"a"
*Main> differentFromNext "aaa"
""
```

Practice exercise LI-18

Problem statement

Implement myTranspose which, similarly to List.Data's transpose, returns the transpose of a 2D list, where the columns of the input list are converted to rows in the output list (and vice-versa). If some of the rows are shorter than the other rows, then their elements are skipped.

Usage examples:

```
*Main> myTranspose [[1,2],[3,4],[5,6]]
```

```
[[1,3,5],[2,4,6]]
*Main> myTranspose [[1,2],[3],[],[5,6]]
[[1,3,5],[2,6]]
```

Practice exercise LI-19

Problem statement

Implement myNub which, similarly to List.Data's nub, removes duplicates from the input list by only keeping the first occurrence of each distinct element.

Usage examples:

```
*Main> myNub "aaabbbcdefga"
"abcdefg"
```

Practice exercise LI-20

Problem statement

Implement mySubsequences which, similarly to List.Data's subsequences, returns the list of all subsequences of the argument.

Usage examples:

```
*Main> mySubsequences "abc"
["","a","b","ab","c","ac","bc","abc"]
*Main> mySubsequences []
[[]]
```

Practice exercise LI-21

Problem statement

Implement iSort which sorts a list using the insertion sort algorithm. The list is divided into an unsorted and sorted part, the latter being initially empty. Elements from the unsorted list are then progressively added to the sorted list.

Suggestion: implement an auxiliary function to insert an element in its proper position in a sorted list.

Usage examples:

```
*Main> iSort [5,9,1,4,7,2,5,3]
[1,2,3,4,5,5,7,9]
```

Practice exercise LI-22

Problem statement

Implement sSort which sorts a list using the selection sort algorithm. This method repeatedly finds the smallest element of the unsorted part of the list and appends it to its beginning.

To implement the solution, you are allowed to use Data.List's delete function.

Usage examples:

```
*Main> sSort [5,9,1,4,7,2,5,3]
[1,2,3,4,5,5,7,9]
```

Practice exercise LI-23

Problem statement

Implement mSort which sorts a list using the merge sort algorithm. This method follows a divide-and-conquer strategy where the list is recursively split into two halves which are then sorted. The sorted halves are then merged into a full sorted list.

To implement the solution, you are allowed to use Data.List's splitAt function.

Usage examples:

```
*Main> mSort [5,9,1,4,7,2,5,3]
[1,2,3,4,5,5,7,9]
```

Practice exercise LI-24

Problem statement

Implement qSort which sorts a list using the quicksort algorithm. This method follows a divide-and-conquer strategy where an element is arbitrarily selected to be the pivot (such as the list's first element). The other elements are then added to two lists according to whether they are smaller or larger than the pivot. These two sublists are then recursively sorted.

Suggestion: implement an auxiliary function to partition a list into two sublists when given a pivot and the remainder of the list. The output of this secondary function can be returned as a pair.

Usage examples:

```
*Main> qSort [5,9,1,4,7,2,5,3]
[1,2,3,4,5,5,7,9]
```

Solutions to the practice exercises

Practice exercise LI-7

Solution

a)

```
deleteOne :: Eq a => a -> [a] -> [a]
deleteOne _ [] = []
deleteOne v (x:xs)
    | (v == x) = xs
    | otherwise = x:(deleteOne v xs)
```

This solution uses patterns with guards.

b)

```
deleteAll :: Eq a => a -> [a] -> [a]
deleteAll _ [] = []
deleteAll v (x:xs)
    | (v == x) = deleteAll v xs
    | otherwise = x:(deleteAll v xs)
```

Practice exercise LI-8

Solution

```
myConcat :: [[a]] -> [a]
myConcat [] = []
myConcat (x:xs) = x ++ myConcat xs
```

Practice exercise LI-9

Solution

```
myReplicate :: Int -> a -> [a]
myReplicate 0 _ = []
myReplicate n v
    | n > 0 = v:(myReplicate (n-1) v)
    | otherwise = error "negative␣number␣in␣first␣argument"
```

Practice exercise LI-10

Solution

```
myCycle :: [a] -> [a]
myCycle l = l ++ (myCycle l)
```

This solution is very similar to the one from exercise LI-8, with myConcat. The difference is that there is no base case since the output is an infinite list. In addition, the function's input is the same as the one used in the recursive call.

Practice exercise LI-11

Solution

a)

```
myIntersperse :: a -> [a] -> [a]
myIntersperse _ [] = []
myIntersperse _ [x] = [x]
myIntersperse v (x:xs) = x:v:(myIntersperse v xs)
```

This solution defines two base cases to prevent an extra element equal to v to be added to the end of the list. The case with the empty list is used to allow this function to handle receiving empty lists.

Practice exercise LI-12

Solution

a)

```
myTake :: Int -> [a] -> [a]
myTake 0 _ = []
myTake _ [] = []
myTake n (x:xs)
    | n > 0 = x:(myTake (n-1) xs)
    | otherwise = error "negative␣number␣in␣first␣argument"
```

b)

```
myDrop :: Int -> [a] -> [a]
myDrop 0 l = l
myDrop _ [] = []
myDrop n (_:xs)
    | n > 0 = myDrop (n-1) xs
    | otherwise = error "negative␣number␣in␣first␣argument"
```

Practice exercise LI-13

Solution

```
mySplitAt :: Int -> [a] -> ([a],[a])
mySplitAt 0 l = ([],l)
mySplitAt _ [] = ([],[])
mySplitAt n (x:xs)
    | n > 0 = (let (a,b) = mySplitAt (n-1) xs in (x:a,b))
    | otherwise = error "negative␣number␣in␣first␣argument"
```

In the recursive step, a let expression is used to bind each component of the pair returned by the recursive call in a variable. These variables are then used to construct the new pair to be returned by the function.

Practice exercise LI-14

Solution

```
myGroup :: Eq a => [a] -> [[a]]
myGroup [] = []
myGroup [x] = [[x]]
myGroup (x:y:xs)
    | x == y = (x:g):gs
    | otherwise = [x]:g:gs
    where (g:gs) = myGroup (y:xs)
```

Special care must be taken with the base cases. The output is a 2D list, so the case where the input has a single element is `[[x]]` and not `[x]`. However, for the case with an empty list, the empty list should be returned, rather than `[[]]`. The empty list can represent a list of any dimension.

Practice exercise LI-15

Solution

a)

```
myInits :: [a] -> [[a]]
myInits [] = [[]]
myInits (x:xs) = []:(addHeadToAll x (myInits xs))

addHeadToAll :: a -> [[a]] -> [[a]]
addHeadToAll _ [] = []
addHeadToAll h (l:ls) = (h:l):(addHeadToAll h ls)
```

An auxiliary function is used to add an element to the beginning of each list.

b)

```
myTails :: [a] -> [[a]]
myTails [] = [[]]
myTails (x:xs) = (x:t):t:ts
    where (t:ts) = myTails xs
```

Practice exercise LI-16

Solution

a)

```
myZip :: [a] -> [b] -> [(a,b)]
```

```
myZip [] _ = []
myZip _ [] = []
myZip (x:xs) (y:ys) = (x,y):(myZip xs ys)
```

b)

```
myZip3 :: [a] -> [b] -> [c] -> [(a,b,c)]
myZip3 [] _ _ = []
myZip3 _ [] _ = []
myZip3 _ _ [] = []
myZip3 (x:xs) (y:ys) (z:zs) = (x,y,z):(myZip3 xs ys zs)
```

Practice exercise LI-17

Solution

```
differentFromNext :: Eq a => [a] -> [a]
differentFromNext [] = []
differentFromNext [x] = []
differentFromNext (x:y:xs)
    | (x /= y) = x:(differentFromNext (y:xs))
    | otherwise = differentFromNext (y:xs)
```

Practice exercise LI-18

Solution

```
myTranspose :: [[a]] -> [[a]]
myTranspose [] = []
myTranspose m = hs:(myTranspose ts)
    where (hs,ts) = splitHeadsTails m

splitHeadsTails :: [[a]] -> ([a], [[a]])
splitHeadsTails [] = ([],[])
splitHeadsTails (ys:xs) =
    case ys of [] -> (hs,ts)
               [z] -> (z:hs,ts)
               (z:zs) -> (z:hs,zs:ts)
    where (hs,ts) = splitHeadsTails xs
```

The auxiliary function splitHeadsTails is responsible for removing the first element (head) of each sublist. It returns a a pair where the first component has all the heads, and the second component has all the tails. It handles rows with different lengths by defining two base cases in the case expression, which handles lists with 0 and 1 elements. In these two cases, the list of tails (second component of the value returned by the auxiliary function) is the same as the one returned by the recursive call.

Practice exercise LI-19

Solution

```
myNub :: Eq a => [a] -> [a]
myNub l = myNubAux l []

myNubAux [] _ = []
myNubAux (x:xs) uniques
    | elem x uniques = myNubAux xs uniques
    | otherwise = x:(myNubAux xs (x:uniques))
```

This solution uses an auxiliary function with an extra argument to store each unique value of the input list, sorted by their first occurrence. This solution runs in $O(n^2)$ time, where n is the length of the input list.

Practice exercise LI-20

Solution

```
mySubsequences :: [a] -> [[a]]
mySubsequences [] = [[]]
mySubsequences (x:xs) = addOrNotToHead x (mySubsequences xs)

addOrNotToHead :: a -> [[a]] -> [[a]]
addOrNotToHead h [] = []
addOrNotToHead h (l:ls) = l:(h:l):(addOrNotToHead h ls)
```

The auxiliary function addOrNotToHead receives an element h and list l, and creates a new list where two elements are created for each non-empty suffix s of l: one with h coming before s, and one without h, only s. In the base case, it does not create a list with the sublist, as adding the empty subsequence to the output of the main function is done via the base case of mySubsequences.

Practice exercise LI-21

Solution

```
iSort :: Ord a => [a] -> [a]
iSort [] = []
iSort (x:xs) = insert x (iSort xs)

insert :: Ord a => a -> [a] -> [a]
insert v [] = [v]
insert v (x:xs)
    | v <= x = v:x:xs
    | otherwise = x:(insert v xs)
```

Insertion sort runs in $O(n^2)$ time.

Practice exercise LI-22

Solution

```
import Data.List

sSort :: Ord a => [a] -> [a]
sSort [] = []
sSort l =
    m:(sSort (delete m l))
    where m = minimum l
```

Selection sort runs in $O(n^2)$ time.

Practice exercise LI-23

Solution

```
import Data.List

mSort :: Ord a => [a] -> [a]
mSort [] = []
mSort [x] = [x]
mSort l =
   merge (mSort l1) (mSort l2)
   where (l1,l2) = splitAt (div (length l) 2) l

merge :: Ord a => [a] -> [a] -> [a]
merge [] l = l
merge l [] = l
merge (x:xs) (y:ys)
    | x < y = x:(merge xs (y:ys))
    | otherwise = y:(merge (x:xs) ys)
```

Merge sort runs in $O(n * log(n))$ time.

Practice exercise LI-24

Solution

```
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (x:xs) =
    (qSort lower) ++ [x] ++ (qSort upper)
    where (lower, upper) = myPartition x xs

myPartition :: Ord a => a -> [a] -> ([a], [a])
myPartition _ [] = ([],[])
myPartition v (x:xs)
    | x < v = (x:a,b)
    | otherwise = (a,x:b)
    where (a,b) = myPartition v xs
```

Quicksort runs in $O(n * log(n))$ time.

3.3 Lists by comprehension

List comprehensions can be used to build lists using other lists.

The general structure for list comprehensions is:

```
[<pattern> | <generator 1>, <generator 2>, ..., <guard 1>, <guard 2> ...]
```

where each generator has the format `<pattern> <- <list>` . Each generator is responsible for iterating through its list and producing a value for each element that is visited. Guards define certain conditions that the combination of generated values must follow. An instance of the pattern is added to the output list only if these conditions are met (i.e. guards work like filters). The pattern corresponds to an expression that depends on the values returned by generators and other variables that are within the scope of the list comprehension. Examples:

```
Prelude> [x^2 | x <- [1..10]]
[1,4,9,16,25,36,49,64,81,100]
Prelude> [x^2 | x <- [1..10], odd x]
[1,9,25,49,81]
Prelude> [x^2 | x <- [1..10], odd x, mod x 3 == 0]
[9,81]
```

The value on the left side of `<-` can be a "pattern", namely those used for lists and tuples. Examples:

```
Prelude> [x | (x:_)<-[[1,2],[3,4]]]
[1,3]
Prelude> [(a,b) | (a,b) <- zip [1..3] [1..]]
[(1,1),(2,2),(3,3)]
```

Using multiple generators works like with nested loops: for each value of the leftmost generator, all combinations of values of the generators to the right are produced. Thus, changing the order of the generators changes the order of the elements in the resulting list. The generators can depend on the variables introduced by generators more to the left. Examples:

```
Prelude> [(x,y) | x <-[1,2], y<-"ab"]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
Prelude> [(x,y) | y<-"ab", x <-[1,2]]
[(1,'a'),(2,'a'),(1,'b'),(2,'b')]
Prelude> [[x,y] | x <-"ab", y<-x:"ab"]
["aa","aa","ab","bb","ba","bb"]
```

Sample exercises

Sample exercise LI-25

Problem statement

Indicate the result of each expression.

a) `[x-1 | x <-[1,2,3]]`

b) `[[x,y] | x <-[1,2], y <- [3,4]]`

c) `[(x^2,y)| (x,y)<- zip [1..5] [1,4..]]`

d) `[x | x <-[1..20], mod x 2 == 0, mod x 3 /= 0]`

e) `[x^2 | x <-[1..]]`

Solution

a) `[0,1,2]`

b) `[[1,3],[1,4],[2,3],[2,4]]` Notice that the generators are processed from the left to the right. If they were processed in the opposite direction, the result would be `[[1,3],[2,3],[1,4],[2,4]]` .

c) `[(1,1),(4,4),(9,7),(16,10),(25,13)]`

d) `[2,4,8,10,14,16,20]` This list consists of the integers from 1 to 20 that are divisible by 2 but not by 3. These divisibility checks are performed using guards.

e) `[1,4,9,16,...]` The result is an infinite list with all the perfect squares.

Sample exercise LI-26

Problem statement

a) Without using the list itself, define the following list by comprehension.

```
myList = [(0,5),(1,4),(2,3),(3,2),(4,1),(5,0)]
```

b) Implement funList that generalizes the list above with respect to a positive integer n.

Usage examples:

```
*Main> myList
[(0,5),(1,4),(2,3),(3,2),(4,1),(5,0)]
*Main> funList 7
[(0,7),(1,6),(2,5),(3,4),(4,3),(5,2),(6,1),(7,0)]
```

Solution

a)

```
myList = [(x,5-x) | x <- [0..5]]
```

Complex expressions, such as tuples, can be used for the pattern section of a list comprehension.

b)

```
funList :: Int -> [(Int,Int)]
funList n = [(x,n-x) | x <- [0..n]]
```

Sample exercise LI-27

Problem statement

Implement the scalarProduct, which computes the scalar product between two lists X and Y with the same length n, using the formula:

$$X \cdot Y = \sum_{k=1}^{n} x_i * y_i$$

Usage examples:

```
*Main> scalarProduct [1,-1,5] [4,3,4]
21
```

Solution

```
scalarProduct :: Num a => [a] -> [a] -> a
scalarProduct l1 l2 = sum [x*y | (x,y) <- zip l1 l2]
```

The zip function is very useful for list comprehensions as it builds lists of correspondences between different lists. In this case, zip was used to build the correspondences between elements of the same position in both lists.

Sample exercise LI-28

Problem statement

Define the variable infL, which contains an infinite list starting with 1 and where the element at i-th position a_i is given by: $a_i = a_{n-1} + n - 1$ for $i \geq 2$ (with indices starting at 1).

Usage examples:

```
*Main> infL
[1,3,8,19,42,89,184...]
```

Solution

```
infL :: Integral a => [a]
infL = 1:[2*prev + n - 1 | (n, prev) <- zip [2..] infL]
```

This solution illustrates how to create a sequence using a list comprehension where each element depends both on its index and previous element. The trick is to define a "recursive generator", whose expression contains the list itself. The generator produces pairs (using `zip`) containing each element's index and previous element.

The first element of the list is concatenated to the list comprehension (using the cons operator, `(:)`), rather than being included in the list comprehension

itself. This is analogous to the “base case” of a recursive implementation of a function.

Practice exercises

Note: in the following exercises, when defining a function that returns a list, list comprehensions should be used. Functions should only have one definition, unless stated otherwise.

Practice exercise LI-29

Problem statement

Indicate the result of each expression.

a) `[mod x 7 | x <- ([1..5] ++ [16..23])]`

b) `[x ++ "␣the␣"++ y | x <- ["buy","loan"], y <- ["car","house"]]`

c) `[x | x <-[-5..5], abs(x^3)<= 20]`

d) `take 10 [-x | x <- cycle [4,7,8]]`

e) `take 10 [ 5*x*y | x<-[1..], y <-[1..]]`

f) `[(a+1,b)| (a,b)<- zip [1..3] [10..]]`

g) `[[x | (x,y)<- zip xs (tail xs), x > y] | xs <- [[3,4,3],[4,3,3],[4,2,3,1],[5,4,1],[4,3,2,1]]]`
↪

Practice exercise LI-30

Problem statement

Implement myConcat, described in exercise LI-8, using a list comprehension.

Practice exercise LI-31

Problem statement

Implement differentFromNext which receives a list and returns all the elements that are different from the one that comes next in the list.

```
*Main> differentFromNext "aabbaacca"
"abac"
*Main> differentFromNext "aab"
"a"
*Main> differentFromNext "aaa"
""
```

Practice exercise LI-32

Problem statement

Implement conseqPairs that receives a list l and returns a list of pairs of consecutive elements in l.

Usage examples:

```
*Main> conseqPairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

Practice exercise LI-33

Problem statement

Implement myZip3, which, similarly to Prelude's zip3, returns a list of triples containing the elements of the three input lists occurring at the same positions.

Usage examples:

```
*Main> myZip3 [1,2] [3,4] ['a','b']
[(1,3,'a'),(2,4,'b')]
```

Practice exercise LI-34

Problem statement

Implement dropN which, given a list l and a natural number n, removes the last element of l in each sublist of n elements.

Usage examples:

```
*Main> dropN [1..10] 3
[1,2,4,5,7,8,10]
```

Practice exercise LI-35

Problem statement

Implement checkMod3ThenOdd which checks whether all of the elements in a list of integers that are divisible by 3 are odd.

Usage examples:

```
*Main> checkMod3ThenOdd [1,15,153,83,64,9]
True
*Main> checkMod3ThenOdd  [1,12,153,83,9]
False
```

Practice exercise LI-36

Problem statement

Implement repeatNTimes which, given a list l and a natural number n, returns a list where each element is repeated n times (with the repetitions appearing in the same order as in l).

Usage examples:

```
*Main> repeatNTimes 4 "ab"
"aaaabbbb"
```

Practice exercise LI-37

Problem statement

Implement myCycle, described in exercise LI-10, using a list comprehension.

Practice exercise LI-38

Problem statement

Implement the following unary functions which approximate some mathematical constant using infinite series. The functions receive as argument an integer indicating how many terms of the series should be used for the approximation.

a) Implement myE which provides an approximation of Euler's number e using the following equation:

$$e = 1 + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + ...$$

a) Implement myPi which provides an approximation of pi using the following equation:

$$\frac{\pi^2}{12} = 1 - \frac{1}{4} + \frac{1}{9} - \frac{1}{16} + \frac{1}{25} + ...$$

Usage examples:

```
*Main> myE 1000
2.7182818284590455
*Main> myPi 1000
3.141591699614916"
```

Practice exercise LI-39

Problem statement

Implement myPermutations, which, similarly to Data.List's permutations, returns a list with all the permutations of a list.

To implement the solution, you are allowed to use Data.List's delete function and to use two definitions for myPermutations.

Usage examples:

```
*Main> myPermutations "abc"
["abc","acb","bac","bca","cab","cba"]
```

Practice exercise LI-40

Problem statement

Define the following infinite lists in comprehension.

a) `[0,1,4,7,8,31,12,127,16,511,...]`

b) The Fibonacci sequence, where each element is the sum of the two previous elements: `fib = [1,1,2,3,5,8,13...]` . Suggestion: store the list in a variable.

Practice exercise LI-41

Problem statement

Caesar's cipher is used to encode messages. When this cipher is applied to a string s, each letter of s is replaced by the letter that comes n positions after in the alphabet (considering that 'a' comes after 'z', forming a full cycle).

a) Implement cipher, which applies Caesar's cipher with n shifts to a string.

Suggestion: Define an auxiliary function, which given the number of shifts n, computes a list of pairs `(char,cipherChar)` where `cipherChar` is the cipher for the character `char` .

Note: n can be negative. In this case, the cipher for a character is the one that comes n positions before in the alphabet (considering 'z' comes before 'a').

b) Implement decipher, which, given the number of shifts n and a string s, computes the string s' whose Caesar's cipher is s (i.e. it decodes the message).

Usage examples:

```
*Main> cipher 1 "test"
"uftu"
*Main> cipher 13 "test"
"grfg"
*Main> decipher 13 "grfg"
"test"
```

Practice exercise LI-42

Problem statement

Consider the definition of function f below:

```
f :: (Integral a) => [a]
f  = g [2 ..]

g :: (Integral a) => [a] -> [a]
g (x:xs) = x:(g [x' | x' <- xs, mod x' x /= 0])
```

Explain concisely what function f computes.

Solutions to the practice exercises

Practice exercise LI-29

Solution

a) `[1,2,3,4,5,2,3,4,5,6,0,1,2]`
b) `["buy the car","buy the house","loan the car","loan the house"]`
c) `[-2,-1,0,1,2]`
d) `[-4,-7,-8,-4,-7,-8,-4,-7,-8,-4]`
e) `[5,10,15,20,25,30,35,40,45,50]`
f) `[(2,10),(3,11),(4,12)]`
g) `[[4],[4],[4,3],[5,4],[4,3,2]]`

Two-dimensional lists can be also be defined by comprehension.

Practice exercise LI-30

Solution

```
myConcat :: [[a]] -> [a]
myConcat l = [x | xs <- l, x <- xs]
```

This solution uses two generators: the first one extracts each sublist xs of the input, while the second one extracts each element x of xs.

Practice exercise LI-31

Solution

```
differentFromNext :: Eq a => [a] -> [a]
differentFromNext l = [x | (x,y) <- zip l (tail l), x /= y]
```

zip was used to build the correspondence between each element and its successor in the input list.

Practice exercise LI-32

Solution

```
conseqPairs :: Eq a => [a] -> [a]
conseqPairs l = [(a,b) | (a,b) <- zip l (tail l)]
```

Practice exercise LI-33

Solution

```
myZip3 :: [a] -> [b] -> [c] -> [(a,b,c)]
myZip3 l1 l2 l3 = [(x,y,z) | (x,(y,z)) <- zip l1 (zip l2 l3)]
```

Practice exercise LI-34

Solution

```
-- version with zip on natural numbers
dropN :: Integral p => [a] -> p -> [a]
dropN l n = [x | (x,p) <- zip l [1..], mod p n /= 0]
-- version with zip on cycle
dropN' :: Integral p => [a] -> p -> [a]
dropN' l n = [x | (x,p) <- zip l (cycle [1..n]), p /= n]
```

Both solutions are quite elegant since they establish a correspondance between the elements of the list and a subset of the natural numbers.

The first solution matches elements with their index (with indices starting at 1). The second list can be infinite since zip's output will be limited by the length of the first argument. Having established these correspondences, it is easy to assert that only elements in positions not divisible by n should be returned.

The second solution labels each element with a number from 1 to n, and excludes those with a label equal to n.

Practice exercise LI-35

Solution

```
checkMod3ThenOdd :: Integral a => [a] -> Bool
checkMod3ThenOdd l = and[mod x 2 == 1| x <- l, mod x 3 == 0]
```

In this problem, we want to determine if all the elements obey the logical implication $x \mod 3 = 0 \rightarrow x \mod 2 = 1$. The antecedent is placed as a guard in the list comprehension to remove elements that do not satisfy it from the resulting list. The consequents are the elements themselves of the filtered list, to which the logical "and" function is then applied to check if the consequent is always satisfied.

Practice exercise LI-36

Solution

```
repeatNTimes :: Integral a => a -> [b] -> [b]
repeatNTimes n l = [x | x <- l, _ <- [1..n]]
```

The second generator is used to replicate each element. This is possible since the second generator returns each element of its list for each element of the first list.

Practice exercise LI-37

Solution

```
myCycle :: [a] -> [a]
myCycle l = [x | _ <- [1 ..], x <- l]
```

This solution is similar to the one from exercise LI-36 as it also uses a generator of natural numbers to repeat the extraction of each element of the input list (without using the value of the natural numbers themselves). The two differences are that list of natural numbers is infinite (since cycle returns an infinite list) and that the order of the generators is switched, to ensure each element of the input is added to the output (in order) before repeating the sequence.

Practice exercise LI-38

Solution

a)

```
-- simple version that computes k! for each k
myE :: Int -> Double
myE n = sum (take n [1/(product[1..k]) | k <- [0..]])
-- alternative version without lazy evaluation
myE' :: Int -> Double
```

```
myE' n = sum [1/(product[1..fromIntegral(k)]) | k <- [0.. n-1]]
-- alternative version with let (more efficient)
myE'' :: Int -> Double
myE'' n = sum (take n eFactors)
eFactors :: [Double]
eFactors = 1:[prev/k | (prev,k) <- zip eFactors [1..]]
```

The formula for the approximation is:

$$e(k) = \sum_{n=0}^{k-1} \frac{1}{k!}$$

The first versions are quite inefficient since a factorial is computed for each position of the list of terms. This version provides the correct approximation since `product []` returns 1.

In the third version, the list of terms is defined once, with each term being defined recursively by the previous one. This allows expansions with as many as 1 000 000 terms to be computed in a short amount of time.

The first and third versions illustrate one of the advantages of lazy evaluation: the logic for generating an infinite list can be separated from the logic to consume it. In both versions, the infinite lists are consumed by `take`, which results in a finite computation.

b)

```
myPi :: Int -> Double
myPi n = sqrt (12 * sum (take n [((-1)**k)/(k+1)^2 | k <- [0..]]))
```

π is computed by isolating it in the expression given in the problem statement. The formula for the approximation is:

$$\pi(k) = \sqrt{12 * \sum_{n=0}^{k-1} \frac{(-1)^k}{(k+1)^2}}$$

The `**` operator is used for exponentiation in the numerator (instead of `^`) to ensure the division is performed on floating-point values, since the `/` operator is not defined for integers.

Practice exercise LI-39

Solution

```
import Data.List

myPermutations :: Eq a => [a] -> [[a]]
myPermutations [] = [[]]
myPermutations l = [h:t | h <- l, t <- myPermutations(delete h l)]
```

The solution contains a recursive list comprehension, as the expression for the rightmost generator depends on myPermutations.

Practice exercise LI-40

Solution

a) `[if (mod i 2 == 0)then 2*i else i^2| i <- [0..]]` if-then-else structures can be used inside list comprehensions since they are evaluated to a value. In this case, they allow one to distinguish two different functions based on the parity of the element's index.

b) `fib = 1:1:[prev1 + prev2 | (prev1, prev2)<- zip fib (tail fib)]`

The reasoning behind this solution is similar to the one from exercise LI-28.

Saving the sequence in a variable allows for it to be used in the expression of the list itself, in a recursive manner. Using `fib` and `tail fib` in the arguments of the zip does not lead to an infinite loop since the first two arguments are directly prepended to the list. Using `drop 2 fib` would lead to an uncontrolled infinite loop where the list could not be properly defined.

Practice exercise LI-41

Solution

a)

```
cipher :: Int -> String -> String
cipher n str = [cipherChar | c <- str, (char,cipherChar) <- codes n, c ==
    ↪ char]

codes :: Int -> [(Char,Char)]
codes n = [pair | pair <- zip ['a' .. 'z'] (drop (mod n 26) (cycle ['a' ..
    ↪  'z']))]
```

The auxiliary function codes computes the correspondences of characters with their cipher. `cycle` is used to compute correspondences where shifts surpass the end of the alphabet, taking advantage of Haskell's lazy evaluation (which prevents an infinite computation from occurring). Finally, `drop` is used to "shift" the alphabet. `mod` is applied to n to produce the correct result when n is negative (since `drop` requires a positive integer as input).

Since `mod` is used to limit n to the range from 0 to 25, the alphabet list only needs to be repeated once (by concatenating `['a' .. 'z']` with itself). Therefore, the usage of `cycle` in this solution is optional.

The cipher function uses the output from codes to find the correct pair for each character in the input string.

b)

```
decipher :: Int -> String -> String
decipher n str = [char | c <- str, (char,cipherChar) <- codes n, c ==
    ↪ cipherChar]
```

Given how the codes auxiliary function is defined, the definition of decipher is very similar to the one for cipher.

Practice exercise LI-42

Solution

Function f computes an infinite (ordered) list of the prime numbers.

Comments about the implementation:

Function g computes the prime number using the sieve of Eratosthenes, where initially all the integers greater than 2 are provided. The algorithm repeats the following steps indefinitely:

1. The first element x of the integer number list is removed and marked as a prime number.
2. All of the multiples of x are removed from the integer number list.

Thanks to lazy evaluation, all of the multiples of x are not removed at once, which would lead to an infinite computation, since the list of integers to be "filtered" is infinite. Instead, the numbers of the infinite list are "extracted" only when they are needed.

Chapter 4
Higher-order functions

A *higher-order function* is a function that either has a function as one of its arguments or returns a function. Since functions are the bread and butter of functional programming, higher-order functions give the programmer a greater degree of freedom. Advantages of using this sort of functions include allowing one to provide operations rather than simple data to other functions, to create re-usable computational patterns and allow one to easily prove certain properties about programs.

This chapter begins by explaining how one can create higher-order functions. Lambdas are then introduced as a useful construction for this sort of functions. Afterwards, the notion of currying is presented, followed by an overview of the main functions in Prelude. The two following sections focus on three notable higher order functions: composition (`.`), application (`$`) and folds. This chapter finishes by presenting a new way of writing functions, in point-free style.

4.1 Fundamentals on higher-order functions

When declaring a functional type, the `->` symbol is right-associative. Thus, indicating that a function f has the type `a -> b -> c` is equivalent to
`a -> (b -> c)` . As a result, parentheses must be used to indicate that an argument is a function. For example, if a function f has two arguments, the first of which being a function, and outputs another function, then its type declaration should be:
`f :: (a -> b)-> c -> (d -> e)` .

To use a function f passed as an argument, one can simply write it using the usual prefix notation $fxyz$, where x, y and z are f's arguments.

Sample exercises

Sample exercise HO-1

Problem statement

Define the twice function which, given a function f and an argument x, applies f twice to x.

Usage examples:

```
*Main> twice succ 0
2
*Main> twice twice succ 0
4
*Main> twice twice twice succ 0
16
```

Solution

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

twice must have the type `(a -> a)-> a -> a` , and not `(a -> b)-> a -> b` since the output of f is used as input to this same function.

Sample exercise HO-2

Problem statement

Implement trapezium/4, which uses the trapezium rule to compute an approximation of the definite integral of $f(x)$ in the interval $[x1, x2]$ with n partitions using the formula:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(x_0) + 2f(x_1) + 2f(x_2) + ... + 2f(x_{n-2}) + f(x_n)\right]$$
$$= \frac{h}{2}\left[f(x_0) + 2\sum_{j=1}^{n-1} f(x_j) + f(x_n)\right]$$

The function has the following arguments (in order): two real numbers x1 and x2, an integer n (which must be even) and an unary function f which receives a real number and returns another number.

Note: n must be even.

Suggestion: Use the seq22 function developed in exercise FT-18 of Chapter 2.

Usage examples:

```
*Main> trapezium 0 (pi/2) 1000 sin
0.9999997943832332
```

```
*Main> let f x = exp(-x**2) in trapezium 0 1 12 f
0.7463982478934403
```

Solution

```
trapezium :: (RealFloat a) => a -> a -> Int -> (a -> a) -> a
trapezium x1 x2 n f
    | even n = (h / 2)*sum [coef * (f (x1 + h*fromIntegral(i))) | (coef,i)
        ↪  <- zip (seq22 (n+1)) [0..n]]
    | otherwise = error "n must be even."
    where h = (x2 - x1)/fromIntegral(n)

seq22 :: Num a => Int -> [a]
seq22 n = 1:(take (n-2) (repeat 2)) ++ [1]
```

fromIntegral is used to obtain a floating-point equivalent of an integer. n is an Int since it is used as input to the auxiliary function seq22, which, in turn, passes it to take, which constrains this variable to be an Int. Since i is generated by n, it is also an Int, which is why fromIntegral must be applied to it before it can be safely multiplied by the floating-point number h.

Practice exercises

Practice exercise HO-3

Problem statement

a) Define applyN, which, given a function f, an integer n and an argument x, applies f n times to x.

b) Using applyN, define cipher which, given an integer n and a string s, applies Caesar's cipher to s, as described in exercise LI-41. n is assumed to be positive.

Usage examples:

```
*Main> applyN succ 10 0
10
*Main> cipher 1 "test"
"uftu"
*Main> cipher 13 "test"
"grfg"
```

Practice exercise HO-4

Problem statement

Write a function with type `(a -> b -> b)-> b -> a -> b`.

Practice exercise HO-5

Problem statement

Implement bisection, which finds the zero of an unary function using the bisection method. The zero of a function is a value for x such that $f(x) = 0$.

The bisection method receives as input two real values x_1 and x_2 such that $x_1 \leq x_2$ and $f(x_1)$ and $f(x_2)$ have opposite signs. Its third argument is the maximum error, $error$. Finally, the fourth argument is the function f.

The method executes the following steps:

1. Compute the midpoint x_m of the interval: $x_m = \frac{x_1+x_2}{2}$.
2. Compute $f(x_m)$.
3. If either $x_2 - x_1 \leq error$ or $f(x_m) \leq error$, then return x_m.
4. If not, replace x_1 with x_m if $f(x_m) and f(x_2)$ have opposite signs and repeat step 1.
5. If not, replace x_2 with x_m and repeat step 1.

Usage examples:

```
*Main> bisection (-2) 2 0.0001 id
0.0
*Main> bisection 0 2 0.0001 cos
1.57080078125
```

Practice exercise HO-6

Problem statement

Implement simpson, which uses Simpson's rule to compute an approximation of the definite integral of $f(x)$ in the interval $[x1, x2]$ with n partitions using the formula:

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{h}{3}\Big[f(x_0) + 4f(x_1) + 2f(x_2) + ... + 4f(x_{n-2}) + f(x_n)\Big] \\ &= \frac{h}{3}\sum_{j=1}^{n/2}\Big[f(x_{2j-2}) + 4f(x_{2j-1}) + f(x_{2j})\Big] \\ &= \frac{h}{3}\Big[f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\Big]\end{aligned}$$

The function has the following arguments (in order): two real numbers x1 and x2, an integer n (which must be even) and an unary function f which receives a real number and returns another number.

Note: n must be even.

Suggestion: Use the seq42 function developed in exercise FT-18 of Chapter 2.

Usage examples:

```
*Main> simpson 0 (pi/2) 1000 sin
1.000000000000033
*Main> let f x = exp(-x**2) in simpson 0 1 12 f
0.7468245263791943
```

Practice exercise HO-7

Problem statement

Implement sortByCond, which given a list l and a comparison function f (which indicates if an element e_1 is smaller than e_2), returns a sorted list of l using the comparison criterion of f.

Usage examples:

```
*Main> sortByCond [1..10] (>)
[10,9,8,7,6,5,4,3,2,1]
*Main> sortByCond [1..10] (<)
[1,2,3,4,5,6,7,8,9,10]
```

Solutions to the practice exercises

Practice exercise HO-3

Solution

a)

```
applyN :: (Integral n) => (a -> a) -> n -> a -> a
applyN _ 0 x = x
applyN f n x
    | (n > 0) = f (applyN f (n-1) x)
    | otherwise = error "Negative number"
```

b)

```
cipher :: (Integral n) => n -> [Char] -> [Char]
cipher _ [] = []
cipher n (x:xs) = (applyN nextChar n x):(cipher n xs)

nextChar :: Char -> Char
nextChar 'z' = 'a'
nextChar c = succ c
```

The auxiliary function nextChar returns the letter that comes after its input, alphabetically. nextChar is then applied n times to each element of the string.

Practice exercise HO-4

Solution

```
f g b a = g a (g a b)
```

For clarity, f's second and third argument, a and b, were named as the type variables representing their respective types.

f receives as input a function g whose first argument must be of the same type as f's third argument (a) and whose second argument must be of the same type as f's second argument b. To ensure these equalities, the innermost call gab is made.

Using gab in the second argument of the outermost call to g ensures g's output is of the same type as its argument, which in turn ensures the output type of f is the same as its second argument b.

Practice exercise HO-5

Solution

```
bisection :: (RealFloat a) => a -> a -> a -> (a -> a) -> a
bisection x1 x2 err f
    | y1*y2 > 0 = error "f(x1) and f(x2) have the same sign."
    | x1 <= x2 = bisectionAux x1 x2 err f
    | otherwise = bisectionAux x2 x1 err f
    where y1 = f x1
          y2 = f x2

bisectionAux x1 x2 err f
    | abs(x1 - x2) > err && abs(ym) > err = if (y1*ym < 0)
       then bisectionAux x1 xm err f
       else bisectionAux xm x2 err f
    | otherwise = xm
    where y1 = f x1
          y2 = f x2
          xm = (x1+x2)/2
          ym = f xm
```

An error is issued if $f(x_1)$ and $f(x_2)$ have the same sign.

The main function, bisection, re-orders x_1 and x_2 to ensure $x_1 \leq x_2$. The auxiliary function, bisectionAux, performs the recursion.

Practice exercise HO-6

Solution

```
simpson :: (RealFloat a) => a -> a -> Int -> (a -> a) -> a
simpson x1 x2 n f
    | even n = (h / 3)*sum [coef * (f (x1 + h*fromIntegral(i))) | (coef,i)
        ↪  <- zip (seq42 (n+1)) [0..n]]
    | otherwise = error "n must be even."
    where h = (x2 - x1)/fromIntegral(n)

seq42 :: Num a => Int -> [a]
seq42 n = 1:(take (n-2) (cycle [4,2])) ++ [1]
```

Simpson's method is quite similar to the trapezium rule. One simply needs to change the auxiliary function from seq22 to seq42 and the coefficient from 2 to 3 in the result.

Practice exercise HO-7

Solution

```
import Data.List -- splitAt

-- Quick sort version
sortByCond :: Ord a => [a] -> (a -> a -> Bool) -> [a]
sortByCond [] _ = []
sortByCond (x:xs) cmp =
    (sortByCond lower cmp) ++ [x] ++ (sortByCond upper cmp)
    where (lower, upper) = myPartition x xs cmp

myPartition :: Ord a => a -> [a] -> (a -> a -> Bool) -> ([a], [a])
myPartition _ [] _ = ([],[])
myPartition v (x:xs) cmp
    | cmp x v = (x:a,b)
    | otherwise = (a,x:b)
    where (a,b) = myPartition v xs cmp

-- Merge sort version
sortByCond' :: Ord a => [a] -> (a -> a -> Bool) -> [a]
sortByCond' [] _ = []
sortByCond' [x] _ = [x]
sortByCond' l cmp =
   merge (sortByCond' l1 cmp) (sortByCond' l2 cmp) cmp
   where (l1,l2) = splitAt (div (length l) 2) l

merge :: Ord a => [a] -> [a] -> (a -> a -> Bool) -> [a]
merge [] l _ = l
merge l [] _ = l
merge (x:xs) (y:ys) cmp
    | cmp x y = x:(merge xs (y:ys) cmp)
    | otherwise = y:(merge (x:xs) ys cmp)
```

4.2 Lambdas

A useful structure for defining higher-order functions are *lambdas*, also known as *anonymous functions* (i.e. functions without a name). Lambdas are used to define functions *on-the-fly*, which means that, rather than writing a definition for a function that will be used once, the function can be written in an expression.

A lambda for a function with arguments x1, x2 ... xN and f(x1,x2,...,xN) is written as: `\x1 x2 ... xN -> f x1 x2 ... xN` . The backslash symbol (`\`) was adopted to mark the beginning of lambdas since it is reminiscent of the greek letter λ (lambda), which is used to define functions in lambda calculus.

A limitation when constructing lambdas (in relation to conventional, named functions) is that they must be defined in a single clause (unlike what is done, for example, in recursive functions, which typically have at least two clauses: one for the base case and another for the recursive case). Pattern matching is also possible with lambdas. If the pattern does not match, a runtime error occurs. Typically, one should surround the whole lambda with parentheses to clearly mark where the function's body ends. Examples:

```
Prelude> (\x -> x + 1) 2
3
Prelude> (\(x:xs) -> x) [1..10]
1
```

Practice exercises

Practice exercise HO-8

Problem statement

Using a lambda, implement myFlip, which, similarly to Prelude's flip, receives a binary function and returns another function with identical behavior but with its arguments in reverse order.

Usage examples:

```
*Main> myFlip (/) 1 2
2.0
*Main> myFlip (/) 2 1
0.5
*Main> myFlip (:) [1..5] 0
[0,1,2,3,4,5]
```

Solutions to the practice exercises

Practice exercise HO-8

Solution

```
-- Alternative with a binary lambda function
myFlip :: (a -> b -> c) -> (b -> a -> c)
myFlip f = (\x y -> f y x)
-- Alternative with a ternary lambda function
myFlip' :: (a -> b -> c) -> (b -> a -> c)
myFlip' = (\f x y -> f y x)
```

Since myFlip returns a function, the output can be a lambda.

4.3 Currying

In Haskell, all functions actually only accept one argument. Functions with multiple arguments can be considered as a series of functions which receive an argument and return a function which receives the second argument, and so on. This is known as *currying*, a reference to the mathematician Haskell Curry, who shares his first name with the programming language.

All functions with multiple arguments are thus considered to be *curried*. This behavior can be observed in the function's type: since `->` is right-associative, then `f :: a -> b -> c` is equivalent to `f :: a -> (b -> c)`, that is, a binary function returns an unary function. To simplify explanation, it has been and will often be considered that functions take multiple arguments.

Arguments are passed to functions, one-by-one, by putting spaces between the function's name and the name of each argument. This is known as *application*. If fewer arguments are passed to a function than what the function can accept, the function is known to be *partially applied*: rather than returning a value of its output type, it returns a function that receives the remainder of its inputs and only then returns its output.

Just like lambdas, partial application has the advantage of avoiding the need to create functions that would otherwise only be used once. It has the advantage of allowing for an even lighter syntax than lambdas. Consider the following example which uses Prelude's `map` function (to be introduced later on in this chapter), that applies a function to each element of a list. In this example, the goal is to remove the first two elements of each sublist in a list:

```
(First alternative: definiting a function explicitly.)
Prelude> let f l = drop 2 l
Prelude> map f [[1,2,3],[4,5,6]]
[[3],[6]]
(Second alternative: using a lambda.)
Prelude> map (\l -> drop 2 l) [[1,2,3],[4,5,6]]
[[3],[6]]
(Third alternative: using partial application.)
Prelude> map (drop 2) [[1,2,3],[4,5,6]]
[[3],[6]]
```

Infix operators can also be partially applied. They are known as *sections*. Examples:

```
Prelude> map (*2) [1..5]
[2,4,6,8,10]
Prelude> map (2*) [1..5]
[2,4,6,8,10]
Prelude> (2*) 3
6
Prelude> (`elem` [1..3]) 3
True
```

As an alternative to currying, one could define unary functions that receive a tuple with all the arguments of their expressions. However, this is strongly discouraged, as tuple arguments do not have the same level of flexibility as curried arguments (as seen in the examples above). Tuple arguments should only be used when one of the input arguments is actually supposed to be a tuple. Example:

```
(First alternative: curried function.)
add :: Int -> (Int -> Int)
add x y = x + y
(Second alternative: function with a single tuple argument.)
add' :: Int -> (Int -> Int)
add' (x,y) = x + y
```

Sample exercises

Sample exercise HO-9

Problem statement

Implement myCurry, which, similarly to Prelude's curry, receives an unary function with a pair as input and returns an equivalent curried function.

Usage examples:

```
*Main> myCurry fst 2 3
2
```

Solution

```
-- Alternative without lambdas
myCurry :: ((a,b) -> c) -> a -> b -> c
myCurry f a b = f (a,b)
-- Alternative with a binary lambda function
myCurry' :: ((a,b) -> c) -> a -> b -> c
myCurry' f = \a b -> f (a,b)
-- Alternative with a ternary lambda function
myCurry'' :: ((a,b) -> c) -> a -> b -> c
myCurry'' = \f a b -> f (a,b)
```

Even though the textual definition of myCurry in the problem statement suggests that it is an unary function, one can define it with three arguments,

as presented in the first alternative, due to how currying works: functional arguments are applied one-by-one.

Practice exercises

Practice exercise HO-10

Problem statement

Implement myUncurry, which, similarly to Prelude's uncurry, receives a binary (curried) function and returns an equivalent unary function that receives a tuple containing both arguments.

Usage examples:

```
*Main> myUncurry (+) (1,2)
3
*Main> myUncurry (mod) (5,2)
1
```

Solutions to the practice exercises

Practice exercise HO-10

Solution

```
-- Alternative without lambdas
myUncurry :: (a -> b -> c) -> (a,b) -> c
myUncurry f (a,b) = f a b
-- Alternative with a binary lambda function
myUncurry' :: (a -> b -> c) -> (a,b) -> c
myUncurry' f = \(a,b) -> f a b
-- Alternative with a ternary lambda function
myUncurry'' :: (a -> b -> c) -> (a,b) -> c
myUncurry'' = \f (a,b) -> f a b
```

4.4 Common higher-order functions

Table 4.1 presents some of the higher-order functions in Prelude. The foldr and foldl functions will be further explored in the last section of this chapter.

Table 4.1 Some higher-order Prelude functions

Function	Comment	Example
map	Applies a function to each element of a list.	`map succ [1,2,3] -> [2,3,4]`
filter	Returns a sublist with the elements that satisfy a predicate (i.e. function that returns a boolean).	`filter odd [1..5] -> [1,3,5]`
any	Checks if at least one element of a list satisfies a predicate.	`any even [1,1,1,3,1] -> False`
all	Checks if all the elements of a list satisfy a predicate.	`all odd [1,1,1,3,1] -> True`
takeWhile	Returns the longest prefix of a list that satisfies a predicate.	`takeWhile odd [1,1,1,3,2] -> [1,1,1,3]`
dropWhile	Returns the remainder of a list after calling takeWhile.	`dropWhile odd [1,1,1,3,2] -> [2]`
iterate	Returns an infinite list where the i-th element is the application of a function f on a value x i times (with indices starting at 0).	`iterate succ 0 -> [0,1,2,3,4,5,6,7,8,9,10,...]`
zipWith	Zips two lists, then combines each pair using a binary function.	`zipWith (+)[1,2,3,4] [3,2..] -> [4,4,4,4]`
flip	Swaps the order of the arguments in a binary function.	`flip (/)1 2 -> 2.0`
`(.)`	Function composition.	(Check section 4.5.)
`($)`	Function application.	(Check section 4.5.)
foldr	Right-associative fold of a structure.	(Check section 4.6.)
foldl	Left-associative fold of a structure.	(Check section 4.6.)

Sample exercises

Sample exercise HO-11

Problem statement

Indicate the result of the following expressions:

a) `map fst [(1,'a'),(3,'b'),(2,'c'),(5,'d'),(4,'e')]`

b) `map (++ "?")["How", "What", "Who", "When", "Where", "Which"]`

c) `filter (`elem` ['A' .. 'Z'])"mAPs and filtErS"`

d) `((map (+)[2,4..])!! 3)5`

e) `zipWith (-)[1,3..] [1..]`

f) `takeWhile (>3)[1,2,3,4,5]`

g) `dropWhile (x-> (succ x)< 'u')['a'..'z']`

Solution

a) `[1,3,2,5,4]`

The `fst` function is mapped against every element of the list, so each element of the output is the first element of the corresponding pair.

b) `["How?","What?","Who?","When?","Where?","Which?"]`

`map` also works when the mapping function is partially applied.

c) `"APES"`

The functional argument of the filter has to be written like so since, for each character c of the string, the following test is performed: `(`‘`elem`‘ `['A' .. 'Z'])c` , which is equivalent to `c` ‘`elem`‘ `['A' .. 'Z'] . elem` is written with backticks since it is used as an infix operator.

d) `13`

maps can also return functions, which can then be applied.

e) `[0,1,2,3,4,5,..]`

f) `[]`

No element is returned in the output list since the first element in the input list is smaller than 3.

g) `"tuvwxyz"`

Sample exercise HO-12

Problem statement

Consider the myAny function, which, similarly to Prelude's any, given a predicate p and a list l, checks if at least one element of l satisfies p.

a) Implement myAny using recursion (and without using user higher-order functions).

b) Implement myAny using map and without using recursion.

c) Implement myAny using other higher-order functions (other than any and map) and without using recursion.

Usage examples:

```
*Main> myAny even [1,1,1,3,1]
False
*Main> myAny even [1,1,1,3,2]
True
```

Solution

a)

```
myAny :: (a -> Bool) -> [a] -> Bool
myAny _ [] = False
myAny p (x:xs) = (p x) || myAny p xs
```

The base case (with the empty list) returns `False` (and not `True`) since no elements satisfy p.

b)

```
myAny' :: (a -> Bool) -> [a] -> Bool
myAny' p l = or (map p l)
-- Version without explicit list argument and composition
myAny'' :: (a -> Bool) -> [a] -> Bool
```

```
myAny'' p = or . (map p)
```

map converts the list into a list of boolean indicating whether each element of the input list satisfies p.

The second alternative solution composes or with `map p` rather than `map` since or expects a list as input and `map p` returns a list. On the other hand, `map` has the type `(a -> b)-> ([a] -> [b])`, so it returns a function that receives a list and returns another list.

c)

```
myAny''' :: (a -> Bool) -> [a] -> Bool
myAny''' p l = not (all (not . p) l)
```

The solution resorts to a logical equivalence established by De Morgan's laws - if there is an x that satisfies p, then it is false to say that "all x's do not satisfy p":

$$\bigvee_{x \in L} P(x) \equiv \neg \bigwedge_{x \in L} \neg P(x)$$

There must be a dot (.) to compose not with p since not receives as input a boolean, not a predicate.

Practice exercises

Practice exercise HO-13

Problem statement

Indicate the result of the following expressions:

a) `map (replicate 3)[1..3]`
b) `map (`div`3)[1..10]`
c) `filter even (map (^2)[1..])`
d) `sum (takeWhile (<= 100)(filter even (map (^2)[1..])))`
e) `zipWith (xy -> (x * 10 + 3)/ y)[5,4,3,2,1] [1,2,3,4,5]`
f) `takeWhile (<200)(iterate (3*)1)`
g) `length(filter (<='z')['a'..])`

Practice exercise HO-14

Problem statement

Using higher-order functions rather than recursion, implement orderedTriples, which, given a list of triples, returns a list containing only the triples `(a,b,c)` where $a \leq b \leq c$ (where the elements appear in the same order as in the input).

Usage examples:

```
*Main> orderedTriples [(1,2,3),(1,3,2),(2,3,1),(10,20,30)]
[(1,2,3),(10,20,30)]
```

Practice exercise HO-15

Problem statement

Implement myMap, which, similarly to Prelude's map, applies a function to each element of a list.

Usage examples:

```
*Main> myMap (*2) [1..5]
[2,4,6,8,10]
*Main> myMap (subtract 2) [1..5]
[-1,0,1,2,3]
```

Practice exercise HO-16

Problem statement

Implement myFilter, which, similarly to Prelude's filter, returns a sublist of the input list with the elements that satisfy a predicate.

Usage examples:

```
*Main> myFilter (>3) [1..10]
[4,5,6,7,8,9,10]
```

Practice exercise HO-17

Problem statement

a) Implement myTakeWhile, which, similarly to Prelude's takeWhile, given a predicate p and a list l, returns the longest prefix of l that satisfies l.

b) Implement myDropWhile, which, similarly to Prelude's dropWhile, given a predicate p and a list l, returns the remainder of a list after an equivalent call to takeWhile.

Usage examples:

```
*Main> myTakeWhile odd [1,1,1,3,1]
[1,1,1,3,1]
*Main> myTakeWhile odd [1,1,1,3,2]
[1,1,1,3]
```

```
*Main> myDropWhile odd [1,1,1,3,1]
[]
*Main> myDropWhile odd [1,1,1,3,2]
[2]
```

Practice exercise HO-18

Problem statement

Consider the myAll function, which, similarly to Prelude's any, given a predicate p and a list l, checks if all the elements of l satisfy p.

a) Implement myAll using recursion (and without using user higher-order functions).

b) Implement myAll using map and without using recursion.

c) Implement myAll using other higher-order functions (other than any and map) and without using recursion.

Usage examples:

```
*Main> myAll odd [1,1,1,3,1]
True
*Main> myAll odd [1,1,1,3,2]
False
```

Practice exercise HO-19

Problem statement

Using a list comprehension, implement myIterate, which, similarly to Prelude's iterate, returns an infinite list where the i-th element is the application of a function f on a value x i times (with indices starting at 0).

Usage examples:

```
myIterate (+2) 1
[1,3,5,7,9,11,..]
myIterate (*3) 1
[1,3,9,27,81,...]
```

Practice exercise HO-20

Problem statement

Consider the myZipWith function, which, similarly to Prelude's zipWith, given a predicate p and two lists, zips both lists and then combines each pair using a binary function.

a) Implement myZipWith using recursion (and without using user higher-order functions).

b) Implement myZipWith using other higher-order functions and without using recursion.

Usage examples:

```
*Main> myZipWith (+) [1..5] [1..5]
[2,4,6,8,10]
*Main> myZipWith (+) [1..5] [2,4..]
[3,6,9,12,15]
```

Practice exercise HO-21

Problem statement

Consider the myUntil function, which, similarly to Prelude's until, given a predicate p, a unary function f and a value x, applies f to x until the predicate p is satisfied.

a) Implement myUntil using recursion (and without using user higher-order functions).

b) Implement myUntil using other higher-order functions and without using recursion.

c) Using myUntil and without using recursion, implement myGcd, which, similarly to Prelude's gcd, computes the greatest common divisor of two integers.

Usage examples:

```
*Main> myUntil (> 5) succ 1
6
*Main> myGcd 12 8
4
*Main> myGcd 1280 (-860)
20
```

Practice exercise HO-22

Problem statement

Using a higher-order function and without using recursion, implement countVowels, which, given a string s, returns the number of vowels (aeiou) in s.

Usage examples:

```
*Main> countVowels "happy days"
2
*Main> countVowels ['a'..'z']
5
```

Practice exercise HO-23

Problem statement

Consider the definition of function f below:

```
f :: (Eq a) => [a] -> [a] -> Int
f x y = length (takeWhile (\(a,b) -> a == b) (zip x y))
```

Explain concisely what function f computes.

Practice exercise HO-24

Problem statement

A Hamming number of a number that can be obtained by multiplying 2, 3 and 5 any amount of times. In other words, a hamming number h is given by $2^i * 3^j * 5^k$, where $i, j, k \geq 0$.

Consider the following function, which computes an infinite list with the Hamming numbers.

```
hamming :: Integral a => [a]
hamming = 1:(map (2*) hamming) ++ (map (3*) hamming) ++ (map (5*) hamming)
```

a) Explain concisely why the output of function hamming is not "ideal".

b) Redefine hamming to correct the issue by replacing both `(++)` operators with a custom operator. Do not change any other part of the hamming function.

Solutions to the practice exercises

Practice exercise HO-13

Solution

a) `[[1,1,1],[2,2,2],[3,3,3]]`

b) `[0,0,1,1,1,2,2,2,3,3]`

c) `[4,16,36,64,100,...]`

d) `220`

e) `[53.0,21.5,11.0,5.75,2.6]`

f) `[1,3,9,27,81]`

This expression returns the powers of 3 which are smaller than 200.

g) `26`

This expression returns the number of letters in the alphabet.

Practice exercise HO-14

Solution

```
orderedTriples :: Ord a => [(a,a,a)] -> [(a,a,a)]
orderedTriples = filter (\(a,b,c) -> (a <= b) && (b <= c))
```

Using a filter is a natural choice, since it can be used to only keep the triples that satisfy a given property.

Practice exercise HO-15

Solution

```
-- Alternative with recursion
myMap :: (a -> b) -> [a] -> [b]
myMap _ [] = []
myMap f (x:xs) = (f x):(myMap f xs)

-- Alternative with a comprehension list
myMap' :: (a -> b) -> [a] -> [b]
myMap' f l = [f x | x <- l]
```

Practice exercise HO-16

Solution

```
-- Alternative with recursion
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter _ [] = []
myFilter p (x:xs)
    | p x = x:(myFilter p xs)
    | otherwise = myFilter p xs

-- Alternative with a comprehension list
myFilter' :: (a -> Bool) -> [a] -> [a]
myFilter' p l = [x | x <- l, p x]
```

Practice exercise HO-17

Solution

a)

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

b)

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

Practice exercise HO-18

Solution

Since `all` is the counterpart of `any`, all solutions can be easily adapted from the different implementations of myAny.

a)

```
myAll :: (a -> Bool) -> [a] -> Bool
myAll _ [] = True
myAll p (x:xs) = (p x) && myAll p xs
```

b)

```
myAll' :: (a -> Bool) -> [a] -> Bool
myAll' p l = and (map p l)
-- Version without explicit list argument and composition
myAll'' :: (a -> Bool) -> [a] -> Bool
myAll'' p = and . (map p)
```

c)

```
myAll''' :: (a -> Bool) -> [a] -> Bool
myAll''' p l = not (any (not . p) l)
```

Practice exercise HO-19

Solution

```
myIterate :: (a -> a) -> a -> [a]
myIterate f x = x:[f prev | prev <- myIterate f x]
```

Practice exercise HO-20

Solution

a)

```
myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
myZipWith _ _ [] = []
myZipWith _ [] _ = []
myZipWith f (x:xs) (y:ys) = (f x y):(myZipWith f xs ys)
```

b)

```
myZipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
myZipWith' f l1 l2 = map (uncurry f) (zip l1 l2)
```

`uncurry` is used to convert the input binary function into an unary function that receives a pair as input.

Practice exercise HO-21

Solution

a)

```
myUntil :: (a -> Bool) -> (a -> a) -> a -> a
myUntil p f x
    | p x = x
    | otherwise = myUntil p f (f x)
```

b)

```
myUntil' :: (a -> Bool) -> (a -> a) -> a -> a
myUntil' p f x = head (dropWhile (\x -> not(p x)) (iterate f x))
-- Alternative with composition
myUntil'' :: (a -> Bool) -> (a -> a) -> a -> a
myUntil'' p f x = head (dropWhile (not . p) (iterate f x))
```

c)

```
myGcd :: Integral a => a -> a -> a
myGcd a b = myGcdAux (max posA posB) (min posA posB)
    where posA = abs a
          posB = abs b

myGcdAux :: Integral a => a -> a -> a
myGcdAux a b = fst(myUntil (\(_,b) -> b == 0) (\(a,b) -> (b,mod a b)) (a,b
    ↪ ))
-- Alternative with composition
myGcdAux' :: Integral a => a -> a -> a
myGcdAux' a b = fst(myUntil ((== 0) . snd) (\(a,b) -> (b,mod a b)) (a,b))
```

The main function myGcd ensures the two arguments are both positive and that the first argument is larger than the second one. The auxiliary function myGcdAux performs the actual computation of the GCD.

Since the until function deals with unary functions, pairs are used in order to work with the two integers needed for the computation of the GCD.

Practice exercise HO-22

Solution

```
-- Alternative with filter
countVowels :: [Char] -> Int
countVowels l =
```

```
    length $ filter (\c -> (c == 'a') || (c == 'e') || (c == 'i') || (c ==
        ↪ 'o') || (c == 'u')) l
-- Alternative with map
countVowels' :: [Char] -> Int
countVowels' l =
    sum $ map (\c -> if ((c == 'a') || (c == 'e') || (c == 'i') || (c == '
        ↪ o') || (c == 'u')) then 1 else 0) l
```

The solution with a filter is more natural.

Practice exercise HO-23

Solution

Function f computes the largest common prefix of the input lists x and y.

Practice exercise HO-24

Solution

a) The function outputs all of the Hamming numbers. However, it does not return them in order: it "gets stuck" in an infinite loop only producing the powers of 2. Therefore, computations that test whether a relatively small number (for instance, smaller than 100) is a Hamming number may not terminate.

b)

```
hamming' :: Integral a => [a]
hamming' = 1:((map (2*) hamming') `merge` (map (3*) hamming') `merge` (map
    ↪  (5*) hamming'))

merge :: (Integral a) => [a] -> [a] -> [a]
merge (x:xs) (y:ys)
  | x < y = x:(xs `merge` (y:ys))
  | y < x = y:((x:xs) `merge` ys)
  | otherwise = x:(xs `merge` ys)
```

This solution uses the auxiliary function merge to combine infinite lists, using a similar logic to the one of merge sort (exercise LI-23). The main difference is that an additional case is used to remove duplicate values.

This correct version of hamming produces the results in the correct order thanks to Haskell's lazy evaluation. The `` `merge` `` operator is left-associative, so the infinite list is the result of merging the lists `(map (2*)hamming')` `` `merge` `` `(map (3*)hamming')` and `map (5*)hamming'`. Due to lazy evaluation, where computations are only performed only when needed, only the first element of merging `(map (2*)hamming')` with `(map (3*)hamming')` is produced, so that it is compared with the head of `map (5*)hamming'` to select the smallest of them. The computation then proceeds by only "extracting" the elements of the three infinite lists as they are needed.

The hamming' function works with recursive lists so that each Hamming number can be multiplied by any of the three integers 2, 3 and 5. This allows for Hamming numbers that are not "pure powers" of one of these integers to be produced.

4.5 Application and composition

Function application consists in passing an argument to a function. By putting spaces between a function's name and arguments, the latter are passed one-by-one, from left-to-right. Thus, by default, application is left-associative: $fxy = (fx)y$. Also, by default, application has a very high precedence (it is performed before other operations).

As an alternative, function application can be performed using the dollar sign (`$`). This function has the lowest precedence and is left associative. It is useful for writing expressions with less parentheses. Examples:

```
Prelude> succ (succ 1)
3
Prelude> succ succ 1
<interactive>:25:1: error:
* Non type-variable argument in the constraint: Enum (a -> a)
(Use FlexibleContexts to permit this)
* When checking the inferred type
it :: forall a. (Enum a, Enum (a -> a), Num a) => a
Prelude> succ $ succ 1
3
Prelude> (*3) $ succ $ (*8) 2
51
```

Function composition `g . f` is the operation of building a new function by passing the output of a function f as input to a function g: $(g \circ f)(x) = g(f(x))$. It allows one to create functions on-the-fly with a more concise syntax than with lambdas by combining functions. Composition is right-associative: $(h \circ g \circ f)(x) = (h \circ (g \circ f))(x) = h(g(f(x)))$. Examples:

```
Prelude> (head . tail) [1..5]
2
Prelude> (succ . (*8)) 4
33
```

Sample exercises

Sample exercise HO-25

Problem statement

a) Implement ap, which works similarly to Prelude's application operator `($)`.

b) Implement cm, which works similarly to Prelude's composition operator `(.)`.

Usage examples:

```
*Main> head `ap` tail [1,2,3]
2
*Main> (head `cm` tail) [1,2,3]
2
*Main> head `cm` tail `ap` [1,2,3]
2
```

Solution

a)

```
-- Alternative with infix definition
ap :: (a -> b) -> a -> b
f `ap` x = f x
-- Alternative with prefix definition
ap' :: (a -> b) -> a -> b
ap' f x = f x
```

The right part of `$` is applied to a function on its left side, thus the first argument of ap must be a function.

The function can be defined using infix notation, but must always be used using this notation (even if defined in prefix notation).

The definition correctly emulates the `$` since a prefix function has a higher precedence than an infix operation, as explained in the solution of exercise IN-1, in Chapter 1.

b)

```
-- Alternative with parentheses
cm :: (b -> c) -> (a -> b) -> (a -> c)
cm f g x = f(g x)
-- Alternative with $
cm' :: (b -> c) -> (a -> b) -> (a -> c)
cm' f g x = f $ g x
-- Alternative with a lambda
cm'' :: (b -> c) -> (a -> b) -> (a -> c)
cm'' f g = \x -> f(g x)
```

Sample exercise HO-26

Problem statement

Rewrite the following expressions to use as few parentheses ("()", not "[]") as possible.

a) `sqrt (1 + 2 + 3)`
b) `length ((map f)(head (reverse xs)))`
c) `drop 2 (tail [1..6])` (in two different ways)

Solution

a) `sqrt $ 1 + 2 + 3`
The `$` is your friend to remove a lot of parentheses.
b) The solution can be reached by removing each pair of unnecessary parentheses, one-by-one:

```
length ((map f)(head $ reverse xs))
length ((map f)$ head $ reverse xs)
length (map f $ head $ reverse xs)
length $ map f $ head $ reverse xs
```

c) First alternative using only $'s: `drop 2 $ tail [1..6]`
Second alternative using a composition, step-by-step:

```
((drop 2). tail)[1..6]
(drop 2). tail $ [1..6]
drop 2 . tail $ [1..6]
```

In this last alternative, the partially applied functions `drop 2` and `tail` are composed. A `$` is used between tail and [1..6] since prefix function application has a higher precedence than composition, so not including the `$` would force `drop 2` to be composed with `tail [1..6]`, the latter of which is not a function, and thus cannot be composed.
The last pair of parentheses can be safely removed without adding any $'s because composition has a lower precedence than prefix function applications.

Practice exercises

Practice exercise HO-27

Problem statement

When implementing Prelude's any function for exercise HO-12, a programmer tried to define it as follows: `myAny = or . map`
Explain briefly why this definition is incorrect.

Practice exercise HO-28

Problem statement

Rewrite the following expressions to use as few parentheses as possible.

a) `sum (map sqrt [1..10])`

b) `sum (filter (> 8)(map (*2)[2..14]))`

c) `sum (replicate 5 (min 1 3))` (in three different ways: with 0, 1 and 2 compositions)

d) `replicate 50 (sum (map (+3)(zipWith min [1..5] [4..8])))`

Practice exercise HO-29

Problem statement

Indicate the result of the following expressions:

a) `(all (>3). filter (>=3))[1,4..10]`

b) `map ($ 4)[(2+), (5*), (^3), sqrt]`

c) `filter (even . (!!2))[[1,2,3],[4,5,6],[8,3,5,8],[12,11,10]]`

d) `(filter even . (!!2))[[1,2,3],[4,5,6],[8,3,5,8],[12,11,10]]`

Solutions to the practice exercises

Practice exercise HO-27

Solution

The definition is wrong since the input of `or` must match the output of `map`. However, the input of `or` is a list, but the output of `map` is an unary function with the type `[a] -> [a]`, since functions in Haskell are curried.

If myAny is defined as `myAny p = or . map p` or `myAny p l = or . map p $ l`, then the function is correct since `map p` returns a list.

Practice exercise HO-28

Solution

a) `sum $ map sqrt [1..10]`

b) `sum $ filter (> 8)$ map (*2)[2..14]`

The parentheses for the sections cannot be removed.

c) First alternative: `sum $ replicate 5 $ min 1 3`

Second alternative: `sum . replicate 5 $ min 1 3`

Third alternative: `sum . replicate 5 . min 1 $ 3`

In this last alternative, the partially applied functions `replicate 5` and `min 1` are composed. A `$` is used between the 1 and 3 since prefix function application has a higher precedence than composition, so not including the `$` would force `replicate 5` to be composed with `min 1 3`, the latter of which is not a function, and thus cannot be composed. The `$` cannot be placed between min and 1, because `replicate 5` does not expect a function, but instead a value it can replicate (all arguments are applied to replicate except for the last one).

d) First alternative: `replicate 50 $ sum $ map (+3)$ zipWith min [1..5] [4..8]`

Second alternative: `replicate 50 . sum $ map (+3)$ zipWith min [1..5] [4..8]`

Third alternative: `replicate 50 . sum . map (+3)$ zipWith min [1..5] [4..8]`

Fourth alternative: `replicate 50 . sum . map (+3). zipWith min [1..5] $ [4..8]`

In the last alternative, the `$` has to be placed between the two lists, so that `zipWith min [1..5]` return a list, rather than a function (all arguments are applied to zipWith except for the last one).

Practice exercise HO-29

Solution

a) `True`

The expression begins by applying a filter to keep only the elements of the range that are greater or equal than 3. Then, it checks if all elements are strictly greater than 3. The result is `True`, since 3 is not included in the range.

b) `[6.0,20.0,64.0,2.0]`

The `map` applies the value 2 to each function in the list.

c) `[[4,5,6],[12,11,10]]`

This expression only keeps the sublists of the input 2D list whose third element (not second, since indices start at 0 for `(!!)`) is even.

d) `[8,8]`

This expression is different from the last one! `filter even` is composed with `(!!2)`. Therefore, the expression first selects the third element of the 2D list, then it keeps those that are even.

4.6 Folds

Folds are a family of higher order functions that process a data structure in a given order and return a value. Typically (but not always), the data structure being processed is a list.

Folds have usually two ingredients: a combining function and an accumulator. The combining function is a binary function that receives as input the accumulator and an element of the data structure. The two main folding functions are `foldr` and

`foldl` , which work on lists. They receive three arguments (in order): the combining function, the initial value of the accumulator and the list.

The accumulator must be of the same type as the return value type of the fold. It initial value is usually the identity/neutral element of the combining function. For example: for addition and multiplication, 0 and 1 are commonly used; for constructing lists, the initial value is usually the empty list `[]` .

`foldr` performs *right folds*: it recursively combines the result of the list's head and accumulator with the result of combining with the tail.

`foldl` performs *left folds*: it recursively combines the result of combining all but the list's last element and accumulator with the last element.

Using left or right folds gives the same result if the operation of the combining f function is associative: $f(fab)c = fa(fbc)$.

The following example presents the difference between left and right folds using the subtraction operation:

```
Prelude> foldr (-) 0 [1..5]
3
Prelude> foldl (-) 0 [1..5]
-15
```

When folding lists, one can consider that the operation consists of replacing the comma in a list (which separates two consecutive elements) by the symbol of the operator. According to the 'direction' of the fold, the initial value of the accumulator is either the rightmost or leftmost operand of the expression.

```
foldr (-) 0 [1, 2, 3, 4, 5] = (1 - (2 - (3 - (4 - (5 - 0))))) = 3
foldl (-) 0 [1, 2, 3, 4, 5] = (((((0 - 1) - 2) - 3) - 4) - 5) = -15
```

Folds have the advantage of allowing for more compact code, relative to recursive functions.

Two useful sets of variants exist: fold1's and scan's. `foldr1` and `foldl1` assume that the initial value for the accumulator is the last/first element of the list and perform a right/left fold on the remaining elements of the list (respectively). Thus, they only receive as arguments a combining function and a list. `scanr` and `scanl` work mostly like `foldr` / `foldl` but instead return a list with all the intermediate values of the computations. Examples:

```
Prelude> foldr1 (-) [1..5]
3
Prelude> foldl1 (-) [1..5]
-13
Prelude> scanr (-) 0 [1..5]
[3,-2,4,-1,5,0]
Prelude> scanl (-) 0 [1..5]
[0,-1,-3,-6,-10,-15]
```

Sample exercises

Sample exercise HO-30

Problem statement

Consider myReverse, which, similarly to Prelude's reverse, given a list, returns a new list with its elements in reverse order.

a) Implement myReverse using foldr.

Note: A solution with $O(n^2)$ temporal complexity is acceptable (n being the length of the input list).

b) Implement myReverse using foldl.

Note: The solution should have a $O(n)$ temporal complexity.

Usage examples:

```
*Main> myReverse [1..5]
[5,4,3,2,1]
```

Solution

a)

```
myReverse' :: [a] -> [a]
myReverse' l = foldr (\x acc -> acc ++ [x]) [] l
```

The initial value for the accumulator is `[]` since the reverse of the empty list is itself. The combining function adds the list element being currently processed, x, to the end of the accumulator acc, to form the new value for the accumulator.

In right-folds, the order of the arguments in the combining function is x-acc. In **right**-folds, the combining function places the accumulator acc to the **right**.

b)

```
myReverse :: [a] -> [a]
myReverse l = foldl (\acc x -> x:acc) [] l
```

Implementing this function with foldl is much more natural, as a simple $O(n)$ time solution can be derived. The cons operator (`:`) is used to append the list element currently being processed, x, to the beginning of the accumulator acc, to form the new value for the accumulator.

In left-folds, the order of the arguments in the combining function is acc-x. In **left**-folds, the combining function places the accumulator acc to the **left**.

Sample exercise HO-31

Problem statement

Using a fold, implement myFilter, which works similarly to Prelude's filter.

Usage examples:

```
*Main> myFilter (> 3) [1..5]
```

```
[4,5]
```

Solution

```
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter f l = foldr (\x acc -> if f x then x:acc else acc) [] l
```

Right-folds are usually preferred when the output is a list since they keep the input's list elements in order. Left-folds, as seen in the previous problem, are more convenient to reverse the order of a list.

Practice exercises

Practice exercise HO-32

Problem statement

Using a fold, implement myMap, which works similarly to Prelude's map.

Usage examples:

```
*Main> myMap (*2) [1..5]
[2,4,6,8,10]
```

Practice exercise HO-33

Problem statement

Using a fold, implement largePairs, which, given a minimum threshold m and a list of pairs l, returns a new list where the pairs whose sum is below m is removed.

Usage examples:

```
*Main> largePairs 4 [(1,2),(10,-5),(4,5),(-30,20)]
[(10,-5),(4,5)]
```

Practice exercise HO-34

Problem statement

Using a fold, implement fuseDigits, which, given a list of digits l, returns a positive integer composed of the digits in l.

Usage examples:

```
*Main> fuseDigits [1,6,4,5,2]
16452
```

Practice exercise HO-35

Problem statement

Using a fold, implement separateSingleDigits, which, given a list of positive integers l, returns a pair where the first component contains a list with the numbers with a single digit, and in the second component, a list with the remaining ones. The numbers in both output lists should appear in the same order as in input list l.

Usage examples:

```
*Main> separateSingleDigits  [100,55,40,2,11,5,10,7,8]
([2,5,7,8],[100,55,40,11,10])
```

Practice exercise HO-36

Problem statement

Consider the definition of function f below:

```
f :: [a] -> [a] -> [a]
f xs ys = foldr (\x acc -> x:acc) ys xs
```

Explain concisely what function f computes.

Practice exercise HO-37

Problem statement

a) Implement myFoldr, which works similarly to Prelude's foldr.

b) Implement myFoldl, which works similarly to Prelude's foldl.

Usage examples:

```
*Main> myFoldr (-) 0 [1..5]
3
*Main> myFoldl (-) 0 [1..5]
-15
```

Practice exercise HO-38

Problem statement

Consider myFoldr1, which, similarly to Prelude's foldr1, applies foldr on a non-empty list and uses the list's last element as the initial value for the accumulator. As a result, foldr1 does not use a value for the initial value of the accumulator. It receives as arguments a folding function f and a list l.

If list l is empty, the function should issue the error "empty list".

a) Implement myFoldr1 using recursion (and without using user higher-order functions).

b) Implement myFoldr1 using foldr and without using recursion.

Usage examples:

```
*Main> myFoldr1 (-) [1..5]
3
```

Practice exercise HO-39

Problem statement

Consider myFoldl1, which, similarly to Prelude's foldl1, is the equivalent of foldr1 for left-folds.

a) Implement myFoldl1 using recursion (and without using user higher-order functions).

Suggestion: For the recursive case, work with the first two elements of the list, rather than just the head.

b) Implement myFoldl1 using foldl and without using recursion.

Usage examples:

```
*Main> myFoldl1 (-) [1..5]
-13
```

Practice exercise HO-40

Problem statement

Consider myScanr, which, similarly to Prelude's scanr, applies foldr to a list and outputs a list with all the intermediate values for the accumulator. The head of the output contains the final value of foldr, while the last element contains the initial value of the accumulator.

a) Implement myScanr using recursion (and without using user higher-order functions).

Suggestion: Use a where clause.

b) Implement myScanr using foldr and without using recursion.

Usage examples:

```
*Main> myScanr (-) 0 [1..5]
[3,-2,4,-1,5,0]
```

Practice exercise HO-41

Problem statement

Consider myScanl, which, similarly to Prelude's scanl, is the equivalent of scanr for left-folds.

a) Implement myScanl using recursion (and without using user higher-order functions).

b) Implement myScanl using foldl and without using recursion.

Suggestion: Use reverse.

Usage examples:

```
*Main> myScanl (-) 0 [1..5]
[0,-1,-3,-6,-10,-15]
```

Practice exercise HO-42

Problem statement

Implement myFoldl, which works just like Prelude's foldl, using foldr.

Suggestion: The call to the mandatory foldr function should return a new unary function. Here are some questions to guide your thought process:

- How many arguments should the lambda function supplied to foldr have?
- Which should be the order of the arguments within the lambda function? Think back about how currying works.
- What value could be used for initial value of the accumulator, given that it should be a function? What function could serve as a neutral element to function application?

Practice exercise HO-43

Problem statement

Consider the kadane function, which implements Kadane's algorithm to solve the maximum sub-array problem.

The goal of the maximum sub-array problem is to determine a contiguous sub-array with the largest sum: given an array L with n integers, the goal is to find indices i and j such that $0 \leq i \leq j < n$ (with indices starting at 0), so that the following sum is maximized:

$$\sum_{x=i}^{j} L[x]$$

In this problem, the goal is just to determine the sum, not i and j. In case all of the integers are negative, the solution is 0 (subarray of length 0).

Kadane's algorithm is a $O(n)$ time method to solve this problem, which computes an auxiliary array A of length (n+1), where $A[0]$ is 0 and $A[i]$ is the largest sum whose sub-array finishes in index (i-1):

$$A[i] = \begin{cases} 0 & i = 0 \\ max(L[i-1], A[i-1] + L[i-1]) & 1 \leq i \leq n \end{cases}$$

The solution is the largest value in A.

Using a fold (or a fold1 or scan function) and without recursion, implement kadane.

Usage examples:

```
*Main> kadane [-1,-2,-3]
0
*Main> kadane [-1,-4,5,-1,-2,1,6,-2]
9
```

Solutions to the practice exercises

Practice exercise HO-32

Solution

```
myMap :: (a -> b) -> [a] -> [b]
myMap f l = foldr(\x acc -> (f x):acc) [] l
```

Just like when implementing filter using folds, it is more convenient to use a right-fold to keep the order of elements of the input.

Practice exercise HO-33

Solution

```
largePairs :: (Ord a, Num a) => a -> [(a, a)] -> [(a, a)]
largePairs max ts = foldr(\(x,y) acc -> if (x + y > max) then (x,y):acc
    ↪ else acc) [] ts
```

This function works similarly to a filter so using a right-fold is more natural.

A pattern is used in the combining function to read list elements that consist of pairs.

Practice exercise HO-34

Solution

```
-- Alternative with foldl
fuseDigits :: (Integral a) => [a] -> a
fuseDigits l = foldl (\acc x -> 10*acc + x) 0 l
-- Alternative with foldr
fuseDigits' :: (Integral a) => [a] -> a
fuseDigits' = fst . foldr (\x (total,mult10) -> (x*mult10 + total,10*
    ↪ mult10)) (0,1)
```

Using a left-fold is more natural since one can simply multiply by 10 the previous value of the accumulator in the combining function so that the element currently being processed, x, can be added to produce the next value for the accumulator.

Using a right-fold leads to a harder implementation of the problem: when combining the next digit, one needs to keep the digit's position in the final number / the power of 10 to multiply with x. To keep the information of the digit's position, the accumulator is changed into a pair, so that it can store both the running total of the number and the power of 10 to be multiplied.

Practice exercise HO-35

Solution

```
separateSingleDigits :: (Integral a) => [a] -> ([a], [a])
separateSingleDigits l = foldr(\x (ys,ns) -> if (x >= 0 && x <= 9) then (x
    ↪ :ys,ns) else (ys,x:ns)) ([],[]) l
```

The combining function uses a pair for the accumulator since its type must match the output of the fold.

Practice exercise HO-36

Solution

Function f appends two lists xs and ys.

Practice exercise HO-37

Solution

a)

```
myFoldr :: (a -> b -> b) -> b -> [a] -> b
myFoldr _ acc [] = acc
```

```
myFoldr f acc (x:xs) = f x (myFoldr f acc xs)
```

It is typical in right-folding functions (foldr, foldr1, scanr...) for the function with the lowest precedence to be the combining function f.

b)

```
myFoldl :: (b -> a -> b) -> b -> [a] -> b
myFoldl _ acc [] = acc
myFoldl f acc (x:xs) = myFoldl f (f acc x) xs
```

It is typical in left-folding functions (foldl, foldl1, scanl...) for the function with the lowest precedence to be the left-folding function itself.

The base case of foldl is the same as the one for foldr.

Practice exercise HO-38

Solution

a)

```
myFoldr1 :: (a -> a -> a) -> [a] -> a
myFoldr1 _ [] = error "empty␣list"
myFoldr1 _ [x] = x
myFoldr1 f (x:xs) = f x (myFoldr1 f xs)
```

The implementation is very similar to the one for foldr and foldl.

b)

```
myFoldr1' :: (a -> a -> a) -> [a] -> a
myFoldr1' _ [] = error "empty␣list"
myFoldr1' f l = foldr f (last l) (init l)
```

This version is less efficient since the list is iterated three times: one for foldr, another for last and another one for init. However, the temporal complexity is still $O(n)$, where n is the length of the input list.

Practice exercise HO-39

Solution

a)

```
myFoldl1 :: (a -> a -> a) -> [a] -> a
myFoldl1 _ [] = error "empty␣list"
myFoldl1 _ [x] = x
myFoldl1 f (x:y:xs) = myFoldl1 f ((f x y):xs)
```

This implementation is quite tricky, since the program must reason with the first two elements of the list, unlike foldr1 and foldl.

b)

```
myFoldl1' :: (a -> a -> a) -> [a] -> a
myFoldl1' _ [] = error "empty␣list"
myFoldl1' f (x:xs) = foldl f x xs
```

This version is much more straightforward to implement than the recursive version and the implementation of foldr1 with foldr.

Practice exercise HO-40

Solution

a)

```
myScanr :: (a -> b -> b) -> b -> [a] -> [b]
myScanr _ acc [] = [acc]
myScanr f acc (x:xs) = (f x (head t)):t
    where t = myScanr f acc xs
```

A where clause is used to avoid performing the recursive call twice.

b)

```
myScanr' :: (a -> b -> b) -> b -> [a] -> [b]
myScanr' f acc l = foldr (\x (h:t) -> (f x h):h:t) [acc] l
```

This version also separates the accumulator (which is equivalent to the output of a recursive call) into its head and tail.

Practice exercise HO-41

Solution

a)

```
myScanl :: (a -> b -> a) -> a -> [b] -> [a]
myScanl _ acc [] = [acc]
myScanl f acc (x:xs) = acc:(myScanl f (f acc x) xs)
```

b)

```
-- More efficient alternative with reverse
myScanl' :: (a -> b -> a) -> a -> [b] -> [a]
myScanl' f acc l = reverse (foldl (\(h:t) x -> (f h x):h:t) [acc] l)

-- Equivalent alternative to the previous one that uses a variable for the
    ↪ folding function's accumulator
myScanl'' :: (a -> b -> a) -> a -> [b] -> [a]
myScanl'' f acc l = reverse (foldl (\subAcc@(h:_) x -> (f h x):subAcc) [
    ↪ acc] l)

-- Less efficient alternative that uses (++) and last in each call to the
    ↪ combining function
myScanl''' :: (a -> b -> a) -> a -> [b] -> [a]
myScanl''' f acc l = foldl (\subAcc x -> subAcc ++ [f (last subAcc) x]) [
    ↪ acc] l
```

In the first alternative, the foldl buils the list in reverse order (the last element of the list will be the initial value of the scanl's accumulator) since it is more efficient to create a new list by combining a new element (the new list's head) with an previously created list (the new list's tail). It runs in $O(n)$ time.

The last alternative removes the need for the reverse at the expense of running in $O(n^2)$ time, due to calling `(++)` and `last` in each call of the combining function.

Practice exercise HO-42

Solution

Given that this problem is quite challenging, the questions from the problem statement that were given to help out will be answered first:

- The function to be returned is unary and the function which is usually supplied to a fold is binary. Since the type `a -> b -> (c -> d)` (a binary function that returns a unary function) is equivalent to `a -> b -> c -> d` (a ternary function), the lambda function to be defined in this challenge will have three arguments.
- Given the two function types presented in the previous point, the "extra" argument will be the "rightmost"/"third" argument of the lambda.
- The neutral element for function application is the identity function, an unary function which returns its single input. Predule already contains this function, `id`, which will be used for the initial value of the accumulator.

Given these three clues, the challenge was to come up with a way of preventing the function to be folded, f, from being applied to the initial value for the accumulator, `id`, and the list's last element as is usually performed by right folds, since one wants to mimic the operation of a left fold.

To begin the definition of the lambda, one can think about the expression for left folds: $f(f(fza)b)c$, where z is the initial value of the accumulator.

```
myFoldl :: (t2 -> t1 -> t2) -> t2 -> [t1] -> t2
myFoldl f acc l =
    foldr (\x accF z ->  f (accF z) x) id l acc
-- Alternative solution
myFoldl' :: (t2 -> t1 -> t2) -> t2 -> [t1] -> t2
myFoldl' f acc l =
    foldr (\x accF z ->  accF (f z x)) id l acc
```

Practice exercise HO-43

Solution

```
kadane :: (Ord a, Num a) => [a] -> a
kadane = maximum . scanl (\acc x -> max x (acc + x)) 0
```

The scanl function is used to produce the A array described in the problem statement. A left scan was preferred over a right scan to match the strategy

adopted on Kadane's algorithm of processing elements by starting with the first one.

4.7 Point-free style

One interesting way of programming in Haskell is by writing functions in *point-free style* (also known as *point-less style* and *tacit programming*), where the arguments (points) of the function are omitted from its definition.

The main tools to program in point-free style are using composition and other higher-order functions (namely maps, filters and folds), rather than application. Since composition is one of the main functions in point-free style, definitions actually use more dots/points.

The main advantage of this paradigm is that (usually) function definitions are more readable, elegant and concise. It is a high-level programming paradigm where one thinks about how to combine functions to achieve the desired output rather than reasoning directly about the data (the function's arguments) and how it is exchanged between functions.

Sample exercises

Sample exercise HO-44

Problem statement

In point-free style, implement capitalize, which given a string s, returns a new string where all lowercase letters are converted to upper case.

You are allowed to use the toUpper function from the Data.Char module.

Usage examples:

```
*Main> capitalize  "hello, world!"
"HELLO, WORLD!"
```

Solution

```
import Data.Char
capitalize :: [Char] -> [Char]
capitalize = map toUpper
```

`map` is used to apply `toUpper` to each element of the string. The function is defined in point-free style by leaving `map` partially applied: only the functional

argument is provided, while the list is left to be applied by those who call `capitalize .`

This solution shows an example of how to write an unary function in point-free style.

Sample exercise HO-45

Problem statement

Implement myElem, which, similarly to Prelude's elem, returns a boolean indicating if an element belongs to a list.

Usage examples:

```
*Main> myElem 5 [1..10]
True
*Main> myElem 11 [1..10]
False
```

Solution

```
myElem :: (Eq a) => a -> [a] -> Bool
myElem = any . (==)
```

In this solution, `any` is composed with `==` since the first argument of the former function is a predicate, and `==` also return a predicate (due to currying), so the types match for the composition.

This solution shows an example of how to write a binary function in point-free style.

Note: lambdas cannot be used in point-free style, since they give names to their internal arguments.

Sample exercise HO-46

Problem statement

Consider the following function, which, given a list of numbers, returns a new list with their squares:

```
squareAll :: Num a => [a] -> [a]
squareAll l = map (\x -> x^2) l
```

Rewrite the function in point-free style.

Solution

Two parts of the function do not respect the point-style: the arguments of squareAll and the usage of a lambda, which also has named arguments.

The conversion to point-free style can be performed in two separate steps to make the process more clear. Firstly, the lambda can be converted to an operator section:

```
squareAll' :: Num a => [a] -> [a]
squareAll' l = map (^2) l
```

Finally, squareAll's argument can be directly removed by erasing the `l` at the end of squareAll's argument list and at the end of squareAll's body. This removal is known as *eta-reduction* or η*-reduction*.

A point-free version of squareAll is:

```
squareAll'' :: Num a => [a] -> [a]
squareAll'' = map (^2)
```

Practice exercises

Practice exercise HO-47

Problem statement

Consider the mystery function f provided in exercise HO-36. Rewrite the function in point-free style.

Suggestion: consider using the flip function. flip is a function from Prelude that receives a binary function as input and switches the order of its arguments. It has the following type declaration:

```
flip :: (a -> b -> c)-> (b -> a -> c) .
```

Practice exercise HO-48

Problem statement

In point-free style and in two different ways, implement myLast, which works similarly to Prelude's last.

Suggestion: in one of the alternatives, consider using the const function along with a fold. const is a binary function from Prelude that returns its first argument. It has the following type declaration:

```
const :: a -> b -> a .
```

Usage examples:

```
*Main> myLast [1..5]
5
```

Practice exercise HO-49

Problem statement

a) In point-free style, implement countLetters, which, given a string composed of some words separated by single spaces, returns the number of letters in the text.

b) In point-free style, implement countFirst, which, given a string composed of some words separated by single spaces, returns the number of letters in the first word.

Suggestion: consider using the words function. words is an unary function from Prelude that splits a text into a list of substrings (with white spaces serving as separators). It has the following type declaration:

```
words :: [Char] -> [[Char]] .
```

Usage examples:

```
*Main> countLetters "hello␣world"
10
*Main> countLetters "hello␣world␣i␣am␣here"
17
*Main> countFirst "hello␣world"
5
*Main> countFirst "hello␣world␣i␣am␣here"
5
```

Practice exercise HO-50

Problem statement

In point-free style and in two different ways, implement myReverse, which works similarly to Prelude's reverse.

Usage examples:

```
*Main> myReverse [1..5]
[5,4,3,2,1]
```

Practice exercise HO-51

Problem statement

Implement the following list functions in point-free style.

a) mySum, which is similar to Prelude's sum.

b) myProduct, which is similar to Prelude's product.

c) myLength, which is similar to Prelude's length.

Usage examples:

```
*Main> mySum [1..5]
```

```
15
*Main> myProduct [1..5]
120
*Main> myLength [1..5]
5
*Main> myLength [2..5]
4
```

Practice exercise HO-52

Problem statement

In point-free style, implement extractDigits, which given a positive integer n, outputs a list with the digits of n.

Usage examples:

```
*Main> extractDigits 1
[1]
*Main> extractDigits 1230
[1,2,3,0]
```

Practice exercise HO-53

Problem statement

In point-free style, implement kadane, which was presented in exercise HO-43.

Usage examples:

```
*Main> kadane [-1,-2,-3]
0
*Main> kadane [-1,-4,5,-1,-2,1,6,-2]
9
```

Solutions to the practice exercises

Practice exercise HO-47

Solution

To begin with, the lambda can be replaced by a section:

```
f' :: [a] -> [a] -> [a]
f' xs ys = foldr (:) ys xs
```

Finally, to remove the explicit arguments from f, one cannot perform two direct eta-reductions since the xs and ys appear in reverse order at the end of the expression. To overcome this, one can simply flip the arguments of the partially applied function `foldr (:)` using `flip`:

```
f'' :: [a] -> [a] -> [a]
f'' = flip (foldr (:))
```

Practice exercise HO-48

Solution

```
-- Alternative with composition
myLast :: [a] -> a
myLast = head . reverse
-- Alternative with foldl and const
myLast' :: [a] -> a
myLast' = foldl (flip const) (error "empty list")
-- Alternative with foldl1 and const
myLast'' :: [a] -> a
myLast'' = foldl1 (flip const)
-- Alternative with foldr1 and const
myLast''' :: [a] -> a
myLast''' = foldr1 (flip const)
```

foldl1 and foldr1 can be safely used since the function is not supposed to be called on empty lists.

In the fold-based alternatives, by using flip, the function returns its second argument instead, emulating a lambda that keeps the original value of the accumulator (for foldr1) and the last element that was visited in the list (for foldl1).

Practice exercise HO-49

Solution

a)

```
-- Alternative with map
countLetters :: [Char] -> Int
countLetters = sum . map length . words
-- Alternative with concat
countLetters' :: [Char] -> Int
countLetters' = length . concat . words
```

In the first solution, there is no composition (.) between map and length so that the output of words (a list) matches the output of the second argument of map (i.e. the first and only argument of the partially applied version of map being used, `map length`).

b)

```
-- Alternative with head
countFirst :: [Char] -> Int
countFirst = length . head . words
-- Alternative with takeWhile
countFirst' :: [Char] -> Int
countFirst' = length . takeWhile (/= ' ')
```

Practice exercise HO-50

Solution

Using left and right-folds, one could write the following functions in pointful style (i.e. not in point-free style):

```
-- Pointful version with foldl
myReverse'' :: [a] -> [a]
myReverse'' = foldl (\acc x -> x:acc) []
-- Pointful version with foldr
myReverse''' :: [a] -> [a]
myReverse''' = foldr (\x acc -> acc ++ [x]) []
```

In point-free style, the result is:

```
-- Alternative with foldl
myReverse :: [a] -> [a]
myReverse = foldl (flip (:)) []
-- Alternative with foldr
myReverse' :: [a] -> [a]
myReverse' = foldr (flip (++)) []
```

Practice exercise HO-51

Solution

a)

```
mySum :: (Num a) => [a] -> a
mySum = foldl (+) 0
```

b)

```
myProduct :: (Num a) => [a] -> a
myProduct = foldl (*) 1
```

c)

```
-- Alternative with map
myLength :: (Integral b) => [a] -> b
myLength = sum . map (const 1)
-- Alternative with foldr
myLength' :: (Integral b) => [a] -> b
myLength' = foldr (const succ) 0
```

The second alternative using folds was derived using the following steps:

```
-- Pointful version of the alternative with foldr
```

```
myLength'' :: (Integral b) => [a] -> b
myLength'' = foldr (\x acc -> succ acc) 0
-- Pointful version of the alternative with foldr after eta-reduction
myLength''' :: (Integral b) => [a] -> b
myLength''' = foldr (\x -> succ) 0
```

Practice exercise HO-52

Solution

```
extractDigits :: Integer -> [Integer]
extractDigits =
    reverse
    . map (`mod` 10)
    . takeWhile (> 0)
    . iterate (`div` 10)
```

Step-by-step explanation using the example of input n = 1230:

- `iterate (`div` 10)` : Obtain a list with the successive divisions of n by powers of 10: [1230,123,12,1,0,0...].
- `takeWhile (> 0)` : Remove the trailing 0's: [1230,123,12,1].
- `map (`mod` 10)` : Extract the rightmost digit of each list element: [0,3,2,1].
- `reverse` : Reverse the result: [1,2,3,0].

Practice exercise HO-53

Solution

Starting with initial solution below, a step-by-step conversion to a point-free version of kadane will be presented:

```
kadane'1 :: (Ord a, Num a) => [a] -> a
kadane'1 = maximum . scanl (\acc x -> max x (acc + x)) 0
```

Swapping the order of (+) and max:

```
kadane'2 :: (Ord a, Num a) => [a] -> a
kadane'2 = maximum . scanl (\acc x -> (+) (max 0 acc) x) 0
```

Partially applying (+) to create an operator section:

```
kadane'3 :: (Ord a, Num a) => [a] -> a
kadane'3 = maximum . scanl (\acc x -> (+ (max 0 acc)) x) 0
```

Eta-reduction of x:

```
kadane'4 :: (Ord a, Num a) => [a] -> a
kadane'4 = maximum . scanl (\acc -> (+ (max 0 acc))) 0
```

Composition of (+) and (max 0):

```
kadane'5 :: (Ord a, Num a) => [a] -> a
kadane'5 = maximum . scanl (\acc -> ((+).(max 0)) acc) 0
```

Eta-reduction of acc:

```
kadane :: (Ord a, Num a) => [a] -> a
kadane = maximum . scanl ((+).(max 0)) 0
```

Chapter 5
User-defined types

Being statically and strongly typed, Haskell puts a lot of emphasis on types. Despite the language's rich collection of built-in types, a programmer might prefer to define their own data types to create new abstractions and define new constraints on the interpretation of the data.

This chapter begins by presenting the `type` and `data` keywords, aimed to define new type synonyms and data types, respectively. Derived data types and named fields are then introduced. Then, the chapter explains how modules can be created and imported to define and use sets of functions/types that are logically related. Finally, three case studies are presented to illustrate complex and useful custom data types: syntax trees, binary search trees and AVL trees.

5.1 Creating type synonyms with the `type` keyword

The `type` keyword can be used to define *type synonyms*. These synonyms have the advantage of increasing the readability of Haskell code by providing syntactic sugar.

The general syntax to define type synonyms is:

```
type <synonym name> <type variable 1> <type variable 2> ... = <expression>
```

Just like a type's name, a synonym's name must start with an uppercase letter.

Type variables can be used in synonyms to allow for more generic synonyms. Synonyms can also be used in the definition of other synonyms. However, synonyms cannot have recursive definitions (i.e. a synonym's name cannot appear in its expression). Examples:

```
type String = [Char]
type Pair a = (a,a)
type HashMap k v = [(k,v)]
```

The first example is from the Prelude.

Type synonyms can be used when declaring the type of other functions or data structures.

Sample exercises

Sample exercise UT-1

Problem statement

Consider the definition of `HashMap` below, which represents a hash map between a key with type k and a value with type v:

```
type HashMap k v = [(k,v)]
```

Assume that all of the elements in a hash map contain a unique key.

a) Using AssocList, define a type synonym `IntMap` where the key is an Int.

b) Define a variable with type `IntMap` and a variable of type `HashMap` that is not an `IntMap`.

c) Define member, which, given a hash map m and a key v, determines if there is an element with key k.

d) Define keySum, given a hash map with integer keys m, returns the sum of all the keys in m.

Usage examples:

```
*Main> member "Bruno" mapString
True
*Main> member "John" mapString
False
*Main> keySum mapInt
6
```

Solution

a)

```
type IntMap v = HashMap Int v
```

The type variable a is applied to the type synonym `HashMap`.

b)

```
mapInt :: IntMap Char
mapInt = [(1,'a'),(2,'b'),(3,'c')]
mapString :: HashMap String Bool
mapString = [("Ana",True),("Sara",True),("Bruno",False)]
```

For the mapInt variable, `Int` is added to the right of the `IntMap` on the type declaration, not next to the name mapInt in the following line.

`IntMap` and `HashMap` are type constructors. `IntMap Char` and `HashMap String Bool` are concrete types.

c)

```
member :: (Eq k) => k -> HashMap k v -> Bool
member _ [] = False
member x ((k,v):kvs)
    | x == k = True
    | otherwise = member x kvs
```

This function does not make assumptions about the type of the key, so member was defined over type `HashMap` rather than `IntMap`.

d)

```
keySum :: IntMap v -> Int
keySum [] = 0
keySum ((k,v):kvs) = k + keySum kvs
```

Sample exercise UT-2

Problem statement

In a regular 8x8 chessboard, a rook can move any number of cells in a line or row.

a) Define a type synonym `Pos` that represents a position in a chessboard as a pair of integers between 1 and 8. The first element is the row index, the second one is the column index.

b) Define rookMoves, which given the initial position of a rook, returns a list with all the possible positions the rook can move to.

Usage examples:

```
*Main> rookMoves (2,8)
[(1,8),(3,8),(4,8),(5,8),(6,8),(7,8),(8,8),(2,1),(2,2),(2,3),(2,4),(2,5)
    ↪ ,(2,6),(2,7)]
```

Solution

a)

```
-- a
```

`Pos` is simply a pair of Int's. As an alternative, one could define `Pos` as `(Integer,Integer)`.

b)

```
rookMoves :: Pos -> [Pos]
rookMoves (row,col) =
    [(row',col) | row' <- [1..8], row' /= row] ++
    [(row,col') | col' <- [1..8], col' /= col]
```

The rook's legal movements are defined as the concatenation of two list comprehensions: one for all the legal movements in the vertical direction, another for all the moves in the horizontal direction.

Practice exercises

Practice exercise UT-3

Problem statement

Consider the definition of `Pair` below, which represents a pair of elements of the same type that are related:

```
type Pair a = (a,a)
```

a) Using `Pair`, define the type synonym `Relation` which represents a list of pairs. This types encodes a binary relation R.

b) Define isReflexive which, given a binary relation R, determines whether the relation is reflexive: for all elements x and y if $(x, y) \in R$ then $(y, x) \in R$.

c) Define isTransitive which, given a binary relation R, determines whether the relation is transitive: for all elements x, y and z, if $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$.

Usage examples:

```
*Main> let r = [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2)]
*Main> let r2 = [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
*Main> r
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2)]
*Main> r2
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]
*Main> isReflexive r
True
*Main> isReflexive r2
True
*Main> isReflexive [(1,2)]
False
*Main> isTransitive r
False
*Main> isTransitive r2
True
```

Practice exercise UT-4

Problem statement

Consider the definition of the type `Pos` described in exercise UT-2.

In a regular 8x8 chessboard, a queen can move any number of cells in a line, row or diagonal.

Define queenMoves, which given the initial position of a queen, returns a list with all the possible positions the queen can move to.

Usage examples:

```
*Main> queenMoves (2,8)
[(1,8),(3,8),(4,8),(5,8),(6,8),(7,8),(8,8),(2,1),(2,2),(2,3),(2,4),(2,5)
    ↪ ,(2,6),(2,7),(1,7),(3,7),(4,6),(5,5),(6,4),(7,3),(8,2)]
*Main> queenMoves (5,4)
[(1,4),(2,4),(3,4),(4,4),(6,4),(7,4),(8,4),(5,1),(5,2),(5,3),(5,5),(5,6)
    ↪ ,(5,7),(5,8),(1,8),(2,7),(3,6),(4,5),(6,3),(7,2),(8,1),(2,1),(3,2)
    ↪ ,(4,3),(6,5),(7,6),(8,7)]
```

Solutions to the practice exercises

Practice exercise UT-3

Solution

a)

```
type Relation a = [Pair a]
```

The type variable is applied to the type synonym `Pair`.

b)

```
isReflexive :: (Eq a) => Relation a -> Bool
isReflexive r = isReflexiveAux r r

isReflexiveAux :: (Eq a) => [Pair a] -> Relation a -> Bool
isReflexiveAux [] _ = True
isReflexiveAux ((x,y):xs) r
    | (y,x) `elem` r = isReflexiveAux xs r
    | otherwise = False
```

One could also define that the type of the first argument of the auxiliary method `isReflexiveAux` is `Relation a`. However, that is not semantically correct, since the first argument represents the binary pairs that are yet to be examined, while the second argument contains the whole list, so that one can lookup for specific pairs to test for reflexivity.

c)

```
isTransitive :: (Eq a) => Relation a -> Bool
isTransitive r = isTransitiveAux [(x,y) | x <- r, y <- r] r

isTransitiveAux :: (Eq a) => [Pair (Pair a)] -> Relation a -> Bool
isTransitiveAux [] _ = True
isTransitiveAux (((x,y),(w,z)):xs) r
    | y == w && not((x,z) `elem` r) = False
    | otherwise = isTransitiveAux xs r
```

The first argument of the auxiliary method `isTransitiveAux` contains all possible combinations of pairs in relation r. Thus, it can be defined as a list of pairs made of pairs of type a.

Practice exercise UT-4

Solution

```
queenMoves :: Pos -> [Pos]
queenMoves (row,col) =
    rookMoves (row,col) ++
    [(row',col') | dir <- [-1,1], offset <- [-7,-6.. -1] ++ [1..7],
        let row' = row+offset, let col' = col+offset*dir,
        row' >= 1, row' <= 8, col' >= 1, col' <= 8]
```

Since the queen's movements are a subset of the ones for a rook, the resulting list can include a call to the rookMoves function.

The additional comprehension list uses two generators: dir to control if the queen moves along a diagonal that is parallel to the board's primary or secondary diagonal, and offset to control the horizontal distance of the movement. The guards ensure that the position is inside the board. Let expressions are used to avoid repeating the expressions to compute the resulting positions in the guards.

5.2 Creating algebraic data types with the `data` keyword

If one wants to define a structure for a person with two strings: one with their name and another one with their email. The following definition could be used:

```
type Person = (String,String)
```

However, the declaration above does not allow one to distinguish a person from any other pair composed of two strings. Consider the definition below:

```
type CountryCapital = (String,String)
```

If one defines a function that receives as input a pair representing a country and its capital (as denoted by the CountryCapital type synonym) one could also pass a Person as input, which is semantically wrong, even though it is syntactically correct.

The `data` keyword circumvents this problem by defining new algebraic data types. Creating new data types allows for better structured code, readability and improves type safety.

The general syntax to define new types is:

```
data <type name> <type variable 1> <type variable 2> ... =
        <value constructor 1> <type 1> <type 2> ... |
        <value constructor 2> <type 1> <type 2> ... |
        ...
```

Both the type's name and its value constructors must start with a capital letter. Examples:

```
data Bool = False | True
data Maybe a = Just a | Nothing
data Shape = Circle Double Double Double | Rectangle Double Double Double
    ↪ Double
```

The first two examples are definitions from the Prelude. `Maybe` can be used as an alternative to errors in functions that may fail (for instance, `head` with an empty list). They allow for the programmers to detect these failure scenarios and handle them without issuing an execution-breaking error. Maybe has two value constructors: Just and Nothing. Just has a type variable argument (i.e. a *field*) while Nothing has no arguments/fields.

The example with Shape has two value constructors: one for circles and another one for rectangles. The Circle constructor receives three doubles: two for the x-y

coordinates of its center and one for the radius. The Rectangle constructor receives four doubles for the x-y coordinates of two opposite vertices (in the format: `<x1> <y1 ↪ > <x2> <y2>`). The sides of the rectangles are assumed to be aligned with the x and y axes.

Each value constructor must only be used once in a data declaration. They can be used in two different ways: as functions or in patterns. Examples:

```
Prelude> data Shape = Circle Double Double Double | Rectangle Double Double
     ↪ Double Double

Prelude> :type Circle
Circle :: Double -> Double -> Double -> Shape
Prelude> :type (Circle 2.0)
(Circle 2.0) :: Double -> Double -> Shape

Prelude> let area (Circle _ _ r) = pi*r^2
Prelude> :type area
area :: Shape -> Double
Prelude> area(Circle 1.0 2.0 1.0)
3.141592653589793
Prelude> map (area.(Circle 1.0 2.0)) [1..5]
[3.141592653589793,12.566370614359172,28.274333882308138,50.26548245743669,78.53981633974483]
     ↪
```

The type of the area function is `Shape -> Double` rather than `Circle -> Double` , since Shape is the actual type's name, while Circle is the name of one of its value constructors. All the patterns presented in the previous chapters also matched against constructors: `2` and `[]` are value constructors for integers and lists, for example.

Unlike type synonyms, type definitions can be recursive. Example:

```
Prelude> data MyList a = List a (MyList a) | EmptyList
*Main> :type (List 4 (List 6 EmptyList))
(List 4 (List 6 EmptyList)) :: Num a => MyList a
```

Sample exercises

Sample exercise UT-5

Problem statement

Consider the definition of the type `Shape` described in this section.

Using pattern matching, implement area, which given a shape s, computes the area of s. The function should be able to handle shapes constructed with Circle and Rectangle.

Usage examples:

```
*Main> area (Circle 0 1 1)
3.141592653589793
*Main> area (Rectangle 0 0 1 2)
2.0
```

```
*Main> area (Rectangle 0 0 4 2)
8.0
```

Solution

```
area :: Shape -> Double
area (Circle _ _ r) = pi*r^2
area (Rectangle x1 y1 x2 y2) = abs((x1-x2)*(y1-y2))
```

Using pattern matching, a clause is defined for each value constructor.

Practice exercises

Practice exercise UT-6

Problem statement

Consider the definition of the type `Shape` described in this section.

Using pattern matching, implement perimeter, which given a shape s, computes the perimeter of s. The function should be able to handle shapes constructed with Circle and Rectangle.

Usage examples:

```
*Main> perimeter (Circle 0 1 1)
6.283185307179586
*Main> perimeter (Rectangle 0 0 1 2)
6.0
*Main> perimeter (Rectangle 0 0 4 2)
12.0
```

Practice exercise UT-7

Problem statement

Consider the definition of `HashMap` below, which represents a hash map between a key with type k and a value with type v:

```
type HashMap k v = [(k,v)]
```

Assume that all of the elements in a hash map contain an unique key.

a) Define lookup, which, given a hashmap m and a key k, returns `Just v` where v is the corresponding value if such a key-value pair exists, or `Nothing` if the such a key is absent.

b) Define areEqual, which, given two association lists l1 and l2, determines whether the two lists contain exactly the same key-value pairs.

c) Define ceilingKey, which, given a hashmap m and a key k, returns `Just k1` where kC is the least key greater than or equal to k, or `Nothing` if such a key is absent.

Usage examples:

```
*Main> mapString = [("Ana",True),("Sara",True),("Bruno",False)]
*Main> mapString2 = [("Mark",True),("Sara",True),("Bruno",False)]
*Main> mapString3 = [("Ana",False),("Sara",True),("Bruno",False)]
*Main> mapInt = [(1,'a'),(2,'b'),(3,'c'),(4,'d'),(6,'e'),(7,'f')]

*Main> lookup "Sara" mapString
Just True
*Main> lookup 3 mapInt
Just 'c'

*Main> areEqual mapString mapString
True
*Main> areEqual mapString mapString2
False
*Main> areEqual mapString mapString3
False

*Main> ceilingKey 2 mapInt
Just 2
*Main> ceilingKey 4 mapInt
Just 4
*Main> ceilingKey 5 mapInt
Just 6
*Main> ceilingKey 6 mapInt
Just 6
*Main> ceilingKey 8 mapInt
Nothing
```

Practice exercise UT-8

Problem statement

Consider flatten, which given a list, recursively converts a list into a "flat" list. The input list has a custom type called `NestedList`, which can either be an element of any type "a" or a list of elements of type "a". The output of flatten is a "regular" list.

a) Given the usage examples, define the `NestedList` type.

b) Implement flatten.

Usage examples:

```
*Main> flatten (List [])
[]
*Main> flatten (List [Elem 1, List [List [Elem 2], List [List [List [Elem
    ↪ 3]]], Elem 4]])
[1,2,3,4]
```

Solutions to the practice exercises

Practice exercise UT-6

Solution

```
perimeter :: Shape -> Double
perimeter (Circle _ _ r) = 2*pi*r
perimeter (Rectangle x1 y1 x2 y2) = 2*abs(x1-x2) + 2*abs(y1-y2)
```

Practice exercise UT-7

Solution

a)

```
myLookup :: (Eq k) => k -> HashMap k v -> Maybe v
myLookup _ [] = Nothing
myLookup x ((k,v):kvs)
    | x == k = Just v
    | otherwise = myLookup x kvs
```

Rather than returning an error, myLookup returns `Nothing` when a key does not exist, thus allowing the program to continue to be executed. This useful property of not breaking the execution is used in the next functions of this exercise.

b)

```
areEqual :: (Eq k, Eq v) => HashMap k v -> HashMap k v -> Bool
areEqual m1 m2 =
    (length m1 == length m2) &&
    m1 `isContainedIn` m2

isContainedIn :: (Eq k, Eq v) => HashMap k v -> HashMap k v -> Bool
[] `isContainedIn` _ = True
((k,v):kvs) `isContainedIn` m2 =
    case (myLookup k m2) of (Just v2) -> v == v2 && kvs `isContainedIn` m2
                            otherwise -> False
```

The two maps are compared by first checking if they have the same length. If so, then checking if all the elements of one of the maps are present in the second map is sufficient to establish map equality.

c)

```
ceilingKey :: (Eq k, Ord k) => k -> HashMap k v -> Maybe k
ceilingKey k m = ceilingKeyAux k m Nothing

ceilingKeyAux :: (Eq k, Ord k) => k -> HashMap k v -> Maybe k -> Maybe k
ceilingKeyAux _ [] acc = acc
ceilingKeyAux k ((k1,v1):kvs) Nothing
    | k1 >= k = ceilingKeyAux k kvs (Just k1)
    | otherwise = ceilingKeyAux k kvs Nothing
ceilingKeyAux k ((k1,v1):kvs) (Just kC)
    | k1 >= k && k1 < kC = ceilingKeyAux k kvs (Just k1)
```

```
    | otherwise = ceilingKeyAux k kvs (Just kC)
```

The auxiliary function `ceilingKeyAux` uses an accumulator on its third argument to store the lowest key greater or equal to k found so far. Prior to finding such a key, the accumulator holds `Nothing`. The auxiliary method uses pattern matching to handle the cases where the accumulator has and does not hold a key.

Practice exercise UT-8

Solution

a)

```
data NestedList a = Elem a | List [NestedList a]
```

A nested list can either be a single element or a list of elements, so two value constructors.

b)

```
-- Alternative with a single function
flatten :: NestedList a -> [a]
flatten (Elem a) = [a]
flatten (List []) = []
flatten (List (x:xs)) = (flatten x) ++ (flatten (List xs))
-- Alternative with an auxiliary function
flatten' :: NestedList a -> [a]
flatten' (Elem a) = [a]
flatten' (List l) = flattenAux l

flattenAux :: [NestedList a] -> [a]
flattenAux [] = []
flattenAux (x:xs) = (flatten' x) ++ (flattenAux xs)
```

In the recursive call of the first solution, the second argument of `(++)` must be `flatten (List xs)`, and not `flatten xs`, since xs has the type `[NestedList a]` rather than `NestedList a`, which is the correct input type for flatten.

As an alternative, one can define an auxiliary function to handle objects with the type `[NestedList a]`.

5.3 Derived types

Consider the `Shape` type defined in the previous section. If ones tries to print a shape on the console or compare two shapes, an error is issued:

```
Prelude> data Shape = Circle Double Double Double | Rectangle Double Double
    ↪ Double Double

Prelude> Rectangle 1.0 2.0 3.0 4.0
<interactive>:2:1: error:
* No instance for (Show Shape) arising from a use of ‘print’
```

```
* In a stmt of an interactive GHCi command: print it

Prelude> Circle 3 4 5 == Circle 3 5 5
<interactive>:3:1: error:
* No instance for (Eq Shape) arising from a use of '=='
* In the expression: Circle 3 4 5 == Circle 3 5 5
In an equation for 'it': it = Circle 3 4 5 == Circle 3 5 5
```

To allow a shape to be printed in the console and compared using `==`, one must define that it derives the typeclass Eq, using the `deriving` keyword. If a type T derives the typeclass TC, then T is an instance of TC. Example:

```
Prelude> data Shape = Circle Double Double Double | Rectangle Double Double
    ↪ Double Double deriving (Show,Eq)

Prelude> Rectangle 1.0 2.0 3.0 4.0
Rectangle 1.0 2.0 3.0 4.0
Prelude> Circle 3 4 5 == Circle 3 5 5
False
Prelude> Circle 3 4 5 == Circle 3 4 5
True
```

As seen above, for types with parameters that derive `Eq`, two values are equal if they use the same value constructor and their arguments are equal.

Consider the following definition of the `Tempo` type, which defines a tempo indicator of a musical piece (i.e. the speed at which it is played):

```
Prelude> data Tempo = Adagio | Andante | Moderato | Allegro | Presto deriving (
    ↪ Eq,Ord,Show,Enum)
```

Since `Tempo` is part of the `Ord` typeclass, one can compare tempos, which works because the type definition above enumerates the different values in increasing order of speed:

```
Prelude> Andante < Allegro
True
Prelude> Adagio >= Moderato
False
```

Also, since `Tempo` is part of the `Enum` typeclass, one can uses the successor and predecessor function, as well as list ranges:

```
Prelude> [Adagio ..]
[Adagio,Andante,Moderato,Allegro,Presto]
Prelude> [Adagio,Moderato ..]
[Adagio,Moderato,Presto]
Prelude> [Presto,Allegro ..]
[Presto,Allegro,Moderato,Andante,Adagio]
```

For types T with parameters that derive `Ord`, a value A is less than a value B if the value constructor of A comes before the one for B in the definition of T. If they have the same constructor, then their arguments are compared from left to right and the first argument where the values differ determines which one is "smaller". Examples:

```
Prelude> data Shape = Circle Double Double Double | Rectangle Double Double
    ↪ Double Double deriving (Eq,Ord)
Prelude> Circle 1 2 3 < Rectangle 1 2 3 4
True
Prelude> Circle 1 2 3 < Circle 1 2 3
False
Prelude> Circle 1 50 3 < Circle 1 2 3
False
Prelude> Circle 1 2 3 < Circle 1 3 3
True
```

5.4 Named fields

When defining a new type using `data`, the fields of a value can be given names using the record syntax. If a class with named fields is an instance of Show, then they are printed in a different manner. Also, value of a type with named fields can be defined in an alternative way. Examples:

```
Prelude> data Date = Date { day :: Int, month :: Int, year :: Int} deriving (
    ↪ Show)
Prelude> Date 18 6 2006
Date {day = 18, month = 6, year = 2006}
Prelude> Date {day = 18, year = 2006, month = 6}
Date {day = 18, month = 6, year = 2006}
```

Named fields have the advantage of allowing one to forget their order in the definition of a value constructor. It also avoids the need to write "getter" functions, which retrieve a field of a value. Instead, the named fields themselves serve as these "getter" functions. Example:

```
Prelude> let d = Date {day = 18, year = 2006, month = 6}
Prelude> month(d)
6
```

Example of the definition of a "getter" function in a class with unnamed fields (since the function is defined in the console, the let keyword is used):

```
Prelude>  data Date = Date Int Int Int
Prelude> let month (Date _ m _) = m
```

Practice exercises

Practice exercise UT-9

Problem statement

Consider a new type called `Country` which contains a country's name, population, surface area (in km^2) and continent.

a) Define the `Country` type.

Suggestion: use named fields.

b) Implement populationDensity, which, given a country, computes its population density (i.e. the ratio between its population and area).

c) Implement countContinent. which, given the name of a continent c and a list of countries l, returns how many countries belong of l to c.

Usage examples:

```
*Main> populationDensity (Ct "Portugal" 10280000 92212 "Europe")
111.48223658526005
*Main> countContinent "Europe" [Ct "Portugal" 10280000 92212 "Europe", Ct
    ↪ "Morocco" 36470000 446550 "Africa", Ct "Spain" 46940000 505990 "
    ↪ Europe"]
2
```

Solutions to the practice exercises

Practice exercise UT-9

Solution

a)

```
data Country = Ct { name :: String
                  , population :: Int
                  , area :: Double
                  , continent :: String
                  } deriving (Show,Eq)
```

b)

```
populationDensity :: Country -> Double
populationDensity c = fromIntegral(population(c)) / area(c)
```

fromIntegral is used to convert population, an integer, to a double, so that the division can be performed with area.

c)

```
countContinent :: String -> [Country] -> Int
countContinent cont l = length (filter (== cont) (map continent l))
```

This solution takes advantage of the fact that `continent` doubles as a "getter" function.

5.5 Modules

A module is a set of related definitions, including those of functions, types and typeclasses. Using modules has the advantage of allowing for code to be reused in other projects that require the same functions, thus avoiding code duplication.

Alongside Prelude, Haskell's standard module which is loaded by default by GHC, there are many useful modules, some of which were already used in some problems of the previous chapters. Notable modules include Data.List and Data.Char, with many useful functions for working with lists and characters, respectively.

To use a module in a source code file or on the console, one can use an import statement with the format: `import <module name>` . Multiple import statements can be used in several lines to import different modules. In source code files, these commands must written before writing definitions.

One can also select to import only certain functions/types/typeclasses. This is done with the following syntax: `import <module name> (<definition 1>, <definition 2> ↪ ...)` . Example:

```
Prelude> import Data.Char
Prelude Data.Char> toUpper 'a'
'A'

Prelude Data.Char Data.List> import Data.List (nub, sort)
Prelude Data.Char Data.List> nub "aabbcca"
"abc"
Prelude Data.Char Data.List> sort "iwuefn"
"efinuw"
```

The Hoogle website is a very useful website to learn more about Haskell's standard packages. It is a search engine of Haskell's API (Application Programming Interface).

Users can also define their own modules. A module can be exported by using the following module statement at the beginning of a source code file: `module <module name ↪ > (<definition 1, definition 2,...)where` . The enumeration of definitions to export is optional. If it is omitted, then all of the definitions in the file are exported. For instance, for exercise UT-5, the following export statement could be written in the top of the file:

```
module Shape (
Shape, -- export the data type
Circle, Rectangle -- export the value constructors
area -- export the function
) where
```

Modules can be used to define abstract data types (ADTs), a model for a type that defines certain key values and operations (such as stacks, maps or sets). When export a module that defines an ADT, only the model's type and operations are exported. The value constructors should be omitted so that applications that use this model are independent of its underlying implementation. Hiding the implementation logic has the advantage of ensuring that the source code of other modules does not need to change if the ADT's implementation is refactored.

Practice exercises

Practice exercise UT-10

Problem statement

Define a module that implements `Stack` , an abstract data type that implements the LIFO (last in, first out) principle. Only the element that was most recently added to the stack is directly accessible to the user. In order to access elements that were introduced before, one must first remove all of the elements that were introduced after it.

The module must define the following operations:

- createEmptyStack: creates an empty stack.
- isEmpty: checks if a stack is empty.
- push: adds an element to the stack.
- pop: removes an element from the stack. If the stack is empty, an "Empty stack" error must be issued.
- top: retrieves the most recently added element to the stack. If the stack is empty, an "Empty stack" error must be issued.

Note: this is an abstract data type, so the module should only expose the functions that work with stacks, not the definition of the stack itself.

Usage examples:

```
*Stack> s1 = createEmptyStack
*Stack> isEmpty s1
True

*Stack> s2 = push 1 s1
*Stack> isEmpty s2
False
*Stack> top s2
1

*Stack> s3 = push 42 s2
*Stack> isEmpty s3
False
*Stack> top s3
42

*Stack> s4 = pop s3
*Stack> top s4
1

*Stack> s5 = pop s4
*Stack> top s5
*** Exception: Empty stack
CallStack (from HasCallStack):
error, called at stack.hs:22:15 in main:Stack
*Stack> isEmpty(s5)
True

*Stack> s6 = pop s5
*Stack> isEmpty s6
*** Exception: Empty stack
CallStack (from HasCallStack):
error, called at stack.hs:18:15 in main:Stack
```

Interesting note: in the second last command of the usage examples, an "Empty stack" error is not issued when calling pop on s5 (which is empty) due to Haskell's "lazyness". The error is only issued when the invalid stack, s6, is subsequently used in the last command.

Practice exercise UT-11

Problem statement

Using the stack abstraction from exercise UT-10, implement checkParenthesis, which given a string composed of two types of parentheses, "()" and "[]", and other characters, determines if, for each opening of a variant of a parenthesis, there is a corresponding closing of the parenthesis.

Usage examples:

```
*Main> checkParentheses "(a[b()[(c)]]d)()"
True
*Main> checkParentheses "(a[b()[(c)]d)()"
False
```

Solutions to the practice exercises

Practice exercise UT-10

Solution

```
module Stack (Stack,
              createEmptyStack, isEmpty,
              push, pop, top) where

data Stack a = St [a]

createEmptyStack :: Stack a
createEmptyStack = St []

isEmpty :: Stack a -> Bool
isEmpty (St []) = True
isEmpty _ = False

push :: a -> Stack a -> Stack a
push x (St xs) = St (x:xs)

pop :: Stack a -> Stack a
pop (St []) = error "Empty␣stack"
pop (St (x:xs)) = St xs

top :: Stack a -> a
top (St []) = error "Empty␣stack"
top (St (x:xs)) = x
```

The stack was implemented with the aid of a list. The elements are sorted by order of insertion, with the head of the list representing the most recently inserted element. This takes advantage of the fact that in Haskell the temporal complexity of accessing a list's head is $O(1)$.

The solution does not export the `St` value constructor to hide the implementation logic of `Stack`.

Practice exercise UT-11

Solution

```
import Stack

checkParentheses :: [Char] -> Bool
checkParentheses str = checkParenthesesAux str createEmptyStack

checkParenthesesAux :: [Char] -> Stack Char -> Bool
checkParenthesesAux [] st = isEmpty st
checkParenthesesAux (c:cs) st
    | isOpeningParenthesis c = checkParenthesesAux cs (push c st)
    | isClosingParenthesis c = not(isEmpty st) && (closingParenthesis(top
        ↪ st) == c) && (checkParenthesesAux cs (pop st))
    | otherwise = checkParenthesesAux cs st

isOpeningParenthesis :: Char -> Bool
isOpeningParenthesis c = elem c "(["

isClosingParenthesis :: Char -> Bool
isClosingParenthesis c = elem c ")]"

closingParenthesis :: Char -> Char
closingParenthesis '(' = ')'
closingParenthesis '[' = ']'
```

This exercise is an example of the usage of an abstract data type. The checkParentheses function does not need to know how `Stack` is implemented.

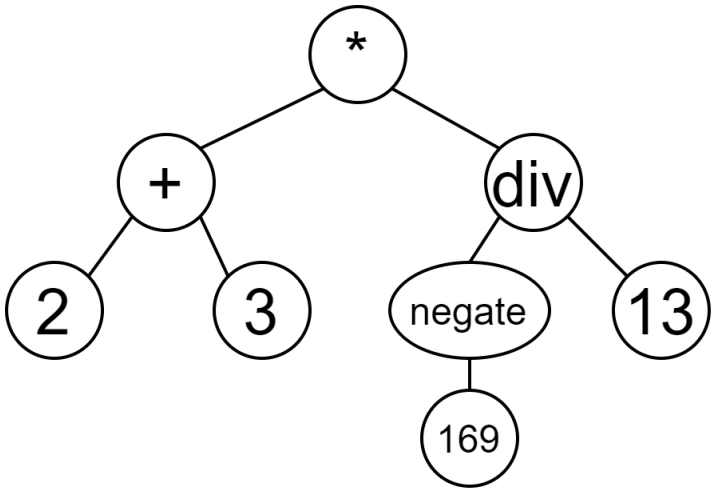
The main function, checkParentheses, calls an auxiliary function, checkParenthesesAux, which holds an additional argument with a stack. This stack keeps track of all the parentheses that have been opened, including their type. When the base case is reached, with an empty string, it is checked if the stack is empty, to ensure that all parentheses were closed. When a closing parenthesis symbol is found, it is checked if the top of the stack contains the matching symbol for opening a parenthesis. In this case, it is also checked if the stack is not empty to prevent "Empty stack" errors from being issued.

The isOpeningParenthesis, isClosingParenthesis and closingParenthesis functions were defined so that checkParenthesesAux is independent of the different types of parentheses being used. It allows the code to be easily updated in the case where a new type of parenthesis is introduced.

In this exercise, if there were only one type of parenthesis, there would be no need for a stack. One could simply use an integer for the accumulator argument of the auxiliary function to count how many parentheses were opened. When the base case is reached, for the string to be well-formed, one would only need to check if the accumulator is equal to 0.

5.6 Case study 1: Syntax trees

As presented in chapter 1, syntax trees can be used to represent expressions. Nodes (which have at least one child) represent operators while leaves (which do not have children) represent constant values. For instance, exercise IN-2 of the first chapter depicts the expression `(+) 2 3 * (negate 169 `div` 13)` with the following syntax tree:



Practice exercises

Note: As with all the other case studies in this book, the reader is recommended to define all the functions of each of these sets of exercises in the same file.

Practice exercise UT-12

Problem statement

Define the type `SyntaxTree`, which represents a syntax tree.
The definition should use three value constructors:

- One for leaves, which contain a constant value.
- Another for nodes of unary functions, which have a child and contain a unary function.
- Another for nodes of binary functions, which have two children and contain a binary function.

Example of the solution of exercise IN-2 as a value with type `SyntaxTree` `↪ : Binary (*)(Binary (+)(Const 2)(Const 3))(Binary div (Unary negate (Const 169))(` `↪ Const 13))`

Note: the tree only needs to support (unary and binary) operations where the operands have the same type as the result.

Practice exercise UT-13

Problem statement

Implement countConsts, which counts the number of constants/leaves in a `SyntaxTree` .

Usage examples:

```
*Main> countConsts(Binary (*) (Binary (+) (Const 2) (Const 3)) (Binary div
    ↪  (Unary negate (Const 169)) (Const 13)))
4
```

Practice exercise UT-14

Problem statement

Implement compute, which computes the result of the underlying expression of a `SyntaxTree` .

Usage examples:

```
*Main> compute(Binary (*) (Binary (+) (Const 2) (Const 3)) (Binary div (
    ↪ Unary negate (Const 169)) (Const 13)))
-65
*Main> compute(Binary (++) (Unary init (Const [0..2])) (Unary tail (Const
    ↪ [1..5])))
[0,1,2,3,4,5]
```

Solutions to the practice exercises

Practice exercise UT-12

Solution

```
data SyntaxTree a = Const a
                  | Unary (a -> a) (SyntaxTree a)
                  | Binary (a -> a -> a) (SyntaxTree a) (SyntaxTree a)
```

As seen in this type's definition, a constructor's fields can have a functional type.

The type was not defined as an instance of Eq or Show due to the functional field, which cannot be compared or printed in the console.

Practice exercise UT-13

Solution

```
countConsts :: (Num b) => SyntaxTree a -> b
countConsts (Const _) = 1
countConsts (Unary _ t) = countConsts t
countConsts (Binary _ t1 t2) = countConsts t1 + countConsts t2
```

The function defines a clause for each different constructor.

Practice exercise UT-14

Solution

```
compute :: SyntaxTree a -> a
compute (Const v) = v
compute (Unary f t) = f(compute t)
compute (Binary f t1 t2) = f (compute t1) (compute t2)
```

5.7 Case study 2: Binary search trees

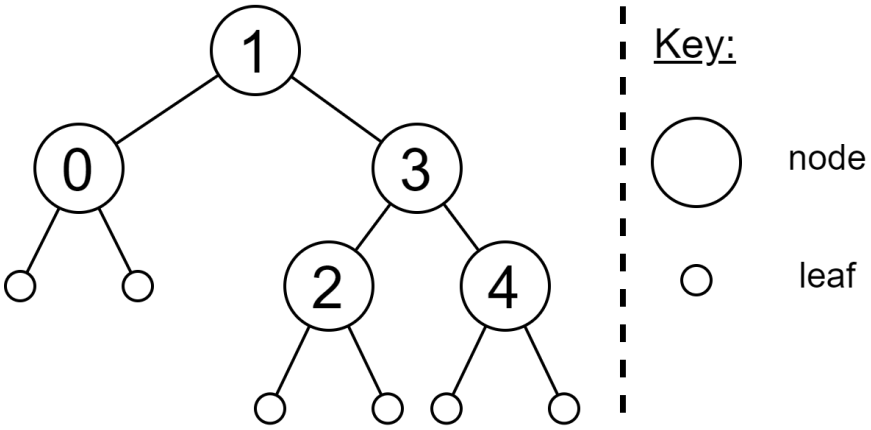
A binary search tree (BST) is a tree where all of the nodes have exactly two children and contain an element/key belonging to a type for which the `<` operator is defined (i.e. an object belonging to a type that is an instance of Ord). The two children of a node n, which are also BSTs, are known as the left and right children of n. Leaves do not contain anything.

In this case study, it will be assumed that all the keys in a BST are unique.

BSTs store their keys in an orderly fashion:

- The left subtree of a node n only contains keys that are lesser than the key of n.
- The right subtree of a node n only contains keys that are greater than the key of n.

The figure below depicts an example of a BST with integers.



Practice exercises

Practice exercise UT-15

Problem statement

Define the type `BST`, which represents a binary search tree.
Example of a value with type `BST`: `Node (Node Empty 0 Empty)1 (Node (Node Empty 2 Empty)3 (Node Empty 4 Empty))`

Practice exercise UT-16

Problem statement

Implement contains, which determines if a key is present in a BST.
Usage examples:

```
*Main> contains 1 (Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty)
    ↪ 3 (Node Empty 4 Empty)))
True
*Main> contains 5 (Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty)
    ↪ 3 (Node Empty 4 Empty)))
False
```

Practice exercise UT-17

Problem statement

Implement smallest, which returns a `Maybe` structure with the smallest key of a BST (if the tree is not empty).

Usage examples:

```
*Main> smallest (Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty) 3
    ↪ (Node Empty 4 Empty)))
Just 0
*Main> smallest Empty
Nothing
```

Practice exercise UT-18

Problem statement

Implement insert, which inserts a key into a BST and returns the resulting tree. If the key is already present, then the function returns the same tree it received as input.

Usage examples:

```
*Main> insert 2.5 (Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty)
    ↪ 3 (Node Empty 4 Empty)))
Node (Node Empty 0.0 Empty) 1.0 (Node (Node Empty 2.0 (Node Empty 2.5
    ↪ Empty)) 3.0 (Node Empty 4.0 Empty))
*Main> insert 3 (Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty) 3
    ↪ (Node Empty 4 Empty)))
Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty) 3 (Node Empty 4
    ↪ Empty))
```

Practice exercise UT-19

Problem statement

Implement delete, which deletes a key from a BST and returns the resulting tree. If the key is not present in the tree, then the function returns the same tree it received as input.

Note: when the node n with the key to be removed has children that are also nodes, one of them must take the place of the removed node. To avoid selecting a node with two non-leaf children as the replacement, one can select the smallest element of the n's right child.

Suggestion: implement an auxiliary function, deleteSmallest, which returns a pair p. The first element of p is a `Maybe` structure with the smallest element s of the input tree (just like smallest). The second element is the resulting tree after removing s.

Usage examples:

```
*Main> deleteSmallest (Node (Node Empty 0 Empty) 1 (Node (Node Empty 2
    ↪ Empty) 3 (Node Empty 4 Empty)))
(Just 0,Node Empty 1 (Node (Node Empty 2 Empty) 3 (Node Empty 4 Empty)))
```

```
*Main> delete 6 (Node Empty 5 Empty)
Node Empty 5 Empty
*Main> delete 6 (Node (Node Empty 4 Empty) 5 Empty)
Node (Node Empty 4 Empty) 5 Empty
*Main> delete 0 (Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty) 3
    ↪ (Node Empty 4 Empty)))
Node Empty 1 (Node (Node Empty 2 Empty) 3 (Node Empty 4 Empty))
*Main> delete 1 (Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty) 3
    ↪ (Node Empty 4 Empty)))
Node (Node Empty 0 Empty) 2 (Node Empty 3 (Node Empty 4 Empty))
*Main> delete 3 (Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty) 3
    ↪ (Node Empty 4 Empty)))
Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty) 4 Empty)
```

Practice exercise UT-20

Problem statement

Implement bst2List, which returns an ordered list with the keys of a BST.
Usage examples:

```
*Main> bst2List (Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty) 3
    ↪ (Node Empty 4 Empty)))
[0,1,2,3,4]
```

Practice exercise UT-21

Problem statement

Implement isOrdered, which returns a boolean indicating if a binary tree is an actual BST (i.e. if the keys are ordered as described in this section).
Usage examples:

```
*Main> isOrdered (Node (Node Empty 0 Empty) 1 (Node (Node Empty 2 Empty) 3
    ↪  (Node Empty 4 Empty)))
True
*Main> isOrdered (Node (Node Empty 0 Empty) (-1) Empty)
False
```

Practice exercise UT-22

Problem statement

Implement list2Bst, which returns a BST with the keys of a list. The output BST must be as balanced as possible (i.e. any node must have approximately the same number of keys in their left and right subtrees).
Suggestion: build the tree by recursively splitting the input list in half.
Note: remember to order the input list first!

Usage examples:

```
*Main> list2Bst [1,4,5,2,3]
Node (Node (Node Empty 1 Empty) 2 Empty) 3 (Node (Node Empty 4 Empty) 5
    ↪ Empty)
*Main> list2Bst [1,4,5,2,3,0]
Node (Node (Node Empty 0 Empty) 1 (Node Empty 2 Empty)) 3 (Node (Node
    ↪ Empty 4 Empty) 5 Empty)
```

Solutions to the practice exercises

Practice exercise UT-15

Solution

```
data BST a = Empty
           | Node (BST a) a (BST a)
           deriving (Show,Eq)
```

One could also swap the first two fields of the `Node` constructor: `Node a (BST a)(BST a)`. It is just a matter of taste!

In the type definition, the type variable a must belong to typeclass Ord. However, class constraints cannot be added to data declarations. Instead, the class constraint should be added to each function that uses `BST` (and requires the Ord property).

Practice exercise UT-16

Solution

```
contains :: (Eq a, Ord a) => a -> BST a -> Bool
contains a Empty = False
contains a (Node t1 v t2)
    | (a == v) = True
    | (a < v) = contains a t1
    | otherwise = contains a t2
```

Practice exercise UT-17

Solution

```
smallest :: (Ord a) => BST a -> Maybe a
smallest Empty = Nothing
smallest (Node Empty v t2) = Just v
```

```
smallest (Node t1 v t2) = smallest t1
```

This function defines two clauses for inputs with the `Node` value: those with and without an empty left subtree.

Practice exercise UT-18

Solution

```
insert :: (Eq a, Ord a) => a -> BST a -> BST a
insert a Empty = Node Empty a Empty
insert a t@(Node t1 v t2)
    | (a == v) = t
    | (a < v) = Node (insert a t1) v t2
    | otherwise = Node t1 v (insert a t2)
```

This solution uses an as-pattern, which enables one to provide a name to a variable while also matching its structure. The as-pattern is completely optional: it is only used to make the solution more elegant.

Practice exercise UT-19

Solution

```
deleteSmallest :: BST a -> (Maybe a, BST a)
deleteSmallest Empty = (Nothing, Empty)
deleteSmallest (Node Empty v t2) = (Just v, Empty)
deleteSmallest (Node t1 v t2) =
    let (s,t1') = deleteSmallest t1 in (s,Node t1' v t2)

delete :: (Eq a, Ord a) => a -> BST a -> BST a
delete a Empty = Empty
delete a (Node t1 v Empty)
    | (a == v) = t1
delete a (Node Empty v t2)
    | (a == v) = t2
delete a (Node t1 v t2)
    | (a == v) = let (s,t2') = deleteSmallest t2 in
        case s of Just m -> Node t1 m t2'
                  Nothing -> t1
    | (a < v) = Node (delete a t1) v t2
    | otherwise = Node t1 v (delete a t2)
```

The auxiliary method deleteSmallest allows for an efficient implementation of delete. A less efficient alternative would be to use smallest to determine the smallest key of the right subtree and then call delete to remove it, which would result in two traversals of the right subtree instead of only one.

As an alternative to using the smallest element of the right subtree as the substitute node, the largest element of the left subtree could also be used. Any other key from any of the subtrees can be used, but to avoid having to change

the tree's structure too much, it is better to choose a node that is guaranteed to have no non-empty children.

One must not forget to define clauses for the cases where exactly one of the node's children is empty. In particular, the case where the right child is empty must be defined so that the result is the left child if the input is equal to the node's key. The other clause is optional and merely serves to avoid having to look for smallest element of the right child if the left child is empty. Also, there is no need to write "otherwise sub-clauses" for the clauses that handle one empty child since they are handled by the last, more general clause.

Practice exercise UT-20

Solution

```
bst2List :: BST a -> [a]
bst2List Empty = []
bst2List (Node t1 v t2) = (bst2List t1) ++ [v] ++ (bst2List t2)
```

Practice exercise UT-21

Solution

```
-- Alternative with recursion
isOrdered :: (Ord a) => BST a -> Bool
isOrdered t = isOrderedAux (bst2List t)

isOrderedAux :: (Ord a) => [a] -> Bool
isOrderedAux [] = True
isOrderedAux [_] = True
isOrderedAux (x:y:xs) = (x < y) && isOrderedAux (y:xs)

-- Alternative with a list comprehension
isOrdered' :: (Ord a) => BST a -> Bool
isOrdered' t =
    let l = bst2List t in and [x < y | (x,y) <- zip l (tail l)]
```

The solution can be defined with the aid of bst2List, so that it is just a matter of checking if a list is ordered.

Practice exercise UT-22

Solution

```
list2Bst :: (Ord a) => [a] -> BST a
list2Bst l = list2BstAux(qSort(l))

qSort :: (Ord a) => [a] -> [a]
qSort [] = []
```

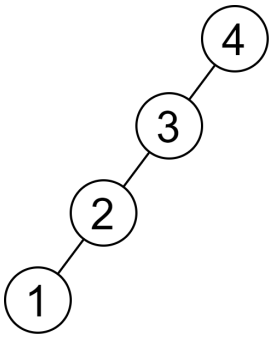
```
qSort (x:xs) = (qSort lower) ++ [x] ++ (qSort upper)
    where lower = [y | y <- xs, y < x]
          upper = [y | y <- xs, y >= x]

list2BstAux :: (Ord a) => [a] -> BST a
list2BstAux [] = Empty
list2BstAux l = Node (list2BstAux l1) x (list2BstAux l2)
    where halfLen = div (length l) 2
          l1 = take halfLen l
          x:l2 = drop halfLen l
```

In the auxiliary function list2BstAux, the element to store as the key of the root node is taken from l2 rather than from l1, since, assuming that l is non-empty, `drop halfLen l` is guaranteed to have at least one element of l as halfLen corresponds to a truncated division of l's length by 2.

The input list was ordered using quicksort. This implementation is more compact than the proposed solution for exercise LI-24 since it uses comprehension lists (which were not introduced back then).

5.8 Case study 3: AVL trees

In an arbitrary binary search tree (BST) t with n elements, a function to look for a value in t has a worst-case temporal complexity of $O(n)$. The worst-case scenario occurs if the height of the tree (i.e. the length of the longest path from the root to one of the leaves) is equal to n, as shown below:

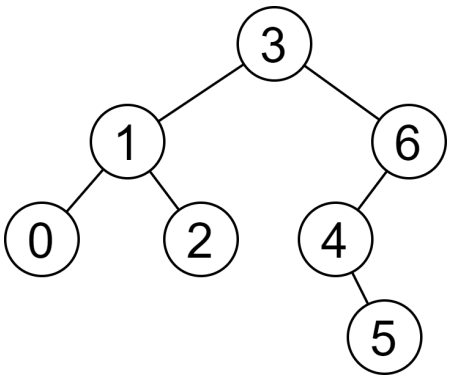


To prevent this problem, one can *balance* the tree, where one of its subtrees (or the whole trees) is reconstructed, in order to reduce its height whilst maintaining all of its keys and satisfying the BST-property (i.e. that the keys are ordered).

An AVL tree is a particular case of a BST that ensures it remains "balanced" after inserting and removing keys (i.e. it is a *self-balancing tree*). The tree is named after the surname of its two creators: Georgy Adelson-Velsky and Evgenii Landis, who

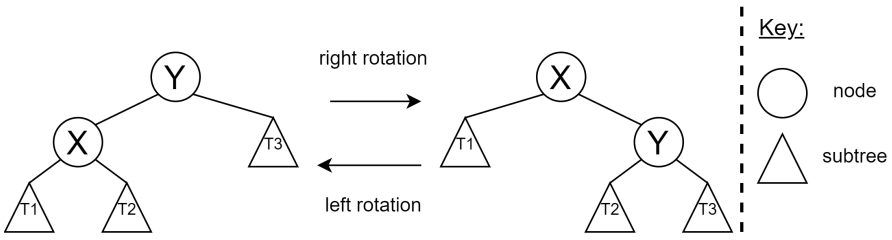
first introduced in 1962 with a scientific journal article entitled "An algorithm for the organization of information".

All AVL trees ensure they are balanced by respecting a property known as the *height invariant*: the heights of the left and right subtrees of any node differ by at most **one** unit. The example of a BST provided in the previous section respects the height invariant. The figure below presents an example of a tree that violates this invariant: the node with key 6 has a left child with a height of 2 and right child with a height of 0 (right subtree is a leaf).



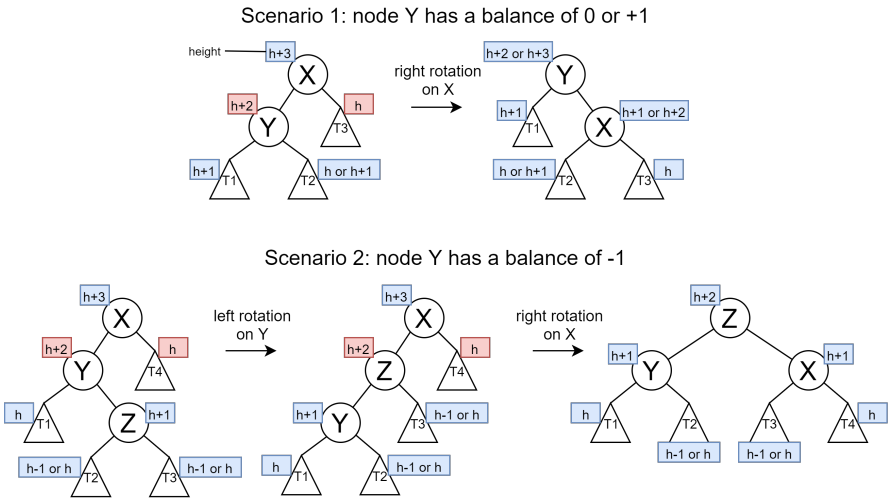
To ensure the height invariant is maintained after inserting or removing a key, first the operation is executed as in a regular BST, then it is checked if each node along the path from the inserted/removed node to the root is balanced and apply a rebalancing operation when needed. Only the nodes along this path may become unbalanced. The rebalancing operation is identical to what may be needed after inserting ou removing a node.

An AVL tree is rebalanced using left and right rotations in a given node, which changes the "shape" of the trees rooted on that node without breaking the BST-property. The two operations are presented below:



When a node is unbalanced, the difference between the heights of the left and right subtree, which will be henceforth called *balance*, is either +2 ou -2. There are

four possible "unbalance scenarios": two for a balance of +2, and two for a balance of -2. The scenarios with a balance of +2 and the rotations needed to solve them are depicted below:



The scenarios with a balance of -2 are symmetrical.

As a result of the height invariant and the rebalancing operations that ensure it, an AVL tree with n keys has a maximum height of approximately $log(n)$. Thus, lookup, insertion and deletion operation have a worse-case temporal complexity of $O(log(n))$.

The proposed definition of the `AVLTree` type is very similar to `BST`. Also, the operations of determining if a key is present and retrieving the smallest key are identical:

```
data AVLTree a = Empty
               | Node (AVLTree a) a (AVLTree a)
               deriving (Show,Eq)

contains :: (Eq a, Ord a) => a -> AVLTree a -> Bool
contains a Empty = False
contains a (Node t1 v t2)
    | (a == v) = True
    | (a < v) = contains a t1
    | otherwise = contains a t2

smallest :: AVLTree a -> Maybe a
smallest Empty = Nothing
smallest (Node Empty v t2) = Just v
smallest (Node t1 v t2) = smallest t1
```

Practice exercises

Practice exercise UT-23

Problem statement

a) Implement height, which returns the height of a tree (i.e. the length of the longest path from the root to one of the leaves).

b) Implement balance, which given an AVL tree t, returns the difference between the height of the left and right subtrees of t's root.

Usage examples:

```
*Main> height(Node Empty 1 Empty)
1
*Main> height(Node (Node (Node Empty 0 Empty) 1 (Node Empty 2 Empty)) 3
     ↪ Empty)
3
*Main> height(Node (Node (Node Empty 0 Empty) 1 (Node Empty 2 Empty)) 3 (
     ↪ Node (Node Empty 4 (Node Empty 5 Empty)) 6 Empty))
4

*Main> balance(Node Empty 1 Empty)
0
*Main> balance(Node (Node (Node Empty 0 Empty) 1 (Node Empty 2 Empty)) 3 (
     ↪ Node (Node Empty 4 (Node Empty 5 Empty)) 6 Empty))
-1
*Main> balance(Node (Node (Node Empty 0 Empty) 1 (Node Empty 2 Empty)) 3
     ↪ Empty)
2
```

Practice exercise UT-24

Problem statement

Implement isBalanced, which returns a boolean indicating if a tree respects the height invariant.

Usage examples:

```
*Main> isBalanced(Node (Node Empty 1 Empty) 2 (Node Empty 3 Empty))
True
*Main> isBalanced(Node (Node (Node Empty 0 Empty) 1 (Node Empty 2 Empty))
     ↪ 3 (Node (Node Empty 4 (Node Empty 5 Empty)) 6 Empty))
False
```

Practice exercise UT-25

Problem statement

a) Implement rotateLeft, which performs a left rotation on the root node of a tree. If such a rotation is not possible, then the function returns its input.

b) Implement rotateRight, which performs a right rotation on the root node of a tree. If such a rotation is not possible, then the function returns its input.

Usage examples:

```
*Main> rotateLeft(Node Empty 1 (Node Empty 2 Empty))
Node (Node Empty 1 Empty) 2 Empty
*Main> rotateLeft(Node (Node (Node Empty 0 Empty) 1 (Node Empty 2 Empty))
    ↪ 3 (Node (Node Empty 4 (Node Empty 5 Empty)) 6 Empty))
Node (Node (Node (Node Empty 0 Empty) 1 (Node Empty 2 Empty)) 3 (Node
    ↪ Empty 4 (Node Empty 5 Empty))) 6 Empty

*Main> rotateRight(Node (Node Empty 1 Empty) 2 Empty)
Node Empty 2 (Node Empty 1 Empty)
```

Practice exercise UT-26

Problem statement

Implement rebalance, which rebalances a tree using left and right rotations in scenarios where the unbalance is present in the input tree's root (i.e. its balance is not in the [-1,1] range). If the root is balanced, then the function returns the input tree.

Note: this function assumes that the subtrees of the root are balanced. Usage examples:

```
*Main> rebalance(Node Empty 1 (Node Empty 3 Empty))
Node Empty 1 (Node Empty 3 Empty)
*Main> rebalance(Node Empty 1 (Node (Node Empty 2 Empty) 3 Empty))
Node (Node Empty 1 Empty) 3 (Node Empty 2 Empty)
*Main> rebalance(Node Empty 1 (Node Empty 2 (Node Empty 3 Empty)))
Node (Node Empty 1 Empty) 2 (Node Empty 3 Empty)
```

Practice exercise UT-27

Problem statement

Implement insert, which inserts a key into a balanced AVL tree and returns the resulting balanced tree. If the key is already present, then the function returns the same tree it received as input.

Suggestion: copy and adapt the solution from exercise UT-18.

Usage examples:

```
*Main> let t1 = insert 1 Empty
*Main> t1
Node Empty 1 Empty

*Main> let t2 = insert 5 t1
*Main> t2
Node Empty 1 (Node Empty 5 Empty)

*Main> let t3 = insert 3 t2
*Main> t3
```

```
Node (Node Empty 1 Empty) 5 (Node Empty 3 Empty)

*Main> let t4 = insert 6 t3
*Main> t4
Node (Node Empty 1 Empty) 5 (Node Empty 3 (Node Empty 6 Empty))

*Main> let t5 = insert 7 t4
*Main> t5
Node (Node Empty 1 Empty) 5 (Node (Node Empty 3 Empty) 6 (Node Empty 7
     ↪ Empty))

*Main> let t6 = insert 8 t5
*Main> t6
Node (Node (Node Empty 1 Empty) 5 (Node Empty 3 Empty)) 6 (Node Empty 7 (
     ↪ Node Empty 8 Empty))
```

Practice exercise UT-28

Problem statement

Implement delete, which deletes a key from a balanced AVL tree and returns the resulting balanced tree. If the key is not present in the tree, then the function returns the same tree it received as input.

Suggestion: similarly to exercise UT-19, implement an auxiliary function, deleteSmallest. Copy and adapt the solution from this exercise.

Usage examples:

```
*Main> t1 = delete 1 (Node (Node (Node Empty 1 Empty) 5 (Node Empty 3
     ↪ Empty)) 6 (Node Empty 7 (Node Empty 8 Empty)))
*Main> t1
Node (Node Empty 5 (Node Empty 3 Empty)) 6 (Node Empty 7 (Node Empty 8
     ↪ Empty))

*Main> t2 = delete 5 t1
*Main> t2
Node (Node Empty 3 Empty) 6 (Node Empty 7 (Node Empty 8 Empty))

*Main> t3 = delete 3 t2
*Main> t3
Node (Node Empty 6 Empty) 7 (Node Empty 8 Empty)

*Main> t4 = delete 7 t3
*Main> t4
Node (Node Empty 6 Empty) 8 Empty

*Main> t5 = delete 8 t4
*Main> t5
Node Empty 6 Empty

*Main> t6 = delete 7 t5
*Main> t6
Node Empty 6 Empty

*Main> t7 = delete 6 t5
*Main> t7
Empty
```

Solutions to the practice exercises

Practice exercise UT-23

Solution

a)

```
height :: (Num b, Ord b) => AVLTree a -> b
height Empty = 0
height (Node t1 _ t2) = 1 + max (height t1) (height t2)
```

b)

```
balance :: (Num b, Ord b) => AVLTree a -> b
balance Empty = 0
balance (Node t1 _ t2) = (height t1) - (height t2)
```

Practice exercise UT-24

Solution

```
isBalanced :: AVLTree a -> Bool
isBalanced Empty = True
isBalanced t@(Node t1 _ t2) = (bal >= -1) && (bal <= 1) && (isBalanced t1)
    ↪  && (isBalanced t2)
    where bal = balance t
```

The solution uses an as-pattern to simplify the where-expression.

Practice exercise UT-25

Solution

a)

```
rotateLeft :: AVLTree a -> AVLTree a
rotateLeft (Node t1 x (Node t2 y t3)) = Node (Node t1 x t2) y t3
rotateLeft t = t
```

The first clause captures the cases where the rotation is possible, while the second one is used for the edge-cases, where rotateLeft works like the identity function.

b)

```
rotateRight :: AVLTree a -> AVLTree a
rotateRight (Node (Node t1 y t2) x t3) = Node t1 x (Node t2 y t3)
rotateRight t = t
```

Practice exercise UT-26

Solution

```
rebalance :: AVLTree a -> AVLTree a
rebalance Empty = Empty
rebalance t@(Node t1 v t2)
    | bal > 1 =
        if(balance t1 >= 0)
            then rotateRight(t)
            else rotateRight(Node (rotateLeft t1) v t2)
    | bal < -1 =
        if(balance t2 <= 0)
            then rotateLeft(t)
            else rotateLeft(Node t1 v (rotateRight t2))
    | otherwise = t
    where bal = balance t
```

The solution distinguishes all four possible unbalance scenarios.

Practice exercise UT-27

Solution

```
insert :: (Eq a, Ord a) => a -> AVLTree a -> AVLTree a
insert a Empty = Node Empty a Empty
insert a t@(Node t1 v t2)
    | (a == v) = t
    | (a < v) = rebalance(Node (insert a t1) v t2)
    | otherwise = rebalance(Node t1 v (insert a t2))
```

The solution is very similar to the one of exercise UT-18 for BSTs. The sole difference is that the output of the recursive calls is applied to the rebalance function.

In the base case, there is no need to call rebalance on `Node Empty a Empty` since this tree with a single element will always be balanced. Also, in the recursive step, there is no need to call rebalance on the case where `a == v` since the input tree (which is assumed to be balanced) is left unchanged.

Practice exercise UT-28

Solution

```
deleteSmallest :: AVLTree a -> (Maybe a, AVLTree a)
deleteSmallest Empty = (Nothing, Empty)
deleteSmallest (Node Empty v t2) = (Just v, Empty)
deleteSmallest (Node t1 v t2) =
    let (s,t1') = deleteSmallest t1 in (s,rebalance(Node t1' v t2))

delete :: (Eq a, Ord a) => a -> AVLTree a -> AVLTree a
delete a Empty = Empty
delete a (Node t1 v Empty)
```

```
    | (a == v) = t1
delete a (Node Empty v t2)
    | (a == v) = t2
delete a (Node t1 v t2)
    | (a == v) = let (s,t2') = deleteSmallest t2 in
        case s of Just m -> rebalance(Node t1 m t2')
                  Nothing -> t1
    | (a < v) = rebalance(Node (delete a t1) v t2)
    | otherwise = rebalance(Node t1 v (delete a t2))
```

The solution is very similar to the one of exercise UT-19 for BSTs. Similarly to the case of exercise UT-27, the sole difference is that the output of the recursive calls is applied to the rebalance function.

In the second and third clauses, which consider nodes with exactly one empty child, there is no need to call rebalance since the non-empty subtree must necessarily have a height of 1 (i.e. be a tree with a single node) due to the height invariant.

Chapter 6
Interactive programs

Haskell is essentially a pure programming language. As a result, functions are free of side-effects such as writing on the console or interacting with a machine's file system. However, Haskell is by no means a limited language and allows for these sort of operations to occur. To bridge the gap between pure and impure code, the language uses the IO monad. This chapter will present this monad by first explaining how to interact with the console and with files, and then how to generate pseudo-random values. Afterwards, monads are briefly presented. The chapter concludes with a case study to implement Conway's Game of Life.

6.1 Standard I/O

The standard input (stdin) is used by a program to read its input data, while the standard output (stdout) is used to write its output data. By default, on console-based applications (such as GHCi), the standard input is assigned to the keyboard, while the standard output is assigned to the program's text terminal.

Table 6.1 presents some relevant Prelude functions for input/output on the console, i.e. the standard input/output.

The end-of-file (EOF) character can be written on the console by typing CTRL+D (on Unix-based systems) or CTRL+Z (on Windows).

Using the functions of Table 6.1 one can write simple functions that write text on the console. Consider the following code, which is written on a source code file named printHi.hs:

```
printHi :: IO ()
printHi = putStrLn "Hi!"

main = putStrLn "Hello, world!"
```

By using the typical load command of GHCi (for example, `:load printHi.hs`), one can then call main or printHi to print both strings in the console as both functions are loaded to the GHCi environment. As an alternative, one can compile the source

Table 6.1 Some standard I/O functions

Function	Comment
putChar	Writes a character on the stdout.
putStr	Writes a string on the stdout.
putStrLn	Writes a string followed by a newline (`'n'`) on the stdout.
print	Prints a value (such as an `Int`) on the stdout.
getChar	Reads a character from the stdin. Only returns after a newline is read.
getLine	Reads text from the console until a newline is read from the stdin.
getContents	Reads all text from the console until an end-of-file (EOF) character is read from the stdin.
return	Returns an empty action with type `IO ()` .

file and then run it using the following commands on the command line (not in the GHCi environment):

```
$ ghc printHi.hs
[1 of 1] Compiling Main             ( printHi.hs, printHi.o )
Linking printHi.exe ...
$ ./printHi
Hello, world!
```

The difference between loading the functions to GHCi and compiling the source file and then running it is that, with the latter alternative, the main function is executed, which, in, this case, prints "Hello, world!" in the console.

Unlike regular functions, one does not usually specify the type of main, which has the type declaration: `main :: IO ()` . Thus, main is a function that takes no arguments and returns a value of the type `IO ()` , which represents an IO (input/output) action with nothing inside it. Actions affect the world outside the program and cannot receive arguments. An IO action with type `IO t` is an entity that, when executed performs input/output and returns a value with type t. Examples:

- putChar has the type `String -> IO ()` . It receives a `Char` and returns an action responsible for printing it on the console, so this action has no need to hold a value.
- getChar, on the other hand, has the type `IO Char` . It takes no arguments and returns an action responsible for asking the user for a character which it can then return.

Therefore, Haskell distinguishes pure and impure functions using their type.

Multiple actions can be executed sequentially as a single IO action by using a do-block. Example:

```
hello :: IO ()
hello = do
    name <- getLine
    putStrLn $ "Hello␣" ++ name ++ "!"
```

The hello function performs two IO actions: reading a line from the keyboard and storing it in the "name" variable, then printing the value of "name" in the console. Hello has the type `IO ()` since the do-block has the same type as the last action. The line `<variable> <- <action>` runs the IO action `<action>` and binds the action's result to <variable>. The `<-` operator and action sequencing can only be used inside

"IO-code blocks" (such as do-blocks), which allows for the separation of pure and impure code. It should be noted that the last action in a do-block cannot be bound to a variable.

A less elegant alternative for the do-notation is to use semi-colons instead of indentation:

```
hello' :: IO ()
hello' = do name <- getLine; putStrLn $ "Hello␣" ++ name ++ "!"
```

The return function works differently from imperative languages, where it usually ends the execution of a function and receives as an argument the function's return value. In Haskell, return receives an argument x with type t creates an IO action that returns x, i.e. a value with the type `IO t`. The resulting IO action is not actually executed. Also, inside a do-block return does not end the execution of the block prematurely, nor does affect the type of the result unless it is the last statement. For instance, consider the functions below:

```
funcX :: IO String
funcX = do
    return ()
    return "hello"
    return "123"

funcY :: IO String
funcY = do
    funcX
    funcX

funcZ :: IO ()
funcZ = do
    funcX
    return ()
```

When funcX is executed, it only prints "123" since the function returns an IO action with a string containing "123". The string is only printed when the IO action is evaluated, not when it is created. As a result, calling funcY also only prints "123" once in the console. Finally, funcZ does not print anything since it returns an IO action with nothing inside it. The `return ()` expression is useful when defining functions that returns objects with the type `IO ()`.

The return function can be considered as the counterpart of the `<-` operator: return receives a value x and creates IO action that holds x, while `x <- <action>` extracts the x value of an action. Thus, `x <- return y` can be viewed as an unnecessarily complicated version of `x = y`.

Let bindings can also be used inside do-blocks. Unlike in pure Haskell code, the "in" is not needed. Example:

```
testLet :: IO ()
testLet = do
    x <- getLine
    let str = "A␣":x
    printStrLn str
```

"Common value types", such as integers, can be converted to string's using the `show` function. Conversely, strings can be parsed into values using the `read` function. Values that support `show` and `read` belong to `Show` and `Read` typeclasses, respectively.

These functions are quite useful to interpret user input and to print messages with numbers. Examples:

```
Prelude> show 3
"3"
Prelude> show [1,2,3]
"[1,2,3]"
Prelude> read "3" :: Int
3
Prelude> read "[True,False]" :: [Bool]
[True,False]
Prelude> read "[1,2,3.5]" :: [Double]
[1.0,2.0,3.5]
```

In the examples above involving `read`, a type annotation had to be supplied as GHCi had no way of inferring the result's type. For instance, `3` can be an `Int`, an `Integer` or possible a `Double`.

In all the exercises of this chapter, consider importing the following source code file as a module:

```
module IOUtils (
    cls,
    goto, writeAt, writeListAt,
    Color (..),
    clrFg, clrBg, clrReset,
    wait) where

import System.Posix (usleep)  -- for Unix-based systems, modify/uncomment
    ↪ accordingly if another operating system is used, if needed
-- import System.Win32.Process (sleep) -- for Windows

-- Clears the console.
cls :: IO ()
cls =  putStr "\ESC[2J"

-- Writes a character at the position (x,y) of the console.
-- (1,1) is the top-left corner.
writeAt :: (Integral a, Show a) => (a,a) -> Char -> IO ()
writeAt p c = do
    goto p
    putChar c

goto :: (Integral a, Show a) => (a,a) -> IO ()
goto (x,y) = putStr ("\ESC[" ++ show y ++ ";" ++ show x ++ "H")

-- Writes the same character at multiple positions on the console.
writeListAt :: (Integral a, Show a) => [(a,a)] -> Char -> IO ()
writeListAt [] _ = return ()
writeListAt (p:ps) c = do
    writeAt p c
    writeListAt ps c

data Color = Black | Red | Green | Yellow | Blue | Magenta | Cyan | White
    deriving (Eq,Show,Enum)
-- Changes the color of the foreground text
clrFg :: Color -> IO()
clrFg Black   = putStr "\ESC[30m"
clrFg Red     = putStr "\ESC[31m"
clrFg Green   = putStr "\ESC[32m"
clrFg Yellow  = putStr "\ESC[33m"
clrFg Blue    = putStr "\ESC[34m"
clrFg Magenta = putStr "\ESC[35m"
clrFg Cyan    = putStr "\ESC[36m"
clrFg White   = putStr "\ESC[37m"
```

```
-- Changes the color of the background text
clrBg :: Color -> IO()
clrBg Black   = putStr "\ESC[40m"
clrBg Red     = putStr "\ESC[41m"
clrBg Green   = putStr "\ESC[42m"
clrBg Yellow  = putStr "\ESC[43m"
clrBg Blue    = putStr "\ESC[44m"
clrBg Magenta = putStr "\ESC[45m"
clrBg Cyan    = putStr "\ESC[46m"
clrBg White   = putStr "\ESC[47m"

-- Resets the foreground and background color
clrReset :: IO ()
clrReset = putStr "\ESC[0m"

-- Waits for some milliseconds.
-- for Unix-based systems
wait :: Int -> IO ()
wait t =  usleep (t*1000)

-- for Windows
-- wait :: Int -> IO ()
-- wait t =  sleep t
```

This module contains useful functions for command-line interfaces, including functions to:

- clear the screen;
- draw at a specific position of the screen;
- change the color of the text and its background;
- wait for a certain amount of time.

The first three types of functions use ANSI escape sequences. For the last function type, depending on the Haskell compiler/version and operating system being used, one might need to change the module being imported and the wait function's implementation.

Sample exercises

Sample exercise IP-1

Problem statement

Implement ioSucc, which asks the user for an integer n, reads it and then prints its successor (i.e. n+1).

Usage examples:

```
*Main> ioSucc
Write an integer:
4
The successor is 5.
```

Solution

```
ioSucc :: IO ()
ioSucc = do
    putStrLn "Write␣an␣integer:"
    input <- getLine
    let num = read input :: Int
    putStrLn $ "The␣successor␣is␣" ++ show (succ num) ++ "."
```

ioSucc has the type `IO ()` since it performs input/output operations and does not need to hold any value for future usage. It has the same type as the last statement of the do-block, `putStrLn $ "The␣successor␣is␣"++ show (succ num)++ "."` .

Using the `$` operator after `putStrLn` (or another printing function) is very useful when printing complex expressions to avoid an extra pair of parentheses.

Sample exercise IP-2

Problem statement

Using getLine, implement getNLines, which, given an integer n, reads n lines from stdin.

Note: getLine does not store the newline character (`'n'`) in the string contained in the resulting IO action, so one can append a newline to the end of each line that is read when writing them in the console.

Usage examples:

```
*Main> getNLines 4
Hello.
Hi, there!
How are you?
I'm fine.
"Hello.\nHi,␣there!\nHow␣are␣you?\nI'm␣fine.\n"

*Main> do str <- getNLines 4; putStr str
Hello.
Hi, there!
How are you?
I'm fine.
Hello.
Hi, there!
How are you?
I'm fine.
```

Note: The output of the first test case literally prints the newlines since the show function is being called with an IO action as input. Writing `return "n"` in GHCi also produces `"n"` .

Solution

```
getNLines :: (Num a, Ord a) => a -> IO String
getNLines 0 = return []
getNLines n
    | n > 0 = do
```

```
        l <- getLine
        ls <- getNLines (n-1)
        return (l ++ '\n':ls)
    | otherwise = error "Negative index"
```

This function is an example of a recursive function that produces an IO action.

For the base case, the result is `return []` (or `return ""`) rather than `return ()` since the function's output type is `IO String`, and not `IO ()`.

For the recursive step, the result is given by the last line of the do-block, which creates an IO action containing a string where the first line is concatenated to a newline, followed by the string contained in the result of the recursive call. Omitting the `ls <- getNLines (n-1)` statement and writing the result as `return (l ++ '\n':(getNLines (n-1)))` would be incorrect as it implies concatenating a string with an IO action.

Sample exercise IP-3

Problem statement

a) Implement drawBorders, which, given a board size n, prints a series of dots ('.') for the border of a n by n square.

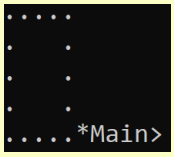
Suggestion: use the writeListAt function from the IOUtils module presented above.

b) Implement patrolPoint, which, given a number of ticks nTicks and a board size n, animates an 'x' character moving clockwise along the perimeter of a n by n square for nTicks iterations/ticks. The borders from drawBorders must also be drawn, so that the 'x' moves "along the dots".

Note: the system should wait some milliseconds (for example, 100 ms) before showing the next frame of the animation. Also, remember to clear the screen before showing each frame.

Usage examples:

After writing `do cls; drawBorders 5` on GHCi, the console should be similar to the following figure:



After writing `patrolPoint 50 5` on GHCi, the first frame should be similar to the following figure:

The seventh frame should be similar to the following figure:

Solution

a)

```
import IOUtils (cls, writeAt, writeListAt, wait)

drawBorders :: Int -> IO()
drawBorders n = writeListAt [(x,y) | x <- [1..n], y <- [1..n], (min x y ==
    ↪  1) || (max x y == n)] '.'
```

A list comprehension is used to create a list with all of the positions within a n by n square. A guard is used to test if a position is located at the border of the square.

b)

```
patrolPoint :: Int -> Int -> IO ()
patrolPoint nTicks n
    | nTicks >= 0 && n >= 0 = patrolPointAux nTicks n (1,1)
    | otherwise = error "Negative argument"

type Pos = (Int,Int)
patrolPointAux :: Int -> Int -> Pos -> IO ()
patrolPointAux 0 _ _ = return ()
patrolPointAux nTicks n dot = do
    cls
    drawBorders n
    writeAt dot 'x'
    wait 200
    patrolPointAux (nTicks - 1) n (nextPoint dot n)

nextPoint :: Pos -> Int -> Pos
nextPoint (x,y) n
    | y == 1 && x < n = (x+1,y)
    | x == n && y < n = (x,y+1)
    | y == n && x > 1 = (x-1,y)
    | otherwise = (x,y-1)
```

For convenience purposes, the solution defines a type synonym for the x's position.

The solution shows an example of how to work with states on Haskell, which are used to store information between iterations of an interactive program. Iteration is achieved via recursion, while the state is kept in arguments of the recursive function.

In this case, the x's position at any given frame corresponds to the state of the program. The next state is computed using the function nextPoint, where one must be mindful that the x changes direction when it reaches one of the four corners of the border.

Each iteration of the recursive step patrolPointAux prints a frame of the animation.

Practice exercises

Practice exercise IP-4

Problem statement

Explain why computing the following expression leads to an error: `sentence = "Hello, my name is " ++ getLine .`

Practice exercise IP-5

Problem statement

Implement reverseUntil, which reads lines from the keyboard and prints them in reverse until an empty line is typed.

Suggestion: use the null function to test if a string is empty.

Usage examples:

```
*Main> reverseUntil
Hello
olleH
Hi
iH

*Main>
```

Practice exercise IP-6

Problem statement

a) Implement myPutStr, which works similarly to Prelude's putStr.

Suggestion: Use putChar.
b) Implement myPutStrLn, which works similarly to Prelude's putStrLn.
Suggestion: Use putStr (or myPutStr).

Practice exercise IP-7

Problem statement

Implement myGetLine, which works similarly to Prelude's getLine.
Suggestion: Use getChar.

Practice exercise IP-8

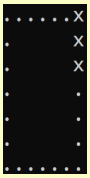
Problem statement

Implement an expanded version of patrolDot from exercise IP-3 called patrolSnake, which animates a snake moving clockwise along the perimeter of a n by n square for nTicks iterations/ticks. The snake has a length equal to $fracn2$ (rounded down) and is drawn as a set of x's. The borders from drawBorders must also be drawn, so that the snake moves "along the dots".

Usage examples:

After writing `patrolSnake 50 7` on GHCi, the first frame should be similar to the following figure:

The seventh frame should be similar to the following figure:



Practice exercise IP-9

Problem statement

Implement hangman, a function without arguments that implements a variant of the hangman game.

This game begins by asking the player 1 for a mystery word and the topic to which this word belongs (example: animals, countries). The mystery word should have at most 10 characters, which must all be lower case letters (to simplify). During this part, player 2 must not be watching the screen.

Then, the game prompts player 2 to guess the letters the compose the mystery word until all of the word is revealed. Letters that do not belong to the word are counted as "incorrect guesses". Before each guessing opportunity, the game should shown the topic and the word as set of underscore characters ("_"), with as many of them as the mystery word. The underscores are progressively replaced by the mystery word's actual letters as player 2 guesses them.

When player 2 guesses all of the letters of the word, the game ends by reporting the number of incorrect guesses.

Note: the game should perform validation on all inputs: if a player inserts a non-lowercase letter for the mystery word or a guess, the game should report it and ask the user for input again.

Usage examples:

Asking player 1 for the mystery word and topic:

```
*Main> hangman
Player 1, what is the mystery word? (max 10 lower case letters, with no spaces)
panda
Player 1, what is the topic of the mystery word?
animals
```

Before player 2 starts guessing letters:

```
Topic: animals

_ _ _ _ _
Num. of wrong attempts: 0
Player 2, type the next letter of the mystery word.
```

After player 2 guesses 'c, 'a' and 'p':

```
Topic: animals
p a _ _ a
Num. of wrong attempts: 1
Player 2, type the next letter of the mystery word.
```

Endgame, after player 2 guesses 'n' and 'd':

```
Topic: animals
p a n _ a
Num. of wrong attempts: 1
Player 2, type the next letter of the mystery word.
d
You guessed the word in 1 attempt. It was panda.
Press enter to continue.
```

Solutions to the practice exercises

Practice exercise IP-4

Solution

An error is issued since the expression attempts to concatenate a string with an IO action, which leads to a type error.

Practice exercise IP-5

Solution

```
reverseUntil :: IO ()
reverseUntil = do
    text <- getLine
    if null text
        then return ()
        else do
            putStrLn $ reverse text
            reverseUntil
```

Optionally, a redundant "do" could be added after the "then" to create a do-block with a single IO-action.

Practice exercise IP-6

Solution

a)

```
myPutStr :: String -> IO ()
myPutStr [] = return ()
myPutStr (x:xs) = do
    putChar x
    myPutStr xs
```

b)

```
myPutStrLn :: String -> IO ()
myPutStrLn str = myPutStr $ str ++ "\n"
```

Practice exercise IP-7

Solution

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

If the first statement of the else's do-block was `xs <- return myGetLine` rather than `xs <- myGetLine` , xs would have the type `IO (IO String)` , which would be incorrect.

Practice exercise IP-8

Solution

```
patrolSnake :: Int -> Int -> IO ()
patrolSnake nTicks n
    | nTicks >= 0 && n >= 0 = patrolSnakeAux nTicks n [(x,1) | x <- [(div
        ↪ n 2),(div n 2 - 1).. 1]]
    | otherwise = error "Negative argument"

patrolSnakeAux :: Int -> Int -> [Pos] -> IO ()
patrolSnakeAux 0 _ _ = return ()
patrolSnakeAux nTicks n snake = do
    cls
    drawBorders n
    writeListAt snake 'x'
    wait 100
    patrolSnakeAux (nTicks - 1) n (nextSnake snake n)

nextSnake :: [Pos] -> Int -> [Pos]
```

```
nextSnake snake n = (nextPoint (head snake) n):(init snake)
```

The program's state in this problem corresponds to a list of positions, rather than a single position like in patrolDot of exercise IP-3. The snake's "head" corresponds to the head of the state list. To update the snake's state, one can use nextPoint to determine the next position of the snake's head. The remaining elements of the snake x_i, at position i (i > 1, with indices starting at 1), correspond to the value of x_{i-1} of the previous frame/state.

Practice exercise IP-9

Solution

```
import IOUtils (cls)

hangman :: IO ()
hangman = do
    word <- readMysteryWord
    topic <- readTopic
    nWrongAttempts <- hangmanGuessLoop [] word topic
    putStrLn $
        "You␣guessed␣the␣word␣in␣" ++ show nWrongAttempts ++ "␣attempt" ++
            ↪ (if nWrongAttempts /= 1 then "s" else "") ++
        ".␣It␣was␣" ++ word ++ "."
    pressEnter

readMysteryWord :: IO (String)
readMysteryWord = do
    putStrLn "Player␣1,␣what␣is␣the␣mystery␣word?␣(max␣10␣lower␣case␣
        ↪ letters,␣with␣no␣spaces)"
    word <- getLine
    if validWord word
        then return word
        else do
            putStrLn "Invalid␣word!"
            readMysteryWord

validWord :: String -> Bool
validWord word = (length word < 10) && (all validLetter word)

validLetter :: Char -> Bool
validLetter c = c >= 'a' && c <= 'z'

readTopic :: IO (String)
readTopic = do
    putStrLn "Player␣1,␣what␣is␣the␣topic␣of␣the␣mystery␣word?"
    topic <- getLine
    return topic

hangmanGuessLoop :: [Char] -> String -> String -> IO (Int)
hangmanGuessLoop guessLetters word topic = do
    cls
    putStrLn $ "Topic:␣" ++ topic
    putStrLn $ concat $ map (\c -> if elem c guessLetters then c:"␣" else
        ↪ "_␣") word
    let nWrongAttempts = countWrongAttempts guessLetters word
    putStrLn $ "Num.␣of␣wrong␣attempts:␣" ++ show nWrongAttempts
    putStrLn "Player␣2,␣type␣the␣next␣letter␣of␣the␣mystery␣word."
    g <- getChar
```

```
    putStrLn ""
    if not $ validLetter g then do
        putStrLn "Invalid␣character.␣Must␣be␣a␣lowercase␣letter."
        pressEnter
        hangmanGuessLoop guessLetters word topic
    else if elem g guessLetters then do
        putStrLn "That␣letter␣has␣already␣been␣guessed!"
        pressEnter
        hangmanGuessLoop guessLetters word topic
    else if guessedTheWord (g:guessLetters) word then
        return nWrongAttempts
    else do
        hangmanGuessLoop (g:guessLetters) word topic

countWrongAttempts :: [Char] -> String -> Int
countWrongAttempts guessLetters word = length [c | c <- guessLetters, not
    ↪ (elem c word)]

guessedTheWord :: [Char] -> String -> Bool
guessedTheWord guessLetters word = all (\c -> elem c guessLetters) word

pressEnter :: IO ()
pressEnter = do
    putStrLn "Press␣enter␣to␣continue."
    getLine
    return ()
```

The readMysteryWord function has to return an IO (String) and not a String since the else part has a do-block in order to print "Invalid word", and both the "then" and "else" parts of an if-then-else expression must be of the same type.

The hangmanGuessLoop function performs the interaction with player 2, who must find out the mystery word. hangmanGuessLoop is an example of a function that returns a `IO Int`.

The readMysteryWord and hangmanGuessLoop function perform validation on the user input. If their input is invalid, they ask for input again.

The pressEnter function is very useful: it forces the user to press a key before moving on with the program, which allows the user to read the text on the screen before it is cleared, providing a better interaction.

6.2 File I/O

Haskell programmers can interact with files by using the withFile function from the `System.IO` module. This function has the signature

`withFile :: FilePath -> IOMode -> (Handle -> IO a)-> IO a .`

The first argument is a type syonym for a string representing the path to a file: `type FilePath = String .`

The second argument is a value from the `IOMode` type:

`data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode .` This value indicates how the program interacts with the file: it can read it (`ReadMode`), overwrite it (`WriteMode`) or add text to the end of it (`AppendMode`). `ReadWriteMode` will not be explored in this

chapter. `WriteMode` and `AppendMode` are both used to write to a file. The difference is that, with `WriteMode`, the file's contents are erased before characters are written to it.

The third and final argument is a function with the type `Handle -> IO a`. The input of this function is a handle associated with the file. It is used to indicate which file the programmer is referring to.

The output of the third argument, which is equal to the global output of the function, represents an IO action that opens the file, performs some sort of operation on it and then closes it. As a result, when using withFile, typically the third argument is a lambda (or another function) with a single argument consisting of the file's handle. Using the handle, one can define the lambda's body as a sequence of IO actions, using a do-block.

Interacting with files is very similar to performing IO with the standard input and output, since a set of similar functions are available: for input, there is `hGetChar`, `hGetLine` and `hGetContents`, and, for output, there is `hPutChar`, `hPutStr` and `hPutStrLn`. The difference between these functions and their "h-less" counterparts is that they take an additional argument with the file's handle, which is the first argument of this family of functions. Thus, while `putStrLn` has the type `String -> IO ()`, `hPutStrLn` has the type `Handle -> String -> IO ()`, and while `putChar` has the type `IO Char`, `hPutChar` has the type `Handle -> IO Char`.

The System.IO module has three useful functions to avoid using withFile and file handles on simpler scenarios:

- `readFile :: FilePath -> IO String`: reads the contents of a file.
- `writeFile :: FilePath -> String -> IO ()`: writes a string to a file opened in `WriteMode`.
- `appendFile :: FilePath -> String -> IO ()`: writes a string to a file opened in `AppendMode`.

Sample exercises

Sample exercise IP-10

Problem statement

Using withFile, implement myWriteFile, which works similarly to System.IO's writeFile.

Usage examples:

```
*Main> myWriteFile "test.txt" "Hi"
```

Solution

```
import System.IO

myWriteFile :: FilePath -> String -> IO ()
myWriteFile p str = withFile p WriteMode $ \hdl -> do
    hPutStr hdl str
```

This solution illustrates the usage of withFile to open a file in `WriteMode` . In this case, the handle function contains a single statement.

Using the `$` is very useful to avoid an annoying pair of parentheses.

Sample exercise IP-11

Problem statement

Implement copyFile, which, given two file paths fp1 and fp2, creates a copy of the file fp1 called fp2.

Usage examples: `*Main> copyFile "test.txt""test2.txt"`

Solution

```
import System.IO

-- Alternative with withFile
copyFile :: FilePath -> FilePath -> IO ()
copyFile fp1 fp2 = do
    withFile fp1 ReadMode $ \hdl1 -> do
        text <- hGetContents hdl1
        withFile fp2 WriteMode $ \hdl2 -> do
            hPutStr hdl2 text

-- Alternative with readFile and writeFile
copyFile' :: FilePath -> FilePath -> IO ()
copyFile' fp1 fp2 = do
    text <- readFile fp1
    writeFile fp2 text
```

The solution with withFile shows that withFile can be called inside the handle function of another call to withFile.

hGetContents is used to ensure all of the text contained in the file is read with little effort (rather than having to write recursive code with hGetLine's).

Practice exercises

Practice exercise IP-12

Problem statement

Implement capitalize, which, given two file paths fp1 and fp2, capitalizes all of the text in file fp1 and stores it in fp2 (overwriting its contents if file fp2 exists).

Suggestion: use the toUpper function from Data.Char.

Usage examples:

```
*Main> capitalizeFile "test.txt" "test2.txt"
```

Practice exercise IP-13

Problem statement

Using withFile, implement myAppendFile, which works similarly to System.IO's appendFile.

Usage examples:

```
*Main> appendFile "test.txt" "Hi"
```

Practice exercise IP-14

Problem statement

Implement myWc which works similarly to the wc command for Unix-based systems. The function receives a file name as argument and prints a triple indicating: the file's number of lines, the number of words (separated by newlines and spaces) and the number of bytes (number of characters, including newlines and spaces).

Suggestion: use the lines and words functions from Prelude to decompose the file's text into lines and words, respectively.

Usage examples:

```
*Main> writeFile "test.txt" "test\nHello␣there!\nHi!"
*Main> myWc "test.txt"
(3,4,21)
```

Practice exercise IP-15

Problem statement

Implement firstWords, which given a file path fp, returns the first n words that come alphabetically in file fp.

Note: The function should ignore whether the words have uppercase or lowercase letters.

Usage examples:

```
*Main> firstWords 2 "test.txt"
["cat","duck"]
*Main> firstWords 10 "test.txt"
["cat","duck","goose","sheep"]
```

Practice exercise IP-16

Problem statement

Implement checkSpelling, which given a file path fp for a text file and file path fpDict for a dictionary, marks the words of file fp that are not in file fpDict.

The function must print line by line all of the words in file fp, while coloring in red those that are not part of the dictionary, which contain a spelling/grammatical error.

Note: The function should ignore whether the words have uppercase or lowercase letters.

Usage examples: a minimalist example:

```
*Main> checkSpelling "test.txt" "dict.txt"
hello tere
hi tere
hi there
high there
```

Solutions to the practice exercises

Practice exercise IP-12

Solution

```haskell
import System.IO
import Data.Char (toUpper)

-- Alternative with withFile
capitalizeFile :: FilePath -> FilePath -> IO ()
capitalizeFile fp1 fp2 = do
    withFile fp1 ReadMode $ \hdl1 -> do
        text <- hGetContents hdl1
        withFile fp2 WriteMode $ \hdl2 -> do
            hPutStr hdl2 (map toUpper text)

-- Alternative with readFile and writeFile
```

```
capitalizeFile' :: FilePath -> FilePath -> IO ()
capitalizeFile' fp1 fp2 = do
    text <- readFile fp1
    writeFile fp2 (map toUpper text)
```

This problem and its solution are very similar to the ones of exercise IP-11.

Practice exercise IP-13

Solution

```
import System.IO

myAppendFile :: FilePath -> String -> IO ()
myAppendFile fp str = withFile fp AppendMode $ \hdl -> do
    hPutStr hdl str
```

Practice exercise IP-14

Solution

```
import System.IO

myWc :: FilePath -> IO (Int,Int,Int)
myWc fp = do
    text <- readFile fp
    let ls = lines text
    let ws = words text
    return (length ls, length ws, length text)
```

Practice exercise IP-15

Solution

```
import System.IO
import Data.List (sort,nub)
import Data.Char (toLower)

firstWords :: Int -> FilePath -> IO [String]
firstWords n fp = do
    text <- readFile fp
    return (firstWordsAux n text)

firstWordsAux :: Int -> String -> [String]
firstWordsAux n = take n
                . sort
                . nub
                . words
                . map toLower
```

Practice exercise IP-16

Solution

```
import System.IO
import IOUtils
import Data.Char (toLower)

checkSpelling :: FilePath -> FilePath -> IO ()
checkSpelling fp fpDict = do
    text <- readFile fp
    dict <- readFile fpDict
    processLines (map words (lines (map toLower text))) (words dict)


processLines :: [[String]] -> [String] -> IO ()
processLines [] _ = return ()
processLines (l:ls) dict = do
    processWords l dict
    putStrLn ""
    processLines ls dict

processWords :: [String] -> [String] -> IO ()
processWords [] _ = return ()
processWords (w:ws) dict = do
    if elem w dict
        then putStr w
        else do
            clrFg Red
            putStr w
            clrReset
    putStr "␣"
    processWords ws dict
```

The performance of the solution can be improved by storing the words of the dictionary in a more efficient data structure (such as a trie).

6.3 Pseudo-random value generation

Note: Haskell code written in this section will use the `System.Random` module. This module is not present by default in many Haskell systems, so the user may need to add this module. If the module is not present in the Haskell system, Windows users may try installing a Haskell interpreter in the Windows Subsystem for Linux (WSL).

As in other programming languages, randomly generated value in Haskell are actually pseudo-random since they rely on algorithms rather than on actual physical unpredictable sources (such as atmospheric noise).

Pseudo-random number generation code is usually impure as it typically updates a global variable responsible for producing the pseudo-random values. If this global generator was not updated, then it would always produce the same value. An alternative to impure code would be to return a new value generator (since variables are immutable in Haskell) every time a value is produced and pass this generator to all

the sections of the code that require random values. This, however, would be quite cumbersome.

Random values can be produced using a partial application of the `randomR` function which is passed as the sole argument of `getStdRandom`. A pair is passed to `randomR` with the lower and upper bounds for the random value to be generated (the generated value can be equal to either of the bounds). These bounds must belong to a type that is an instance of the `Random` typeclass, such as `Int`, `Bool` and `Char`. Examples:

```
Prelude> import System.Random
Prelude System.Random> getStdRandom (randomR (1,10))
5
Prelude System.Random> getStdRandom (randomR (1,10))
10
Prelude System.Random> getStdRandom (randomR (1,10))
1

Prelude System.Random> getStdRandom (randomR (False,True))
False

Prelude System.Random> getStdRandom (randomR (0.0,1.0))
0.22508491975707723
Prelude System.Random> getStdRandom (randomR (0.0,1.0))
0.6610123310684166

Prelude System.Random> do num <- getStdRandom (randomR (1,10)); putStrLn $ "Num
    ↪ ␣=␣" ++ (show num)
Num = 6
```

As it can be seen in the examples above, using `getStdRandom` alongside `randomR` is quite flexible, since it can generate random values of various types (which is why this section refers to "pseudo-random value generation", rather than "pseudo-random number generation").

Sample exercises

Sample exercise IP-17

Problem statement

Implement headsNTails, which given an integer n, performs an IO action to return a list of n randomly generated booleans.

Usage examples:

```
*Main> headsNTails 4
[False,False,True,True]
*Main> headsNTails 5
[False,True,False,False,False]

*Main> do coins <- headsNTails 3; putStrLn $ "There␣are␣" ++ (show $
    ↪ length $ filter id coins) ++ "␣heads."
There are 1 heads.
```

Solution

```
import System.IO
import System.Random

headsNTails :: (Integral a) => a -> IO [Bool]
headsNTails 0 = return []
headsNTails n
    | n > 0 = do
        coin <- getStdRandom (randomR (False,True))
        coins <- headsNTails (n-1)
        return (coin:coins)
    | otherwise = error "Negative number"
```

The solution returns an IO action that holds a list of booleans.

The function's behavior does not change if the order of the booleans in the pairs passed to `randomR` is reversed.

Practice exercises

Practice exercise IP-18

Problem statement

Implement highLo, which implements a number guessing game, played between the user and the computer. It receives a single argument, nMax, an integer representing the highest possible value for the "mystery number".

The game begins with the computer randomly choosing an integer between 0 and nMax. Then, the user must guess this number. If the user guesses correctly, then the game immediately ends and the number of tries needed to find out the mystery number is reported. If the guess is wrong, then the computer must tell the user if the mystery number is higher or lower than the guess.

Note: the function should check if the user's input is a positive integer (i.e it is composed solely of digits).

Usage examples:

```
*Main> highLo 20
Current number of tries: 0
What's your guess?
10
Lower
Current number of tries: 1
What's your guess?
5
Lower
Current number of tries: 2
What's your guess?
2
Guessed in 3 tries.
```

Practice exercise IP-19

Problem statement

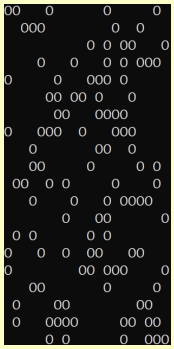
Implement rain, which continuously prints randomly generated raindrops that gradually fall along the screen. The function receives as argument the probability pb of creating a new raindrop at each position of the topmost row.

A raindrop is represented by an 'O' in a (x,y) position of a 2D grid. At each iteration/step, all raindrops move one position downwards (i.e. their y coordinate increases by 1). At each step, for each position (x,1) (with x between 1 and the screen's width), there is a probability pb that a new raindrop is created.

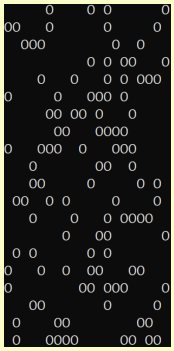
Note: when raindrops reach the bottom of the screen, they should be removed.

Suggestion: To simplify, define two constants in the source code file for the screen's width and height (rather than having to pass them to the rain function and its auxiliary functions).

Usage examples: In a 20 by 20 screen, after writing `rain 0.3` on GHCi and waiting for some iterations:



After one more iteration:



Solutions to the practice exercises

Practice exercise IP-18

Solution

```
import System.IO
import System.Random

highLo :: (Integral a, Read a, Random a) => a -> IO ()
highLo nMax
    | nMax >= 1 = do
        mysteryNum <- getStdRandom (randomR (1,nMax))
        nTries <- highLoLoop 0 mysteryNum
        putStrLn $ "Guessed␣in␣" ++ (show nTries) ++ "␣tries."
    | otherwise = error "Upper␣bound␣must␣be␣greater␣than␣0"

highLoLoop :: (Integral a, Read a, Random a) => Int -> a -> IO Int
highLoLoop nCurTries mysteryNum = do
    putStrLn $ "Current␣number␣of␣tries:␣" ++ (show nCurTries)
    putStrLn "What's␣your␣guess?"
    text <- getLine
    if isPositiveNum text then do
        let guess = read text
        if mysteryNum == guess then
            return (nCurTries + 1)
        else do
            putStrLn $ if mysteryNum > guess then "Higher" else "Lower"
            highLoLoop (nCurTries + 1) mysteryNum
    else do
        putStrLn "Invalid␣number"
        highLoLoop nCurTries mysteryNum

isPositiveNum :: String -> Bool
isPositiveNum = all (\c -> c >= '0' && c <= '9')
```

The auxiliary function highLoLoop implements the game's loop and returns an IO action with the number of tries needed to guess the number.

isPositiveNum tests if the user's input is solely composed of digits.

The input argument must belong to the `Integral`, `Read` and `Show` typeclasses, since: it is an integer, string values must be convertible to the same type as it (via `read`) and random values of its type must be generatable.

Practice exercise IP-19

Solution

```
import System.IO
import System.Random
import IOUtils

width = 20
height = 20

rain :: (RealFloat a, Random a) => a -> IO ()
```

```
rain prob = rainLoop prob []

type Drop = (Int,Int)

rainLoop :: (RealFloat a, Random a) => a -> [Drop] -> IO ()
rainLoop prob drops = do
    newDrops <- createDrops 1 prob
    let drops' = newDrops ++ drops
    cls
    writeListAt drops' 'O'
    wait 100
    rainLoop prob (removeOldDrops (moveDrops drops'))

createDrops :: (RealFloat a, Random a) => Int -> a -> IO [Drop]
createDrops x prob
    | x > width = return []
    | otherwise = do
        newDrops <- createDrops (x+1) prob
        rand <- getStdRandom (randomR (0,1))
        if rand <= prob then
            return ((x,1):newDrops)
        else
            return newDrops

moveDrops :: [Drop] -> [Drop]
moveDrops = map (\(x,y) -> (x,y+1))

removeOldDrops :: [Drop] -> [Drop]
removeOldDrops = filter (\(_,y) -> y <= height)
```

The rain function immediately invoke the rainLoop auxiliary function to perform the actual loop for the animation. The loop was placed in the auxiliary function so that it can work with an additional variable that stores the animation's state, which, in this case, is a list with the positions of all the raindrops.

An iteration of the loop performs the following steps:

- Random creation of new raindrops with moveDrops.
- Rendering the program's state and waiting a small amount of time so the user can properly enjoy the animation.
- Moving all the drops one unit downwards.
- Removing all raindrops that go out of bounds (to avoid unnecessary computations in the next iterations).

6.4 Brief description of monads

Monads are an abstraction where a value is wrapped inside a structure that provides a context to it. Up until now, this book has covered three monads: lists, Maybe and IO. The list type is an instance of Monad where the context is a computation that yields a variable amount of results (i.e. non-deterministic computations). Maybe provides the context that the computation may or may not have failed, and that, if it suceeded

then the result is the value wrapped inside of it. Finally, IO provides the context where the value wrapped inside it was obtained via side-effects.

Monads have three essential ingredients:

- A type constructor that builds a monadic type.
  Example: `Maybe` in `Maybe 1`.
- A type converter, that wraps a variable inside a monad. This corresponds to the `return` function in Haskell, which has the type Monad m => a -> m a.
  Example: if a function returns an `IO String`, then if it has an expression with `return "hello"`, it produces `IO "hello"`.
- A combinator, which is used to chain monadic functions. Intuitively, they "unwrap" a monad's internal value, then apply it to a monadic function, which receives an "unwrapped" value and return a "wrapped value". This is achieved using the bind operator (`>>=`) in Haskell, which has the type `(>>=) :: Monad m =>m ↪ a -> (a -> m b)-> m b` (the second argument of `>>=` is the monadic function). The do expression is actually syntactic sugar for `>>=`.
  Example: the following function:

```
readNWrite :: IO ()
readNWrite = do
    text <- getLine
    putStrLn text
```

is equivalent to:

```
readNWrite' :: IO ()
readNWrite' = getLine >>= putStrLn
```

In general, monads are useful to eliminate boilerplate code (i.e. code fragments that are repeated in multiple section with little to no variation) and to define structures that behave similarly to code from imperative languages (functions with side-effects, states ...) without breaking the purity of a functional language.

6.5 Case study: Game of Life

The Game of Life was invented by the mathematician John Conway in 1970. It is a game without players played within a cellular automaton: an intial configuration is given to a 2D grid of cells that can either be alive or dead, then the system evolves in a deterministic manner without interaction with the outside environment. It is a very interesting game from a theoretical point-of-view as it shows how complex, chaotic patterns can be created using a very small and simple set of rules.

At a given instant / generation, each cell in the 2D grid of the world can either be alive or dead. The state of the cell in the next generation is determined by its 8 neighbors:

- If a live cell has fewer than 2 live neighbors, then it dies due to *underpopulation*.
- If a live cell has more than 3 live neighbors, then it dies due to *overpopulation*.

- If a dead cell has exactly 3 live neighbors, then it becomes a live cell due to *reproduction*.
- In all other cases, a cell's state remains *unchanged*.

It should be noted that all cells have exactly 8 neighbors since the 2D grid wraps around itself: each cell of the leftmost column is adjacent to the one in the same row in the right rightmost column, and the ones in the top row are adjacent to the ones in the same column in the bottom row.

This section will guide the reader into developing, step-by-step an implementation of the Game of Life in Haskell. The game will be displayed as a 2D grid where live cells are represented by an 'O', and dead cells by a '.'. The game will make use of some colors to help illustrate how the cells evolve overtime. The initial configuration of the world will be provided via three alternatives: a list with all the live cells, a file with the state of the world or via random generation.

An example of the representation of the game is presented below.

As a starting point, consider the source code below, which includes the import statements, some type and functions definitions and "global variables" for the screen's width and height (similarly to what was done in exercise IP-19).

```
import System.IO
import System.Random
import IOUtils

type Pos = (Int,Int)
type CellState = Bool
data Outcome = Underpopulatation | Overpopulation | Birth | Unchanged
```

```
    deriving (Eq,Show,Enum)
type State = [[CellState]]
type StateOutcomes = [[(CellState,Outcome)]]

width = 60
height = 20

wrap :: Pos -> Pos
wrap (x,y) = (normalize x width, normalize y height)
    where normalize v len = 1 + mod (v-1) len
```

A cell' state, represented by `CellState` is a boolean indicating whether the cell is alive or not. `Outcome` is a type containing all the possible events that can occur in a cell between consecutive generations/iterations. `State` represents the state of the game at a given iteration. `StateOutcome` is a type that groups a cell's state and its outcome in a given generation.

The wrap function is used to convert a position that can be potentially out of bounds to a position inside the board. This function will make it much easier to determine the 8 neighbors of a cell.

Practice exercises

The goal of the exercises of this section is it define all the functions needed to implement the Game of Life. Rather than presenting use cases (which would not be practical for some of the functions), the function's type declaration will be provided.

For each exercise, the reader should use the initial source code for the game as well as their solutions to all the preceding exercises of this section.

Practice exercise IP-20

Problem statement

Implement isAlive, which, given a state and a position p on the grid, determines if a cell in p is alive or not.

```
isAlive :: State -> Pos -> CellState
```

Practice exercise IP-21

Problem statement

Implement neighbors, which, given a state and a position p on the grid, returns the 8 neighboring cells of p.

Suggestion: use the wrap function.

```
neighbors :: State -> Pos -> [CellState]
```

Practice exercise IP-22

Problem statement

Implement getStateOutcome, which, given a state and a position p on the grid, returns a pair with cell p's state and its outcome in the following generation.

```
getStateOutcome :: State -> Pos -> (CellState,Outcome)
```

Practice exercise IP-23

Problem statement

Implement getStateOutcomes, which, given a state, computes the corresponding `StateOutcomes` structure.

```
getStateOutcomes :: State -> StateOutcomes
```

Practice exercise IP-24

Problem statement

Implement nextState, which, given a `StateOutcomes` structure, returns a `State` object presenting the state of the next generation.

```
nextState :: StateOutcomes -> State
```

Practice exercise IP-25

Problem statement

Implement drawStateOutcomes, which, given a `StateOutcomes` structure, draws the 2D grid of given generation in the screen.

A live cell is represented by an 'O', and a dead cell, by a '.'.

Colors represent a cell's outcome: magenta for underpopulation, red for overpopulation, green for birth and white for unchanged cells.

```
drawStateOutcomes :: StateOutcomes -> Int -> IO ()
```

Practice exercise IP-26

Problem statement

Implement lifeLoop, which performs the game's loop. It receives as input the number of generations the game will last and the initial state, and returns

an IO action that holds the latest state, which could then be read into a file. The last generation, which is returned by the function, is not drawn.

In each iteration, the current generation should be printed in the screen, then the function should halt for some milliseconds. Finally the next state should be computed.

Note: It is not necessary to clear the screen in this function. The call to `cls` will be performed by the functions that will be implemented in the following exercises.

Suggestion: call the getStateOutcomes function only once per iteration.

```
lifeLoop :: Int -> State -> IO State
```

The last three functions represent the "entry points" to the game. Each function initializes the game's state in a different manner. All of these functions perform three steps: they obtain the initial state, then they clear the screen, then they call lifeLoop. Their first argument is the same as the one for lifeLoop.

Practice exercise IP-27

Problem statement

Implement lifeList, which initializes the game's state using a list of positions that indicate where the live cells are located.

```
lifeList :: Int -> [Pos] -> IO State
```

Practice exercise IP-28

Problem statement

Implement lifeFile, which initializes the game's state by reading it from a file.

Note: test cases for this function can be generated by writing the state returned by lifeLoop to a file.

```
lifeFile :: Int -> FilePath -> IO State
```

Practice exercise IP-29

Problem statement

Implement lifeRand, which initializes the game's state by randomly setting each cell to be alive or dead. The function's second argument is the probability of a cell being set to be alive.

```
lifeRand :: (RealFloat a, Random a)=> Int -> a -> IO State
```

Solutions to the practice exercises

Practice exercise IP-20

Solution

```
isAlive :: State -> Pos -> CellState
isAlive st (x,y) = st !! (y-1) !! (x-1)
```

The function uses the `(!!)` operator to obtain an element of a 2D list.

Practice exercise IP-21

Solution

```
neighbors :: State -> Pos -> [CellState]
neighbors st (x,y) = [isAlive st (wrap (x + dx,y + dy)) | dx <- [-1,0,1],
    ↪ dy <- [-1,0,1], dx /= 0 || dy /= 0]
```

The 8 neighbors are obtained using a list comprehension that considers offset of up to one unit in the x and y axes. A guard is used to prevent the cell's owns position to be added to the list of neighbors.

Practice exercise IP-22

Solution

```
getStateOutcome :: State -> Pos -> (CellState,Outcome)
getStateOutcome st pos
    | alive && (nLiveNeighbors < 2) = (alive,Underpopulatation)
    | alive && (nLiveNeighbors > 3) = (alive,Overpopulation)
    | (not alive) && (nLiveNeighbors == 3) = (alive,Birth)
    | otherwise = (alive,Unchanged)
    where nLiveNeighbors = length $ filter id (neighbors st pos)
          alive = isAlive st pos
```

The function contains a clause for each possible outcome.

Practice exercise IP-23

Solution

```
getStateOutcomes :: State -> StateOutcomes
getStateOutcomes st = [[getStateOutcome st (x,y)| x <- [1 .. width]] | y
    ↪ <- [1 .. height]]
```

The function is implemented using a list comprehension and getStateOutcome.

Practice exercise IP-24

Solution

```
nextState :: StateOutcomes -> State
nextState [] = []
nextState (l:ls) = (nextLine l):(nextState ls)

nextLine :: [(CellState,Outcome)] -> [CellState]
nextLine [] = []
nextLine (c:cs) = (nextCellState c):(nextLine cs)

nextCellState :: (CellState,Outcome) -> CellState
nextCellState (s,Unchanged) = s
nextCellState (_,Underpopulatation) = False
nextCellState (_,Overpopulation) = False
nextCellState (_,Birth) = True
```

Recursion is used to build the 2D list representing the new state.

nextCellState has a clause for each possible outcome to determine the next state of a cell.

Practice exercise IP-25

Solution

```
drawStateOutcomes :: StateOutcomes -> IO ()
drawStateOutcomes so = drawLines so 1

drawLines :: StateOutcomes -> Int -> IO ()
drawLines [] _ = return ()
drawLines (l:ls) y = do
    drawLine l y 1
    drawLines ls (y+1)

drawLine :: [(CellState,Outcome)] -> Int -> Int -> IO ()
drawLine [] _ _ = return ()
drawLine (c:cs) y x = do
    drawCell c y x
    drawLine cs y (x+1)

drawCell :: (CellState,Outcome) -> Int -> Int -> IO ()
drawCell (s,o) y x = do
    clrFg (outcome2Color o)
    writeAt (x,y) (cellState2Char s)
    clrReset

cellState2Char :: CellState -> Char
cellState2Char False = '.'
cellState2Char True = 'O'

outcome2Color :: Outcome -> Color
outcome2Color Underpopulatation = Magenta
```

```
outcome2Color Overpopulation = Red
outcome2Color Unchanged = White
outcome2Color Birth = Green
```

Similarly to nextState, this function uses double recursion. However, unlike the previous function, this one requires keeping track of each cell's position so that it is possible to indicate where the next colored character should be drawn in the screen.

In relation to drawStateOutcomes, drawLines uses an additional argument with the index y of the next row to draw.

Practice exercise IP-26

Solution

```
lifeLoop :: Int -> State -> IO State
lifeLoop 0 st = return st
lifeLoop nGens st = do
    let stateOutcomes = getStateOutcomes st
    drawStateOutcomes stateOutcomes
    wait 200
    lifeLoop (nGens - 1) (nextState stateOutcomes)
```

The whole reason why this solution defines a `StateOutcomes` structure and the getStateOutcomes function is to avoid having lifeLoop needing to compute the outcomes twice for each iteration: to draw the current generation in the screen and to compute the next generation.

Practice exercise IP-27

Solution

```
lifeList :: Int -> [Pos] -> IO State
lifeList nGens l = do
    let st = [[elem (x,y) l| x <- [1 .. width]] | y <- [1 .. height]]
    cls
    lifeLoop nGens st
```

Practice exercise IP-28

Solution

```
lifeFile :: Int -> FilePath -> IO State
lifeFile nGens fp = do
    text <- readFile fp
    cls
    lifeLoop nGens (read text)
```

Practice exercise IP-29

Solution

```
lifeRand :: (RealFloat a, Random a) => Int -> a -> IO State
lifeRand nGens prob = do
    st <- initRandMatrix prob 1
    cls
    lifeLoop nGens st

initRandMatrix :: (RealFloat a, Random a) => a -> Int -> IO State
initRandMatrix prob y
    | y > height = return []
    | otherwise = do
        line <- initRandLine prob y 1
        lines <- initRandMatrix prob (y+1)
        return (line:lines)

initRandLine :: (RealFloat a, Random a) => a -> Int -> Int -> IO [
    ↪ CellState]
initRandLine prob y x
    | x > width = return []
    | otherwise = do
        rand <- getStdRandom (randomR (0,1))
        let cell = rand <= prob
        line <- initRandLine prob y (x+1)
        return (cell:line)
```