CS_191 Functional Programming I

Ulrich Berger

Department of Computer Science
Swansea University
Spring 2011

u.berger@swan.ac.uk , http://www.cs.swan.ac.uk/∼csulrich/
tel 513380, fax 295708, room 306, Faraday Tower

1 Introduction

This course gives a first introduction to functional programming. No prior programming knowledge is assumed. The emphasis of the course is not on a full coverage of a particular programming language, but on general functional programming skills which can be applied in any functional and most main-stream programming languages.

The course is mainly based on material from the following books:

- Simon Thompson. *Haskell: The Craft of Functional Programming*, 2nd edition. Addison-Wesley, 1999.
- Graham Hutton. *Functional Programming in Haskell*. Cambridge University Press, 2007.
- Richard Bird. *Introduction to Functional Programming using Haskell*, 2nd edition. Prentice Hall, 1998.
- Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.
- Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O'Reilly, 2008. `http://book.realworldhaskell.org`
- Christoph Lüth. Praktische Informatik 3 (lecture slides). University of Bremen[1].

More information about books and online tutorials on functional programming can be found on the course web page `http://www.c.swa.ac.uk/∼csulrich/fp1.html`

[1] developed as part of the MMiSS project (MultiMedia Instruction in Safe Systems) and financially supported by the German Ministry for Education and Research

1.1 The fundamental ideas of functional programming

What is a functional program?

- A functional program is an *expression*.

  Example: `(2 * 3) + 4`

- To *run* a functional program means to *evaluate* it.

  Example: `(2 * 3) + 4` evaluates to `10`.

- There are *no assignments* (such as `x = x + 1`).

One can say that functional programming is similar to using a calculator.

Expressions are built from *functions* and other expressions by *function application*. In the example above, the expression `(2 * 3) + 4` consists of the function addition `+` applied to the expressions `2 * 3` and `4`. The expression `2 * 3` in turn consists of the function multiplication `*` applied to the expressions `2` and `3`.

As with advanced calculators it possible to define new functions. For example, one can define the squaring function

```
square x = x * x
```

which can then be used, for example, in the expression `square (square (9 + 1))`.

Defining new functions from old ones is the main activity in functional programming, hence the name.

Here is an example highlighting the difference between functional and other styles of programming.

Example: Summing up the integers 1 to 10

In Java:

```
total = 0;
for (i = 1; i <= 10; ++i)
total = total+i;
```

The computation method is *assignment*.

In the functional language Haskell:

```
sum [1..10]
```

The computation method is *function application*

In the Haskell expression `sum [1..10]`

- `sum` is a function that computes the sum of the elements of a given list,
- the expression `[1..10]` evaluates to the list containing the integers 1 to 10,
- the whole expression `sum [1..10]` is the application of the function `sum` to the expression `[1..10]`, it evaluates to the integer 55.

Let us try to draw an analogy between different programming styles and building a house:

> An imperative (C, Java, . . .) program correspond to detailed instructions of what to *do* to build a house and in which *order*.
>
> A functional program on the other hand corresponds to the architect's *construction plan*. Details of the execution of the program (plan) are left to the compiler (builder).

For example, in the functional expression `(3 * 4) + (2 * 5)` it is neither specified which of the subexpressions, `3 * 4` or `5 * 2`, is evaluated first, nor is it specified where the intermediate results are stored. These details are simply left to the compiler.

A very important property of functional programs is that, because there are no assignments or other side effects, *the values of variables are never changed*. In other words, *there is no state*. Therefore, *the value of a complex expression depends only on the values of its subexpressions* and nothing else. Due to this fact it is much easier to prove the *correctness* of a functional program than that of an imperative program.

Strengths of functional programming

- Simplicity and clarity (easy to learn, readable code)
- Reliability (correctness of a program can be proven),
- Productivity (algorithmic problems can be solved quickly).
- Maintainability and adaptability (programs can easily be adapted to changing requirements without introducing errors).
- Ericsson measured an improvement factor in productivity and reliability of between 9 and 25 in experiments on telephony software.

Because of these advantages functional programming is becoming more and more widespread. Typical application areas are Artificial Intelligence, Scientific computation, Theorem proving, Program verification, Safety critical systems, Web programming, Network toolkits and applications, XML parsing, Natural Language processing and speech recognition, Data bases, Telecommunication, Graphic programming, Finance and trading.

1.2 From Lambda-calculus to functional programming

In this course we use the functional programming language *Haskell*.

`http://www.haskell.org/`

Haskell is named after *Haskell B Curry*, an American Mathematician and major contributor to the theoretical foundation of functional programming.

Haskell B Curry (1900 – 1982)

Source: `http://www-groups.dcs.st-and.ac.uk/~history`

Curry and Alonzo Church (1903-1995) developed two closely related systems, *Combinatory Logic* and *Lambda-calculus*, which are based on the fundamental concept of a *function*. Their motivation was to build an alternative foundation for mathematics (which is usually founded on the notion of a *set*).

Today, Combinatory Logic and Lambda-calculus play an important role in many branches of theoretical computer science and they are the conceptual basis of functional programming.

We will not study Combinatory Logic and Lambda-calculus in this course, but if you have understood functional programming you have understood large parts of these theories.

Short history of functional programming

- Foundations 1920–30

  Combinatory Logic and Lambda-calculus (Schönfinkel, Curry, Church)

- First functional languages 1960

  LISP (McCarthy), ISWIM (Landin)

- Further functional languages 1970–80

  FP (Backus); ML (Milner, Gordon), later SML and CAML; Hope (Burstall); Miranda (Turner)

- 1990: Haskell, Clean

Haskell has a polymorphic type system (variable types), a lazy semantics (this means that the evaluation of subexpressions is delayed as long as possible), and input/output is modelled by monads (in essence this means that expressions performing input/output are tagged and therefore clearly distinguished from other expressions).

The main current functional programming languages

Language	Types	Evaluation	I/O
Lisp, Scheme	type free	eager	via side effects
ML, CAML	polymorphic	eager	via side effects
Haskell, Clean	polymorphic	lazy	via monads

1.3 Organisation of the course

The lectures for this course will be supported by weekly lab classes where students may ask questions and test their skills by solving simple practical exercises. These exercises have to be completed and are assessed during that class. In the lab exercises and the courseworks it is recommended to *work in pairs*. This means i.p. that a coursework submission may be signed by up to two students. In this case it is assumed that both students have made approximately equal contributions.

Lab classes and exams

- **Lab classes**

  Attendance is compulsory and will be monitored.

  Lab exercises are compulsory and are assessed (10%).

  Place and time: Will be announced in the lectures.

- **Coursework**

  There will be **2 courseworks**, each worth 10%.

  Deadlines are strict! Solutions will be handed out on the first Thursday after each submission date.

  Submission: please put printout with signature(s) in the wooden box on the second floor. The box will be emptied in the morning after the submission date.

2 Getting started

This section explains how to work interactively with Haskell. Furthermore, some basic principles of Haskell's syntax are discussed.

2.1 Interactive Haskell

ghci

We will work with an interactive versions of Haskell. This makes it particularly easy to learn the language and test programs.

- ghci (Glasgow Haskell Compiler Interactive) comes with the GHC distribution. It is installed on the Linux computers in the labs.

Information on how to download, install and start ghci and can be found on the course web page.

We start ghci and evaluate some expressions:

```
>ghci
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.6.1, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Prelude> 2 + 3
5
Prelude> (2 + 3) * 5
25
Prelude> max 4 7
7
```

2.2 Haskell scripts

We can write definitions of expressions into a *script*, that is, a text file and load the script.

The file must have the extension `.hs` (for "Haskell script"), for example `test.hs`

```
x :: Int
x = 2 + 3

y :: Int
y = x * 10
```

Definitions in Haskell

A Haskell definition consists of

a *type declaration* (or signature)

```
x ::  Int
```

the *defining equation*

```
x = 2 + 3
```

Both, type declaration and defining equation must start at the beginning of a line!

Loading and testing a script.

```
Prelude> :load test.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
Main> x
5
Main> y
50
Main> x - y
-45
Main> x * x
25
Main> z
<interactive>:1:0: Not in scope: ‘z’
```

Developing a script and terminating a session.

We can extend our script by typing into the file `test.hs` more definitions:

```
area :: Float
area = 1.5^2 * pi
```

Before testing the new definitions we have to reload the file

```
Main> :reload
Main> area
7.068583
Main>:quit
```

- The command `:quit` terminates the session.
- The command `:?` lists all available commands.
- Commands can be abbreviated by their first letters.

A good script contains plenty of comments.

```
-- My first Haskell script
-- Here are my first Haskell definitions

a :: Float
a = 3.5

b :: Float
b = -7.1

{- Here I try then out (copy and paste from the terminal)
*Main> a
3.5
*Main> b
-7.1
*Main> a + b
-3.6
*Main> b^2
-}
-- End of script
```

2.3 Functions

Defining a function

We can define a *function* that computes the area of a circle for a given radius `r`.

```
ar :: Float -> Float
ar r = r^2 * pi
```

The signature `ar :: Float -> Float` means that `ar` is a function that expects a floating point number as input and returns a floating point number as output.

The variable `r` is called *formal parameter*. It represents an arbitrary input.

Instead of "input of a function" one often says "argument of a function".

```
*Main> ar 1.5
7.0685835
```

Functions of more than one argument

```
average :: Float -> Float -> Float
average x y = (x+y)/2
```

The signature `average :: Float -> Float -> Float` means that `average` is a function that expects two floating point numbers as input (represented by the formal parameters `x` and `y`) and returns a floating point number as output.

```
*Main> average 3 4
3.5
*Main> average (3,4)

<interactive>:1:8:
    Couldn't match expected type 'Float' against inferred type  ...
```

The second function call is incorrect since the function `average` expects two arguments which must be separated by spaces (no brackets, no comma).

Exercise 1 Define a function that computes the average of three numbers.

Exercise 2 Define a function that computes the hypotenuse c of a right-angled triangle from its catheti (legs) a and b using the formula

$$c = \sqrt{a^2 + b^2}$$

You may use the predefined Haskell function `sqrt :: Float -> Float` that computes the square root of a non-negative floating point number.

Exercise 3 Define a function `tri-area` that computes the area of the triangle above from its catheti a, b.

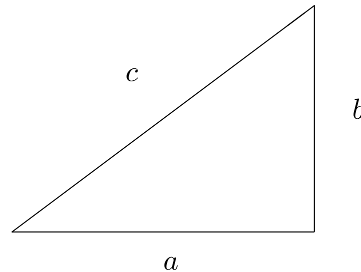


Figure 1: A right-angled triangle with catheti a, b and hypotenuse c.

Exercise 4 Define a function `tri-circum` that computes the circumference of the triangle above from its catheti a, b.

Exercise 5 Define a function `rect-area` that computes the area of a rectangle above from its sides a, b.

Exercise 6 Define a function `rect-circum` that computes the circumference of a rectangle above from its sides a, b.

Exercise 7 Visit the course web page

`http://www.cs.swan.ac.uk/~csulrich/fp1.html`

in particular check out the links

`Instructions on how to get started in the labs`

`Haskell predefinded functions and type classes`

2.4 Syntactic issues

While in mathematics the arguments of a function are usually enclosed in brackets and separated by commas, in Haskell arguments are written without brackets and separated by spaces. Brackets are only used to resolve syntactic ambiguities, for example to distinguish the expression `(3+4)*5` from `3 + (4 * 5)`.

Mathematical notation versus Haskell syntax

Maths	Haskell
$\max(x, y)$	`max x y`
$3(x^2 + y^2) + 5$	`3 * (x^2 + y^2) + 5`
$\text{ar} : \text{Float} \to \text{Float}$	`ar :: Float -> Float`
$\text{ar}(a, b) = \sqrt{a^2 + b^2}$	`ar a b = sqrt (a^2 + b^2)`

The preference rules for mathematical operations are as usual (multiplication binds stronger than addition, exponentiation binds stronger than multiplication). For example, the expression `(3^2) + (4 * 5)` can be written equivalently as `3^2 + 4 * 5`.

Haskell ignores spaces between operators with special names (such as `+, *, ^`) and arguments. For example the expressions `3 + 4` and `3+4` are the same for Haskell.

Case sensitivity, prefix vs. infix

- Haskell is case sensitive:

  identifiers, function names and formal parameters must be lower cases (`x` and `area`, but not `X` and not `Area`)

  types must be upper case (`Float`, but not `float`)

- Prefix operators bind stronger than infix operators:

  `max 3 4 + 5` means `(max 3 4) + 5`
  but not `max 3 (4 + 5)`.

Which editor should I use for writing a Haskell script?

In principle any text editor can be used. It is important that a font is used which displays each character (including space) with equal width.

A highly recommended editor is **Emacs** (or XEmacs). Emacs is installed on the Linux machines in the lab.

The use of emacs in the labs is compulsory.

Other editors, in particular inadequate ones like Notepad will not be tolerated.

Instructions on how to use Emacs and how to download Emacs on your own computer can be found on the course web page.

Emacs recognises Haskell scripts and assists, for example, in checking brackets.

It is recommended not to use tabs.

Exercise 8 Familiarise yourself with the use of emacs in the labs.

Further Exercises

Exercise 9 Define a function `mark` that computes the final mark for cs_191 from its components.

The function should take as as inputs

- The mark for the lab exercises (counts 10%),
- The mark for Coursework 1 (counts 10%),
- The mark for Coursework 2 (counts 10%),
- The mark for the exam (counts 70%).

All marks (for the components and the final mark) should be given in percent, that is, as a floating point number between 0 and 100.

Don't forget to define the signature of your function.

Exercise 10 Continuing Exercise 9, define a function `exam` that takes as inputs marks for the labs and the courseworks and computes the minimum mark sufficient for getting a final mark of at least 40%.

3 Types

In this Chapter we introduce Haskell's most common basic types and their associated predefined functions.

What are types?

A type is a name for a collection of similar objects.

There are

- *predefined primitive types* such as Bool, Int, Integer, Float, Double, Char (these will be discussed next)
- *complex types* such as pairs, lists, arrays, function types
- *user defined types*

Haskell is a *strongly typed* programming language, that is, Haskell offers type safety.

Haskell does *static type checking*, that is, types are checked during compile time.

For first users Haskell's type checker may appear a bit fussy (many type errors), but the strict typing discipline has many advantages.

Why types?

- Early detection of errors at compile time
- Compiler can use type information to improve efficiency
- Type signatures facilitate program development
- and make programs more readable
- Types increase productivity and security

3.1 Booleans

The type `Bool` of Boolean values (named after George Boole (1815-1864) an English mathematician and philosopher) contains exactly two values, `True` and `False`. It is the second simplest Haskell type. There is one even simpler type, `()`, called "unit type", which contains only one value. This type will be discussed later.

Booleans

- Values: `True` and `False` (uppercase!)
- Predefined functions:

```
not  :: Bool -> Bool              negation (not)
(&&) :: Bool -> Bool -> Bool      conjunction (and)
(||) :: Bool -> Bool -> Bool      disjunction (or)
```

Here is how these functions behave ("⇒" means "evaluates to"):

```
not True         ⇒  False
not False        ⇒  True

True && True     ⇒  True
True && False    ⇒  False
False && True    ⇒  False
False && False   ⇒  False

True || True     ⇒  True
True || False    ⇒  True
False || True    ⇒  True
False || False   ⇒  False
```

Example: Exclusive Or

The predefined function `||` defines *Inclusive Or*:

`x || y` evaluates to `True` if `x`, or `y`, or both are `True`.

We can define *Exclusive Or* using the predefined functions `||`, `&&` and `not`:

```
(|*|) :: Bool -> Bool -> Bool
x |*| y  =  (x || y)  &&  not (x && y)
```

For example:

```
True |*| False  ⇒  True
True |*| True   ⇒  False
```

but

```
True || True  ⇒  True
```

Using infix operators in prefix notation and vice versa.

We may use the operator `|*|` as well in prefix notation. For this to work, the operator has to be enclosed in round brackets. For example, we may write `(|*|) True True` which for Haskell is the same as `True |*| True`.

On the other hand, the maximum operator `max` may be used in infix notation if enclosed by inverted commas, for example, `3 ‘max‘ 4` is equivalent to `max 3 4`.

Here is the general rule for infix and prefix notation:

Infix vs. prefix

A function of two arguments whose name is composed from the special symbols

`!,#,$,%,&,*,+,-,/,<,=,>,?,@,^,|`

for example `||`, can be used

- infix without brackets: `x || y`
- prefix with brackets: `(||) x y`

A function of two arguments whose name is composed of letters, for example `max`, can be used

- infix with inverted commas: `x ‘max‘ y`
- prefix without inverted commas: `max x y`

Beware of the difference: `x ‘max‘ y` (correct) vs. `x ’max’ y` (incorrect).

In a type signature the prefix version of a name must be used:

`(||) :: Bool -> Bool -> Bool`.

Exercise 11 Define *Exclusive Or* using the name `xor` and using prefix notation throughout the definition.

Solution.

```
xor :: Bool -> Bool -> Bool
xor x y  =  (&&) ((||) x y) (not ((&&) x y))
```

Exercise 12 Define *Inclusive Or* using `not` and `&&`. That is, the defined function should have the same behaviour as the function `||`.

Solution.

```
(|||) :: Bool -> Bool -> Bool
x ||| y  =  not ((not x) && (not y))
```

Exercise 13 Define `&&` using `not` and `||`.

Give two definitions, one using infix notation, the other using prefix notation.

Note: you cannot use the names `&&` or `and` since these are names of (different) predefined functions.

Boolean functions are also called *logic gates*.

The boolean values `True` and `False` are called *constructors*. They can be used on the left hand side of the defining equations of a logic gate.

Defining a logic gate by its truth table

We can define the function `xor` by listing its truth table:

```
xor :: Bool -> Bool -> Bool
xor  True   True   =  False
xor  True   False  =  True
xor  False  True   =  True
xor  False  False  =  False
```

One may also use wildcards:

```
xor :: Bool -> Bool -> Bool
xor  True   False  =  True
xor  False  True   =  True
xor  _      _      = False
```

If wildcards are used, then the *order* of the defining equations matters!

Exercise 14 Use wildcards to give short definitions of the logic gates `&&` and `||`.

Exercise 15 Define the logic gate `nand` which returns `False` exactly when both inputs are `True`. This gate is also known as the *Sheffer stroke* and is often denoted with a vertical bar.

Give two definitions of `nand`: from predefined gates and by listing its truth table.

Exercise 16 Define `&&` using `nand` only. You are not allowed to use the constructors `True` and `False` or any other logic gate.

Hint: Define `not` first.

3.2 Numbers

Basic numeric types

Computing with numbers

Limited precision constant cost	$\longleftrightarrow$	arbitrary precision increasing cost

Haskell offers:

- `Int` - integers as machine words
- `Integer` - arbitrarily large integers
- `Rational` - arbitrarily precise rational numbers
- `Float` - floating point numbers
- `Double` - double precision floating point numbers

Some predefined numeric functions

```
(+), (*), (^), (-), min, max :: Int -> Int -> Int
-    :: Int -> Int          -- unary minus
abs  :: Int -> Int          -- absolute value
div  :: Int -> Int -> Int   -- integer division
mod  :: Int -> Int -> Int   -- remainder of int. div.
gcd  :: Int -> Int -> Int   -- greatest common divisor
```

```
3 ^ 4         ⇒  81
-9 + 4        ⇒  -5              div 33 12  ⇒  2
2-(9 + 4)     ⇒  -11             mod 33 12  ⇒  9
abs -3        ⇒  error           gcd 33 12  ⇒  3
abs (-3)      ⇒  3
```

Exercise 17 (a) Let X be a set containing m elements and let Y be a set containing n elements.

How many functions from X to Y do exists?

For example, if $X = Y =$ `Bool`, then there are 4 such functions: `id` (identity, predefined, reproduces the input), `not` (negation, predefined, swaps True and False), `constTrue` (always returns True), `constFalse` (always returns False).

(b) How many n-ary logic gates do exist?

An n-ary logic gates is a function $f :: \underbrace{\texttt{Bool} \to \ldots \to \texttt{Bool}}_{n} \to \texttt{Bool}$.

(c) Write a Haskell program that computes for a given input n the number of n-ary logic gates.

Solution.

(a) To construct an arbitrary function $f :: X \to Y$ one selects for every $x \in X$ one of the n elements of Y. Hence, there are $\underbrace{n * \ldots * n}_{m} = n^m$ such functions

(b) Mathematically, an n-ary logic gate is the same as function from the set

$$\texttt{Bool}^n := \underbrace{\texttt{Bool} \times \ldots \times \texttt{Bool}}_{n}$$

to the set `Bool`. Since $\texttt{Bool}^n$ has 2^n elements and `Bool` has 2 elements, there are, according to part (a), $2^{(2^n)}$ such functions, that is, there are $2^{(2^n)}$ n-ary logic gates.

(c)

```
gates :: Int -> Int
gates n = 2^(2^n)

{-
*Main> gates 1
4
*Main> gates 2
16
```

```
*Main> gates 3
256
*Main> gates 4
65536
*Main> gates 5
0
-}
```

What went wrong?

Answer: `gates 5 = 2^(2^5) = 2^32 = 65536^2` which is too big for the type `Int`. The problem can be solved by replacing `Int` with `Integer`:

```
gates1 :: Integer -> Integer
gates1 n = 2^(2^n)

{-
*Main> gates1 4
65536
*Main> gates1 5
4294967296
*Main> gates1 10
1797693134862315907729305190789024733617976978942306572734300811577326
7580550096313270847732240753602112011387987139335765878976881441662249
2847430639474124377767893424865485276302219601246094119453082952085005
7688381506823424628814739131105408272371633505106845862982399472459384
79716304835356329624224137216
-}
```

Comparison operators

`(==),(/=),(<=),(<),(>=),(>) :: Int -> Int -> Bool`

But also

`(==),(/=),(<=),(<),(>=),(>) :: a -> a -> Bool`

where `a = Bool, Integer, Rational, Float, Double`

`-9 == 4`	$\Rightarrow$	`False`
`9 == 9`	$\Rightarrow$	`True`
`4 /= 9`	$\Rightarrow$	`True`
`9 >= 9`	$\Rightarrow$	`True`
`9 > 9`	$\Rightarrow$	`False`

Floating point numbers: `Float`, `Double`

- Single and double precision Floating point numbers
- The arithmetic operations `(+),(-),(*),-` may also be used for `Float` and `Double`
- `Float` and `Double` support the same operations

```
(/) :: Float -> Float -> Float
pi  :: Float
exp,log,sqrt,logBase,sin,cos :: Float -> Float
```

`3.4/2`	⇒	`1.7`
`pi`	⇒	`3.14159265358979`
`exp 1`	⇒	`2.71828182845905`
`log (exp 1)`	⇒	`1.0`
`logBase 2 1024`	⇒	`10.0`
`cos pi`	⇒	`-1.0`

Exercise 18 The formula for the roots of a quadratic is

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Define functions `smallerRoot`, `largerRoot` which return the smaller and larger root of a quadratic. The coefficients a, b, c shall be given as floats and the result shall be a float. You may assume that the quadratic does have roots ($b^2 - 4ac \geq 0$) and is not degenerate ($a \neq 0$).

One can switch to double precision floating point numbers by using the type `Double`.

```
Prelude> pi :: Float
3.1415927
Prelude> pi :: Double
3.141592653589793
```

Note that the constant `pi` is overloaded. It can be used as a value of type `Float` and of type `Double`. The overloading is resolved by explicit type annotation.

All constants and functions for floating point numbers are overloaded in this way.

Exercise 19 Define two versions of the function `sqrt`. One should operate on `Float`, the other on `Double`.

Solution.

```
sqrt1 :: Float -> Float
sqrt1 x = sqrt x

sqrt2 :: Double -> Double
sqrt2 x = sqrt x

{-
*Main> sqrt1 2
1.4142135
*Main> sqrt2 2
1.4142135623730951
-}
```

Conversion from and to integers

```
fromIntegral :: Int -> Float
fromIntegral :: Integer -> Float

round :: Float -> Int     -- round to nearest integer
round :: Float -> Integer
```

Example

The following code does not compile:

```
half :: Int -> Float
half x = x / 2
```

The reason is that division (`/`) expects two floating point numbers as arguments, but `x` has type `Int`.

Correct code:

```
half :: Int -> Float
half x = (fromIntegral x) / 2
```

Exercise 20 (a) Define a function that computes the average of three integers, rounded *down* to the nearest integer.

(b) Define a function that computes the average of three integers, rounded to the nearest integer.

Solution.

```
avround :: Int -> Int -> Int -> Int
avround x y z = round (fromIntegral (x+y+z) / 3)

avdown :: Int -> Int -> Int -> Int
avdown x y z = (x+y+z) `div` 3

{-
*Main> avround 3 4 4
4
*Main> avdown 3 4 4
3
-}
```

3.3 Characters and strings

Notation for characters: `'a'`

Notation for strings: `"Hello"`

```
(:)  :: Char -> String -> String   -- prefixing
(++) :: String -> String -> String -- concatenation
```

`(:)` binds stronger than `(++)`

`"Hello " ++ 'W' : "orld!"` $\Rightarrow$ `"Hello World!"`

Is the same as

`"Hello " ++ ('W' : "orld!")`

Exercise 21 (a) Define a function `rep` that repeats a string (that is, concatenates it with itself). Evaluate the expressions `rep "hello! "`, `rep (rep "hello! ")`, and so on.

(b) Define a function `rep2` that applies the function `rep` to a string twice, that is, `rep2 x = rep (rep x)`. How many repetitions of the string `"hello! "` do we get when evaluating the expression `rep2 (rep2 (rep2 "hello! "))` ?

The ASCII code of a character

```
fromEnum :: Char -> Int
```

computes the ASCII code of a character:

`fromEnum 'a'` ⇒ 97
`fromEnum 'b'` ⇒ 98
`fromEnum '9'` ⇒ 57

```
toEnum :: Int -> Char
```

is its inverse. `toEnum 97 :: Char` ⇒`'a'`

The ASCII codes of characters denoting capital letters, small letters and digits are arranged in blocks. For example, the characters `'A'`, ..., `'Z'` have the codes 65, ..., 90.

Exercise 22 Define a function that transforms a capital letter into a small letter.

Solution. Since the codes of letters are arranged in blocks we can simple compute the code of a capital letter shift the code by a certain offset and compute the corresponding character.

```
offset :: Int
offset = fromEnum 'a' - fromEnum 'A'

toLower :: Char -> Char
toLower x = toEnum (fromEnum x + offset)
```

Exercise 23 Define a function that tests whether a character is a digit, that is, one of the characters `'0'`,...,`'9'`.

Exercise 24 Define a function that tests whether an integer is the ASCII code of a lower case vowel. The lower case vowels are the letters a, e, i, o, u.

4 Control and structure

In this chapter we learn how to

- control the execution of a program by various forms of *definition by cases* (if-then-else, guarded equations, case-expressions, pattern matching),
- handle *exceptions*,
- structure programs by *local definitions* (`let`, `where`),
- organise large systems of programs into *modules* and use Haskell's *libraries*,
- make programs more *readable* by introducing, *type synonyms*.

4.1 Definition by cases

If-then-else

```
max2 :: Int -> Int -> Int
max2 x y = if x < y then y else x
```

```
signum :: Int -> Int
signum x = if x > 0  then 1 else
           if x == 0 then 0 else (-1)
```

The general rule for forming an if-then-else expression is:

> If e is an expression of type `Bool` and
>
> e_1, e_2 are expressions of the same type a,
>
> then `if` e `then` e_1 `else` e_2 is an expression of type a.

Note that if-then-else builds an ordinary expression which can be used at any place where the type fits. For example

```
(if 3 < 2 then 4 else 5) * 6
```

is a well-formed expression. What is its value?

Exercise 25 Define a function that computes the absolute value of a floating point number (note that this function is predefined and has name `abs`.

Exercise 26 Define a function that computes the maximum of three integers.

The nesting of the "if-then-else" in the definition of `signum` can be expressed more elegantly using *guarded equations*.

Guarded equations

```
signum1 :: Int -> Int
signum1 x
   | x > 0       = 1
   | x == 0      = 0
   | otherwise   = 1
```

Note that `otherwise` is a constant that has value `True`.

Exercise 27 Define a function that takes as inputs three integers m, n, x, and computes one of the strings `"far left"`, `"left"`, `"middle"`, `"right"`, or `"far right"` depending on whether

$x < m$,

$x = m$,

$m < x < n$,

$x = n$, or

$x > n$.

Case analysis with pattern matching

```
empty :: String -> Bool
empty s = case s of
             ""   -> True
             _    -> False
```

```
tl :: String -> String
tl xs = case xs of
          ""     -> ""
          (_:xs) -> xs
```

Layout. The clauses in a case-expression (the lines with the arrow in the middle) must be *left aligned*. For example, this code does *not* compile

```
tl :: String -> String
tl xs = case xs of
          ""     -> ""
         (_:xs)  -> xs
```

but that one does

```
tl :: String -> String
tl xs = case xs of
          ""    -> ""
          (_:xs) -> xs
```

The Haskell layout rules can be switched off by enclosing the clauses in curly brackets and separating them by semicolons.

```
empty :: String -> Bool
empty s = case s of {""   -> True ; _ -> False}

tl :: String -> String
tl xs = case xs of {"" -> "" ; (_:xs) -> xs}
```

Booleans or Strings as pattern

Case-expressions can be also used with Boolean constants or constant strings as pattern:

```
(&&&) :: Bool -> Bool -> Bool
x &&& y = case x of {False -> False ; True -> y}

age :: String -> Int
age x = case x of
          {"John" -> 21 ; "Alice" -> 20 ; _ -> -1}
```

The functions `(&&&)` and `age` can be equivalently defined by a list of defining equations.

Defining equations with pattern matching

```
(&&&) :: Bool -> Bool -> Bool
False &&& y = False
_     &&& y = y

age :: String -> Int
age "John"   = 21
age "Alice"  = 20
age _        = -1
```

Exercise 28 Define a function `swap12` that swaps the first two characters of a string provided the string has at least length 2. Otherwise reproduce the string unchanged. Define `swap12` in two ways: using a case-expression and using a list of defining equations.

Solution.

```
swap12 :: String -> String
swap12 (x:y:s) = y:x:s
swap12 t       = t

swap12 :: String -> String
swap12 t = case t of {(x:y:s) -> y:x:s ; _ -> t}
```

The general rule for forming case-expressions is as follows: If

- e is an expression type a,
- $p_1, \ldots, p_n$ are patterns of type a,
- $e_1, \ldots, e_n$ are expressions of type b,

then `case` e `of` $\{$ p_1 `->` e_1 `;` `...` `;` p_n `->` e_n $\}$ is an expression of type b.

A *pattern* is an expression built from constructors and different variables.

Constructors are constants such as `True, False, 0, 1,...,""`, and `:`(attaching a character in front of a string. Note that a string like `"hi"` is syntactic sugar for the pattern `'h':'i':""`.

A case-expression is evaluated as follows:

1. the expression e is evaluated,
2. the first pattern p_i matching this value is selected,
3. the expression e_i is evaluated and returned as the value of the case-expression,
4. if no match is possible an error is raised.

Note that a pattern may contain wildcards (`_`) since different occurrences of the wildcard are just syntactic sugar for different variables.

Exercise 29 Define a function that contracts the first two characters of a string into one provided these two characters are equal and the string has at least length 2. Otherwise reproduce the string unchanged.

4.2 Exception handling

A better solution for the function `age` is to raise an exception when the person is not known:

```
age :: String -> Int
age "John"   = 21
age "Alice"  = 20
age x = error ("The age of " ++ x ++ " is not known")
```

The general formation rule for error-expressions is:

- If e is an expression of type `String`,
- then `error` s is an expression of type a for any type a.
- When an error-expression is evaluated, then the computation of the whole expression where it occurs is aborted and the string s is displayed.

Exercise 30 Modify the function in Exercise 28 such that if the string does not have at least length 2, then an exception with an appropriate error message is raised.

4.3 Local definitions

`let` and `where`

```
g :: Float -> Float -> Float
g x y = (x^2 + y^2) / (x^2 + y^2 + 1)
```

better

```
g x y = let s = x^2 + y^2
        in  s / (s + 1)
```

or

```
g x y = s / (s + 1)  where
   s = x^2 + y^2
```

Exercise 31 Recall that in Exercise 18 we computed the roots x of a quadratic equation

$$ax^2 + bx + c = 0$$

from the coefficients a, b, c according to the formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Improve the solution by structuring the code and raising exceptions when no root exists or infinitely many root exist. The function that computes the roots should have an extra Boolean argument that allows one to choose between the two roots.

Solution.

```
roots :: Float -> Float -> Float -> Bool -> Float
roots a b c p

  | a /= 0      = let d = b^2 - 4*a*c
                    in if d >= 0
                       then ((-b) + k * sqrt d) / (2*a)
                       else err

  | a == 0      = if b /= 0 then (-c)/b else err

 where k   = if p then 1 else (-1)
       err = error "no root or infinitely many roots"
```

In order to test whether our function works correctly we define a function that computes $ax^2 + bx + c$ for given a, b, c, x.

```
quad :: Float -> Float -> Float -> Float -> Float
quad a b c x = a*x^2 + b*x + c

{-
*Main> roots 1 2 (-3) True
1.0
*Main> quad 1 2 (-3) 1
0.0
-}
```

In order to make testing more convenient we define a test function.

```
testroots :: Float -> Float -> Float -> Bool
testroots a b c =
   let {x1 = roots a b c True ; x2 = roots a b c False}
   in  quad a b c x1 == 0  &&  quad a b c x2 == 0

{-
*Main> testroots 1 2 (-3)
True
*Main> testroots 1 5 (-3)
False
-}
```

What's wrong?

How can we fix the error?

Exercise 32 Write a program that tests the correctness of the function `sqrt`.

Make sure that the test cannot be applied to negative numbers by an appropriate exception handling. Soften your test so that the expected results are obtained. How "hard" can you make your test?

Local definition of a function

The sum of the areas of two circles with radii `r,s`.

```
totalArea :: Float -> Float -> Float
totalArea r s = pi * r^2 + pi * s^2
```

We make the program more modular and readable by using an auxiliary function to compute the area of one circle:

```
totalArea :: Float -> Float -> Float
totalArea r s = circleArea r + circleArea s

 where  circleArea x = pi * x^2
```

Note that `circleArea` is a function from `Float` to `Float`.

We could add the signature of `circleArea` to make this fact explicit:

```
totalArea :: Float -> Float -> Float
totalArea r s = circleArea r + circleArea s

  where  circleArea :: Float -> Float
         circleArea x = pi * x^2
```

Exercise 33 Define the function `totalArea` using `let`.

Exercise 34 Define a variation of the solution of Exercise 31 where the line

```
then ((-b) + k * sqrt d) / (2*a)
```

is replaced by

```
then ((-b) +- sqrt d) / (2*a)
```

and the function `(+-)` is defined locally.

Solution.

```
roots :: Float -> Float -> Float -> Bool -> Float
roots a b c p

  | a /= 0     = let d = b^2 - 4*a*c

                  in if d >= 0
                     then ((-b) +- sqrt d) / (2*a)
                     else err

  | a == 0     = if b /= 0 then (-c)/b else err

 where

   (+-) = if p then (+) else (-)

   err  = error "no root or infinitely many roots"
```

Exercise 35 [Suggested by a student]

Write a program computing the *standard deviation* of five values.

Note that the standard deviation of $x_1, \ldots, x_5$ is given by the formula

$$\sqrt{\frac{(x_1 - m)^2 + (x_2 - m)^2 + (x_3 - m)^2 + (x_4 - m)^2 + (x_5 - m)^2}{5}}$$

where m is the mean value of $x_1, \ldots, x_5$.

Use local definitions of functions and values to obtain a short and readable program.

Remark. Later we will learn how to compute the standard deviation of an arbitrary *list* of values.

Solution.

```
stdev :: Float -> Float -> Float -> Float -> Float -> Float
stdev x1 x2 x3 x4 x5 = sqrt (mean (d x1) (d x2) (d x3) (d x4) (d x5))

  where

    mean y1 y2 y3 y4 y5 = (y1 + y2 + y3 + y4 + y5)/5

    m = mean x1 x2 x3 x4 x5

    d x = (x-m)^2
```

4.4 Modules

Modules for large program systems

Local definitions are useful for structuring small program units.

In order to structure large program systems one uses *modules*.

- A module must have a name beginning with a capital letter, for example `Geometry`
- The module must reside in a file with the same name, but with extension `.hs`. In our example `Geometry.hs`

A module can look, for example, as follows:

```
module Geometry

-- Exported functions
 (
```

```
  rectangleArea,           -- :: Float -> Float -> Float
  rectangleCircumference,  -- :: Float -> Float -> Float
  circleArea,              -- :: Float -> Float
  circleCircumference      -- :: Float -> Float
 )

 where

-- Definitions of exported functions

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a*b

rectangleCircumference :: Float -> Float -> Float
rectangleCircumference a b = 2*(a+b)

circleArea :: Float -> Float
circleArea r = pi*r^2

circleCircumference :: Float -> Float
circleCircumference r = 2*pi*r
```

- If the list of exported functions is omitted, then all functions defined in the module are exported.
- In order to use a module in a Haskell script one has to import it.

For example

```
import Geometry

-- imported functions
 (
  rectangleArea,
  circleArea
 )

twocircleArea :: Float -> Float -> Float
twocircleArea r1 r2 = circleArea r1 + circleArea r2
```

- If the list of imported functions is omitted, then all functions defined in the module are imported.

Exercise 36

Create a module that contains test functions for Haskell's implementation of the square root function (`sqrt`) and the natural logarithm (`log`). Note that the following equations are expected to hold.

$$\begin{aligned} (\sqrt{x})^2 &= x \quad \text{for } x \geq 0 \\ e^{\log(x)} &= x \quad \text{for } x > 0 \end{aligned}$$

The Haskell function mapping x to e^x is called `exp`. Your test functions should have an extra parameter controlling the "hardness" of the test.

Import your module in another Haskell script and try it out.

On the Haskell web page one finds a large collection of library modules. They can be used in the same way as our example module. For example, by writing

```
import Char
```

we import the Haskell library `Char` with all its functions. These functions are concerned with operations on characters.

It is also possible to import a module directly from the command line.

```
Prelude> isControl '\n'

<interactive>:1:0: Not in scope: `isControl'
Prelude> :module Char
Prelude Char> isControl '\n'
True
Prelude Char> isControl 'a'
False
Prelude Char> :t isControl
isControl :: Char -> Bool
```

4.5 Type synonyms

Programs become more readable if types are given names that indicate what its members are used for.

For example, in the definitions of the areas of a rectangle and a circle the formal parameters of type `Float` sometimes play the role of the side of a rectangle and sometimes the role of the radius of a circle. The results represent areas.

```
type Side   = Float
type Radius = Float
type Area   = Float

rectangleArea :: Side -> Side -> Area
rectangleArea a b = a*b

circleArea :: Radius -> Area
circleArea r = pi*r^2
```

Remark. With the keyword `type` one does not define a new type. One just introduces a new name for an existing type.

Exercise 37

Introduce type synonyms to make the program computing the roots of a quadratic more readable.

5 Input/Output

So far we studied "pure" functional programs which can be evaluated, but which don't interact with the "outside world".

Of course, *some* interaction *does* happen when we run a program:

(1) the expression we type is read from the terminal as a string;

(2) the string is parsed, evaluated and the value transformed back into a string;

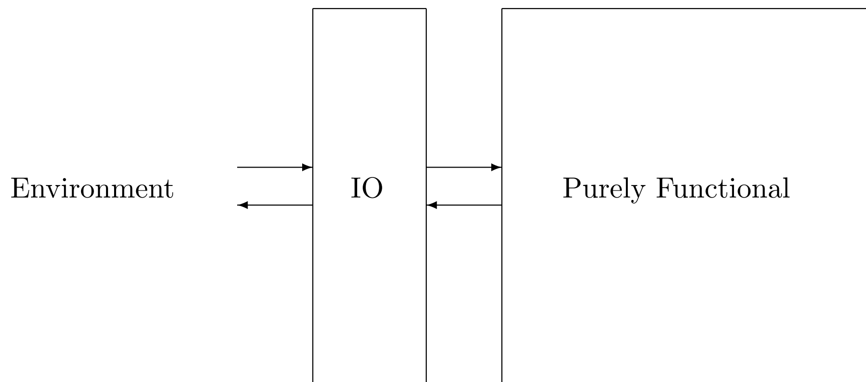
(3) that string is printed at the terminal, and haskell is ready to read the next expression.

This so-called *read-eval-print loop* is carried out by the interactive Haskell program ghci. The actions (1) and (3) are typical examples of *Input/Output*, *IO*, for short.

We now discuss how to write simple IO-programs and how to combine them with purely functional programs in a sensible way.

IO-programs aren't purely functional, but they are necessary to run and use purely functional programs. It is the philosophy of functional programming to strictly separate the purely functional and the IO part of a program. The functional part should do the main computation (and it should be the by far bigger part), whereas the IO-part acts as an interface between the functional part and the environment. The advantage of this separation is that the largest and complicated part of a program can be made highly reliable because it can, for example, be *proven* to be correct. On the other hand, the IO-part should be kept as small and simple as possible because it is there where errors (in the hardware or software, or by the user) are most likely to occur and hardest to detect and eliminate. The separation of the functional and the IO part can be done, in principle, in any programming language, but functional programming languages particularly encourage and support it.

IO-program as interface between environment and purely functional program



5.1 Writing to and reading from the terminal

We begin with simple interaction with the terminal.

Printing a string at the terminal

```
hw :: IO ()
hw = putStrLn "Hello, world!"
```

- The type `IO ()` is the type of *actions*.
- `putStrLn :: String -> IO ()` is a pre-defined Haskell function that takes a string as input and performs the action of printing it at the standard output (usually terminal or console) and inserting a new line.
- Evaluating `hw` will print `Hello, world!` on the terminal.

Carrying out several actions in sequence

```
helloGoodbye :: IO ()
helloGoodbye = do  putStrLn "Hello world!"
                   putStrLn "Goodbye world!"
```

Equivalent notation with curly brackets and semi-colons:

```
helloGoodbye :: IO ()
helloGoodbye =
  do {
       putStrLn "Hello world!" ;
       putStrLn "Goodbye world!"
     }
```

For `do`-expressions the same layout rules apply as for `case`- and `let`-expressions. If the curly-bracket-semi-colon syntax is used no layout rules apply. (Recall that this is the same as for `case`- and `let`-expressions.)

Reading a string from the terminal

```
echo :: IO ()
echo = do {
            putStr "Please enter a string: " ;
            s <- getLine                     ;
            putStrLn s                       ;
            putStrLn s
          }
```

- `getLine :: IO String` reads a line from the standard input (usually the console or terminal) and returns it as a string. A line is usually entered by the user and terminated by pressing the Return-key.
- `IO String` is the type of actions that return a string as result.
- The line `s <- getLine` means that the line read form the terminal is bound to the local variable `s` which then can be used in subsequent clauses of the `do`-expression.

Remarks.

1. The variable `s` is local to the do-expression and therefore not visible outside of it.

2. The line `s <- getLine` is similar to an assignment in imperative languages. However, the difference is that `s` is a *new* local variable. When executing `s <- getLine` it never happens that an existing variable is overwritten.

3. In the program `echo` the two clauses printing `s` at the terminal could be contracted to one that prints the concatenation of `s` with itself plus the newline character `'\n'` in the middle:

```
echo1 :: IO ()
echo1 = do {
             putStr "Please enter a string: " ;
             s <- getLine                     ;
             putStrLn (s ++ "\n" ++ s)
           }
```

4. One can use `getLine` repeatedly:

```
echo2 :: IO ()
echo2 = do {
             putStr "Please enter a string: "   ;
             s <- getLine                       ;
             putStrLn s                         ;
             putStr "and another one, please: " ;
             t <- getLine                       ;
             putStrLn (s ++ " " ++ t)
           }
```

5.2 The type `IO a`

- For any type `a`, the type `IO a` is the type of actions (side-effects) that in addition return a value of type `a`.
- The only way to access the value returned by an action `e :: IO a` is by a clause `x <- e` within a `do`-expression.
- It is *impossible* to define a function `getVal :: IO a -> a` that would "extract" the value returned by an action.
- As a consequence, an expression that has a type not containing any `IO` is purely functional and does not perform any action (thatis, side effect). In particular, its value does not depend on the environment.
- Therefore, the type `IO a` visibly separates the action part of a program from the purely functional part.

Remark. In most (virtually all) other programming languages `IO`-types do not exist. This means, in effect that, for example, the types `IO String` and `String` are identified. If this were the case in Haskell, then, we would have, for example,

```
getLine :: String
```

and we could perform the test

```
getLine == getLine
```

The result would not always be the expected value `True`, but depend on the evaluation strategy and on the strings entered by the user.

This shows that without the distinction between the types `String` and `IO String` even the most basic reasoning about programs (for example the law that everything is equal to itself) breaks down.

The type `IO ()`

- `()` is the type containing exactly one element which is again denoted `()`.
- Hence, the type `IO ()` is the type of actions that do not return any useful value.
- Typically, the `main` program is of type `IO ()` because its only purpose is to do some action.
- The type `()` is often called "unit type". It corresponds to Java's type `void`.

Exercise 38 Program an interactive English-French dictionary.

First Solution.

```
dictionary1 :: IO ()
dictionary1 =
 do en <- getLine
    case en of
      {
        "apple"  -> putStrLn "pomme"  ;
        "bread"  -> putStrLn "pain"   ;
        "butter" -> putStrLn "beurre" ;
        "milk"   -> putStrLn "lait"   ;
        _        -> putStrLn "sorry, don't know"
      }
```

This works, but it is bad programming style since actions (`putStrLn ...  ::  IO ()` are scattered all over the program.

Second Solution.

```
dictionary2 :: IO ()
dictionary2 = do {
                  en <- getLine ;
                  putStrLn (en2fr en)
                 }

en2fr :: String -> String
en2fr en =
    case en of
      {
        "apple"  -> "pomme"   ;
        "bread"  -> "pain"    ;
        "butter" -> "beurre"  ;
        "milk"   -> "lait"    ;
        _        -> "sorry, don't know"
      }
```

This solution is better. It clearly divides the program in a small interactive part (`dictionary2`) and a large purely functional part (`en2fr`).

Note that in order to see that the program `en2fr` is purely functional it suffices to look at its type (`String -> String`) which has no `IO` in it. It is not necessary to check the code (which might be large, in practice).

Exercise 39 Write the program `en2fr` using

(a) pattern matching with several equations,

(b) `if-then-else`,

(c) guarded equations.

The program `dictionary2` can be made slightly more readable using a `let`-clause:

```
dictionary3 :: IO ()
dictionary3 = do {
                  en <- getLine        ;
                  let {fr = en2fr en}  ;
                  putStrLn fr
                 }
```

Note that the `let`-clause has no "`in`".

5.3 Using recursion to create an interaction loop

We can modify our dictionary so that one can enter as many queries as one likes until the empty string is entered:

```
dictloop :: IO ()
dictloop = do {
               en <- getLine  ;
               if en == "" then return ()
                           else do {
                                    putStrLn (en2fr en) ;
                                    dictloop
                                    }
              }
```

- `return ()` is the action that does nothing. Here it is used to terminate the interaction.
- In general, the function `return :: a -> IO a`, where `a` is any type, transforms any value `x` of type `a` into an action that does nothing but returning that value `x`.
- The looping behaviour is achieved by a *recursive call* of `dictloop`.
- In Haskell, *recursion* is the only means for creating loops.

Exercise 40 Write a program that collects nonempty strings entered at the terminal and prints them all if the user enters the empty string.

In an imperative language we would solve this by writing a loop with a variable that holds the words entered so far and which is updated by the new word entered in each round. At the end the content of the local variable would be returned as result.

Here is how we do this in a functional language?

```
collect :: IO ()
collect = do {
               allwords <- coll "" ;
               putStrLn allwords
             }
 where
```

```
coll :: String -> IO String
coll history = do {
                    s <- getLine            ;
                    case s of
                      ""  -> return history
                      _   -> coll (history ++ " " ++ s)
                  }
```

We see that the local variable is modelled by the local *function* `coll` which is recursively called with an “updated” argument.

```
*Main> collect
when
is
this
lecture
over?

 when is this lecture over?
*Main>
```

5.4 Input/Output with numbers

Suppose we want to compute the solutions of a quadratic whose coefficients are entered through the terminal. The solutions shall be displayed at the terminal.

Because at the terminal we can only enter strings, we need a function that transforms strings into numbers:

```
read ::  String -> Float
```

Furthermore, in order to display the computed roots on the terminal, we need a function that translates numbers into strings:

```
show ::  Float -> String
```

Recall that we did use the function `show` earlier.

The functions `read` and `show` can be used for other number types and types such as `Bool` or `Char` as well.

Solving a quadratic interactively

```
solve :: IO ()
solve = do {
            sa <- getLine ;
            sb <- getLine ;
            sc <- getLine ;
            let {
                 a   = read sa ;
                 b   = read sb ;
                 c   = read sc ;
                 x1  = roots a b c True ;
                 x2  = roots a b c False ;
                 sc1 = show x1 ;
                 sc2 = show x2
                } ;
           putStrLn ("The solutions are " ++ sc1 ++ " and " ++ sc2)
          }

roots :: Float -> Float -> Float -> Bool -> Float
roots a b c p

  | a /= 0     = let { d = b^2 - 4*a*c }
                 in if d >= 0
                    then ((-b) + k * sqrt d) / (2*a)
                    else err

  | a == 0     = if b /= 0 then (-c)/b else err

 where k   = if p then 1 else (-1)
       err = error "no root or infinitely many roots"

*Main> solve
2
5
-7
The solutions are 1.0 and -3.5
*Main> solve
-5
8
20
The solutions are -1.3540659 and 2.9540658
```

Note that the computationally interesting part is completely covered by the purely functional program `roots`. The main program `solve` only handles the interaction with the terminal.

Exercise 41 Write a program that computes the length of a string entered by the user.

You may use the predefined function

```
length :: String -> Int
```

Exercise 42 Modify the program of exercise 41 such that the user can compute lengths of strings repeatedly (see program `echoLoop`).

Exercise 43 Write a program that computes the sum of numbers entered by the user (see program `collect`).

Exercise 44 Modify the program of exercise 43 such that the user can compute sums repeatedly.

Exercise 45 Modify the program `solve` such that the user is asked whether the solutions should be checked and, if that's the case, the user may specify the required accuracy of the solutions.

5.5 Communication with files

Reading from and writing to files can be done with the operations

- Reading a file:

  ```
  readFile     ::  FilePath -> IO String
  ```

- Writing into a file (override, append):

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

where `FilePath` is a predefined type synonym:

```
type FilePath =  String
```

Example: Displaying the length of a file

```
lengthFile :: IO ()
lengthFile =
  do {
      fn <- getLine ;
      file <- readFile fn ;
      putStrLn ("File " ++ fn ++ " contains " ++
                show(length file) ++ " characters.\n")
     }
```

```
*Main> lengthFile
fp1-io.tex
File fp1-io.tex contains 23723 characters.
```

Example: Reversing the content of a file and writing it into another one

```
reverseFile :: FilePath -> FilePath -> IO ()
reverseFile fn1 fn2 =
  do {
      str <- readFile fn1 ;
      writeFile fn2 (reverse str)
     }
```

Note that the above program expects the filenames as arguments. Hence a valid call of `reverseFile` is, for example

```
reverseFile "28-2.hs" "test.hs"
```

Notice that the filenames have to enclosed in double quotes because we use ghci's predefined IO.

Exercise 46 Write a program that writes for a given integer n all numbers $1 \ldots n$ into a specified file.

Exercise 47 Write a program that checks whether two files have the same content.

5.6 Random data

The Haskell module `Random`, which is loaded by including the line

```
import Random
```

at the beginning of a Haskell script, has several operations for generating random numbers. For example,

```
randomRIO :: (Random a) => (a,a) -> IO a
```

The type of `randomRIO` says that for any type in the type class `Random` (for example, `Int`, `Char`) and any two values `x`, `y` of that type the expression

```
randomRIO (x,y)
```

evaluates to an action returning an element in the range between `x` and `y`.

Example: Generating a random password.

```
password :: IO ()
password = do {
               let {c = randomRIO ('a','z')} ;
               c1 <- c ;
               c2 <- c ;
               c3 <- c ;
               c4 <- c ;
               putStrLn [c1,c2,c3,c4]
              }
```

Exercise 48 Improve the password program such that the length of the password can be determined by the user.

Exercise 49 Using the solution to Exercise 48, write a program that generates random passwords of random lengths between 1 and 10.

Exercise 50 Write a program that allows the user to enter names (as a strings) and that assigns to each name a random password (say of length 8) and writes names and passwords into a file.

Haskell has many more libraries (that is, modules) for other kinds of IO operations, for example, GUIs, communication with the operating system, and interfaces with other programming languages. Click the link *Haskell libraries* for more information.

6 Tuples and lists

In this chapter we introduce tuples and lists. The main difference between tuples and lists is that a type of tuples contains tuples of fixed length whose components may have different types, while a type of lists contains lists of different lengths whose components all have the same type.

6.1 Tuples

Pairs

- If `a` and `b` are types, then `(a,b)` is a type.
- The elements of `(a,b)` are pairs `(x,y)` where `x :: a` and `y :: b`.
- Examples:

```
(3+4,7)                       ::  (Int,Int)
(342562,("George","Sand"))    ::  (Int,(String,String))
```

In mathematics the type (a, b) is usually denoted $a \times b$ and called the *cartesian product* of a and b.

Pattern matching: first and second component of a pair

The predefined polymorphic functions computing the first and second component of a pair are defined by pattern matching:

```
fst :: (a,b) -> a
fst (x,y) = x

snd :: (a,b) -> b
snd (x,y) = y
```

Exercise 51 Define a polymorphic function `swap` that swaps the components of a pair. Give two definitions, one using pattern matching, the other using the functions `fst` and `snd`.

Solution

```
swap1 :: (a,b) -> (b,a)
swap1 (x,y) = (y,x)

swap2 :: (a,b) -> (b,a)
swap2 p = (snd p,fst p)
```

Exercise 52 What's wrong with the following definition?

```
diag :: a -> (a,b)
diag x = (x,x)
```

Correct the error.

Exercise 53 Suppose we represent a point in the two-dimensional plane by a pair of floats

```
type Point = (Float,Float)
```

Define the following operation:

(a) The distance of two points.

(b) Reflection of a point at the

- x-axis,
- y-axis,
- diagonal,
- origin.

(c) scaling a point by a factor.

(d) translating a point by another point (viewed as a vector), that is, adding two points component-wise.

(e) rotating a point around the origin by a given angle.

Tuples

- Generalising pairs, one can form the type of tuples of a fixed length n.

  For example triples `(a,b,c)`.

- One can define functions by pattern matching on tuples:

```
first :: (a,b,c) -> a
first (x,y,z) = x

second :: (a,b,c) -> b
second (x,y,z) = y

third :: (a,b,c) -> c
third (x,y,z) = z
```

Exercise 54 Suppose we represent time by a quadruple:

```
type Time   = (Day,Hour,Minute,Second)
type Day    = Int      -- >= 0
type Hour   = Int      -- between 0 and 23
type Minute = Int      -- between 0 and 59
type Second = Float    -- between 0 and 59
```

Define a function that adds two times.

6.2 Lists

- Haskell has a predefined data type of *polymorphic lists*, `[a]`.
- Lists are generated by the empty list and the operation of adding an element to a list.

```
[]    ::  [a]                -- the empty list
(:)   ::  a -> [a] -> [a]    -- adding an element
```

- Hence, the elements of `[a]` are either `[]`, or of the form `x:xs` where `x` is of type `a` and `xs` is of type `[a]`.
- Special notation for lists:

```
[1,2,3]  =  1 :  2 :  3 :  []
         =  1 :  (2 :  (3 :  []))
```

- The type of elements of a list can be arbitrary, but all elements must have the same type.

Some lists and their types

```
[1,3,5+4]                        :: [Int]

['a','b','c','d']                :: [Char]

[ [ True , 3<2 ] , [] ]          :: [[Bool]]

[(198845,"Cox"),(203187,"Wu")] :: [(Int,String)]
```

Strings are lists of characters

- The type `String` is the same as the type `[Char]` of lists of characters.
- Haskell has a special way of displaying strings:

```
['H','e','l','l','o']  ⇒  "Hello"
"Hello"                ⇒  "Hello"
[0,1,2,3]              ⇒  [0,1,2,3]
['0','1','2','3']      ⇒  "0123"
```

Some predefined polymorphic functions on lists

```
(:)    :: a -> [a] -> [a]   -- adding an element
(++)   :: [a] -> [a] -> [a] -- concatenating two lists
null   :: [a] -> Bool       -- emptyness test
length :: [a] -> Int
head   :: [a] -> a
tail   :: [a] -> [a]
(!!)   :: [a] -> Int -> a  -- accessing elements by index
```

```
3 : [4,5]      ⇒  [3,4,5]
[3] ++ [4,5]   ⇒  [3,4,5]
head [3,4,5]   ⇒  3
tail [3,4,5]   ⇒  [4,5]
[3,4,5] !! 0   ⇒  3
[3,4,5] !! 2   ⇒  5
```

```
concat

concat :: [[a]] -> [a]


 concat [[3,4],[5],[6,7]]        ⇒  [3,4,5,6,7]
 concat ["How ","are ","you?"]   ⇒  "How are you?"
```

```
zip

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

 zip [1,2,3] ["A","B","C"]  ⇒  [(1,"A"),(2,"B"),(3,"C")]
 zip [1,2,3] ["A","B"]      ⇒  [(1,"A"),(2,"B")]
```

Pattern matching with lists

Some of the predefined functions are defined by simple pattern matching on lists.

```
null :: [a] -> Bool
null []  = True
null _   = False -- applies if first line does not match

head :: [a] -> a
head []     = error "head of empty list"
head (x:_) = x                     -- brackets required!

tail :: [a] -> [a]
tail []      = error "tail of empty list"
tail (_:xs) = xs
```

Nested patterns for lists

We can specify lists of certain lengths by pattern matching, without using the predefined function `length`.

```
exactlytwo :: [a] -> Bool
exactlytwo [_,_] = True
exactlytwo _    = False
```

The first equation is equivalent to

```
exactlytwo (_:_:[]) = True

atleasttwo :: [a] -> Bool
atleasttwo (_:_:_) = True
atleasttwo _       = False
```

Equidistant numbers

Haskell has a special syntax for lists of equidistant numbers:

```
[1..5]         ⇒  [1,2,3,4,5]
[1,3..11]      ⇒  [1,3,5,7,9,11]
[10,9..5]      ⇒  [10,9,8,7,6,5]
[1,3..11]      ⇒  [1,3,5,7,9,11]
[3.5,3.6..4]   ⇒  [3.5,3.6,3.7,3.8,3.9,4.0]
```

The `..` notation can be used for any type in the class `Enum`:

```
['a'..'z']  ⇒  "abcdefghijklmnopqrstuvwxyz"
```

Defining functions using the .. notation

```
interval :: Int -> Int -> [Int]
interval n m = [n..m]

evens :: Int -> [Int]
evens n = [0,2..2*n]
```

```
interval 3 7  ⇒  [3,4,5,6,7]
evens 5       ⇒  [0,2,4,6,8,10]
```

List comprehension

- *Select* all elements of a given list
- which pass a given *test*
- *transform* them in to a result
- and collect the results in a list.

Example: Squaring all odd numbers in a list

`[x * x | x <- [1..10], odd x ]` ⇒ [1,9,25,49,81]

- `x <- [1..10]` is a *generator*
- `odd x` is a *test*
- `x * x` is the *transformation*

Note that

- generators and tests may be repeated or omitted,
- later generators and tests may depend on earlier generators,
- the transformation may be the identity (just `x`).

Example: Cartesian product

All pairings between elements of two lists:

```
[(x,y) | x <- [1,2,3], y <- [4,5]]
⇒ [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]

[(x,y) | y <- [4,5], x <- [1,2,3]]
⇒ [(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Example: Pythagorean triples

Find all triples (x, y, z) of positive integers $\leq n$ such that

$$x^2 + y^2 = z^2$$

```
pyth :: Integer -> [(Integer,Integer,Integer)]
pyth n = [(x,y,z) | x <- [1..n],
                    y <- [1..n],
                    z <- [1..n],
                    x^2 + y^2 == z^2]
```

```
pyth 10  ⇒  [(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
```

Exercise 55 Improve the efficiency of the function `pyth` by avoiding essentially repeated solutions by generating only $x < y < z$ and requiring x and y to have no common factor (using the function `gcd`).

Generators with pattern matching

Adding the components of pairs:

```
addPairs :: [(Int,Int)] -> [Int]
addPairs ps = [ x + y | (x, y) <- ps ]
```

```
 addPairs (zip [1,2,3] [10,20,30])  ⇒  [11,22,33]
```

Collecting all singletons in a list of lists

```
singletons :: [[a]] -> [[a]]
singletons xss = [[x] | [x] <- xss]
```

```
 singletons ["take","a","break","!"]  ⇒  ["a","!"]
```

Exercise 56 Use list comprehension to define a function that counts how often a character occurs in a string (you may use all predefined functions discussed so far).

Hint: `(==)` can be used for characters.

Exercise 57 Use list comprehension and the predefined function

```
sum :: [Int] -> Int  -- the sum of a list of integers
```

to define a function that, for two lists of integers of the same length, $[x_0, \ldots, x_{n-1}]$ and $[y_0, \ldots, y_{n-1}]$, computes

$$x_0 * y_0 + \ldots + x_{n-1} * y_{n-1}$$

Hint: Use `zip`.

7 Higher-order functions

A distinguished feature of functional programming is the fact that it gives functions the status of ordinary data which can be used in the same way as any other data. In particular, functions can be input to other functions.

Higher-order functions

- A *higher-order function* is a function that accepts functions as input.
- As an example, consider the function that computes the value of a function f at 0:

```
eval0 ::  (Float -> Float) -> Float
eval0 f = f 0
```

- The type `Float -> Float` is called a *function type.*
- In functional programming function types are "first-class citizens", that is, they can be used in the same way as other types.

```
*Main> eval0 sin
0.0
*Main> eval0 cos
1.0
*Main> eval0 exp
1.0
*Main> eval0 sqrt
0.0
*Main> let f x = (x+3)^2 + 5
*Main> eval0 f
14.0
```

Exercise 58 Why does the following higher-order function, testing whether two functions on the integers are equal, not work?

```
testEq :: (Integer -> Integer) -> (Integer -> Integer) -> Bool
testEq f g  = f == g
```

Solution

Two functions are considered equal if they return the same result for all inputs. Since there are infinitely many integers this would require an infinite amount of computation time. For this reason there exists no equality test on function types such as `Integer -> Integer`.

7.1 Functions as values

`(+) :: Float -> Float -> Float` is a function that takes two integers as inputs and returns an integer as output.

For example, `(+) 3 4` (this is the same as `3 + 4`) evaluates to `7`.

The type `Float -> Float -> Float` is in fact shorthand for
the type `Float -> (Float -> Float)`

Therefore, `(+) :: Float -> (Float -> Float)` is a function that takes *one* float as input and returns a function from `Float` to `Float` as output.

For example, `(+) 3 :: Float -> Float`. Which function is it?

`(+) 3` is the function that, when applied to any number n, computes the number
`((+) 3)` n `= 3 +` n.

The expression `(+) 3 4` is shorthand for `((+) 3) 4`.

Sections

- Expressions such as `(+) 5 :: Float -> Float`, or `(/) 5 :: Float -> Float` are called *sections*.

  One has `(/) 5 x = 5/x`.
- Instead of `(/) 5` one may write `(5 /)`.
- One may also form the section `(/ 3) :: Float -> Float`. This is the function that divides every number by 3, that is `(/ 3) x = x/3`.

7.2 Bracketing rules for function types and application

- Function types are *right associative:*

  `a -> b -> c` is shorthand for
  `a -> (b -> c)`

- Function application is *left associative:*

  `f x y` is shorthand for
  `(f x) y`.

Note that this applies more generally to chains of arrows and function applications. For example

`a -> b -> c -> d`	is shorthand for	`a -> (b -> (c -> d))`
`f x y z`	is shorthand for	`((f x) y) z`

Exercise 59 What is the value of the expression `eval0 ((+) 5)` ?

Solution

```
eval0 ((+) 5) = ((+) 5) 0 = 5 + 0 = 5
```

Exercise 60 What happens if in Exercise 59 we omit the outer brackets and evaluate the expression `eval0 (+) 5` ?

Solution

Since `eval0 (+) 5` is shorthand for `(eval0 (+)) 5` this would result in a type error: We have

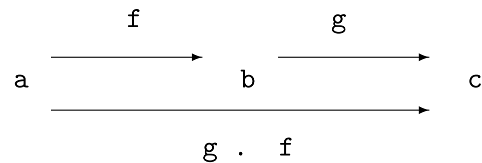
```
eval0 :: (Float -> Float) -> Float
```

therefore, in order for `(eval0 (+))` to be type correct we would need `(+) :: Float -> Float`. However, we have `(+) :: Float -> Float -> Float`.

Even if we replace in the expression `(eval0 (+)) 5` the function `(+)` by a function of the correct type, say `sqrt :: Float -> Float`, the expression `(eval0 sqrt) 5` is not type correct because the type of `(eval0 sqrt)` is `Float` which is not a function type, so it doesn't make sense to apply `(eval0 sqrt)` to the argument `5`.

7.3 Composition

An important higher-order function is composition




```
(.) :: (b -> c) -> (a -> b) -> (a -> c)

(g . f) x = g (f x)
```

Or, equivalently

```
(.) g f x = g (f x)
```

Exercise 61 Consider the functions

```
square, times2, pol :: Float -> Float
square x = x^2
times2 x = x*2
pol    x = x^2 + 2*x  + 1
```

(a) Does `square . times2 = times2 . square` hold?

(b) Find a function `f :: Float -> Float` such that `pol = square . f` holds.

7.4 Higher-order functions on lists

We now discuss some useful predefined higher-order functions on lists.

map

`map` applies a function to all elements of a given list and returns the list of results. Informally:

$$\texttt{map}\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$$

`map` can be defined using list comprehension:

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]

*Main> map null [[1,2],[],[1,2,3]]
[False,True,False]
*Main> map sin [0,pi/2,pi]
[0.0,1.0,1.2246063538223773e-16]
```

Exercise 62 Define a function that computes the square roots of the absolute values of a list of integer, that is $[x_1, \ldots, x_n]$ should be mapped to $[\sqrt{|x_1|}, \ldots, \sqrt{|x_n|}]$.

Solution

1. Using list comprehension:

```
sqrtabs1 :: [Float] -> [Float]
sqrtabs1 xs = [sqrt(abs x) | x <- xs]
```

2. Using `map`:

```
sqrtabs2 :: [Float] -> [Float]
sqrtabs2 xs = map sqrt (map abs xs)
```

3. Using `map` and composition:

```
sqrtabs3 :: [Float] -> [Float]
sqrtabs3 = map sqrt . map abs
```

4. Another solution using `map` and composition:

```
sqrtabs4 :: [Float] -> [Float]
sqrtabs4 = map (sqrt . abs)
```

Exercise 63 In Exercise 35 we wrote a program computing the standard deviation of five values

$$\sqrt{\frac{(x_1 - m)^2 + (x_2 - m)^2 + (x_3 - m)^2 + (x_4 - m)^2 + (x_5 - m)^2}{5}}$$

where m is the mean value of $x_1, \ldots, x_5$.

Modify this program so that it computes the standard deviation of an arbitrary nonempty list of values.

Solution

```
stdev :: [Float] -> Float
stdev values = sqrt(sum (map f values) / n)

  where

    n = fromIntegral (length values)
    m = sum values / n

    f :: Float -> Float
    f x = (x-m)^2
```

filter

`filter` selects from a given list those elements that pass a given test.

`filter` can be defined by list comprehension:

```
filter :: (a -> Bool) -> [a] -> [a]
filter test xs = [x | x <- xs, test x]

*Main> filter null [[1,2],[],[1,2,3]]
[[]]
*Main> filter odd [1..10]
[1,3,5,7,9]
```

Exercise 64 Define a function that computes the inverses of the non-zero elements of a list.

Solution

1. Using list comprehension:

```
invs1 :: [Float] -> [Float]
invs1 xs = [1/x | x <- xs, x /= 0]
```

2. Using list `map`, `filter` and sections:

```
invs2 :: [Float] -> [Float]
invs2 xs = map (1/) (filter (/=0) xs)
```

3. Using list `map`, `filter`, sections and composition:

```
invs3 :: [Float] -> [Float]
invs3 = map (1/) . filter (/=0)
```

Exercise 65 Define a function that computes the square roots of the non-negative elements of a list. Give three solutions following the pattern of Exercise 64.

take, drop, takeWhile and dropWhile

- The functions

  ```
  take, drop :: Int -> [a] -> [a]
  ```

  take respectively *drop* a *specified number* of elements from a list

- There are related higher-order functions

  ```
  takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
  ```

  that take respectively drop elements from a list *as long as a given property holds*.

```
Prelude> take 5 [1..10]
[1,2,3,4,5]
Prelude> drop 5 [1..10]
[6,7,8,9,10]
Prelude> takeWhile even [2,4,6,7,8,10]
[2,4,6]
Prelude> dropWhile even [2,4,6,7,8,10]
[7,8,10]
```

zipWith

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

takes a binary operation, applies it pairwise to the elements of two given lists, and returns the list of results.

```
*Main> zipWith (<) [1,2,3] [3,2,1]
[True,False,False]
*Main> zipWith (*) [1,2,3] [3,2,1]
[3,4,3]
```

Exercise 66 Define `zipWith` from `zip` using list comprehension.

Exercise 67 Define `zip` from `zipWith`.

7.5 Lambda-abstraction

- *Lambda-abstraction* (also written λ-*abstraction*) is a means to define a function without giving it a name (anonymous function).
- If `e` is an expression containing a free variable `x`, then the lambda-abstraction `\x -> e` is an expression denoting the function mapping `x` to `e`
- If `x` has type `a` and `e` has type b, then `\x -> e` has type `a -> b`
- For example, `\x -> x ++ x` has type `String -> String`.

  Applying `\x -> x ++ x` to the argument `"hello "` yields

  `(\x -> x ++ x) "hello " = "hello " ++ "hello " = "hello hello "`

For example, `\x -> sin(pi*x)` has type `Float -> Float`.

Applying `\x -> sin(pi*x)` to the argument `0.5` yields

`(\x -> sin(pi*x)) 0.5 = sin(pi*0.5) = 1`

7.6 Equality between functions: Extensionality

When are two functions equal?

- Two functions $f, g :: a \to b$ are called *extensionally equal*, written

  $$f = g$$

  if they produce the same results for all possible arguments:

  $$\text{for all } x :: a \ (f\ x = g\ x)$$

- A functional program does not change its input/output behaviour if one replaces in it a pure function (that is, a function not involving `IO`) by another pure function which is extensionally equal (although possibly differently defined)
- This is called the principle of *extensionality* (also known as *referential transparency*).
- Extensionality is essential for proving the correctness of programs or program optimisations.
- In non-functional programming languages extensionality does not hold.

Remark. In the following we mean by "equal" always "extensionally equal" as far as functions are concerned.

Exercise 68 Define function `eq1`, `eq2`, `eq3`, `eq4`, `eq5` such that `eq`n tests whether two n-ary Boolean functions are equal.

Solution

If we solved this exercise naively we would have to write a lot of code. For example, to define `eq56` we would have to write down 32 equations because there are 32 ($= 2^5$) different 6-tuples of Boolean values. However, if we define `eq`n using `eq`$(n-1)$ using a higher order function "lifting" equalities the programs become short:

```
type Bool1 = Bool -> Bool
type Bool2 = Bool -> Bool1  -- = Bool -> Bool -> Bool
type Bool3 = Bool -> Bool2  -- etc.
type Bool4 = Bool -> Bool3
type Bool5 = Bool -> Bool4

type Eqtype a = a -> a -> Bool

lifteq :: Eqtype a -> Eqtype (Bool -> a)
lifteq eq f g = eq (f True) (g True) && eq (f False) (g False)
```

To better understand this definition note that `Eqtype a -> Eqtype (Bool -> a)` is the same as `Eqtype a -> (Bool -> a) -> (Bool -> a) -> Bool`.

```
eq1 :: Eqtype Bool1
eq1 =  lifteq (==)

eq2 :: Eqtype Bool2
eq2 =  lifteq eq1

eq3 :: Eqtype Bool3
eq3 =  lifteq eq2

eq4 :: Eqtype Bool4
eq4 =  lifteq eq3

eq5 :: Eqtype Bool5
eq5 =  lifteq eq4
```

Exercise 69 Define a function that test whether for a given integer $n \geq 0$ and functions f, g `::` `[Bool] -> Bool`, f and g coincide for all boolean lists of length n.

Solution. We first compute a list of all boolean lists of lists of length n:

```
blists :: Int -> [[Bool]]
blists 0     = [[]]
blists (n+1) = [True:bs | bs <- bss ] ++ [False:bs | bs <- bss]
  where bss = blists n
```

Now we simply compare the lists of values of f and g on `blist` n.

```
eqfun :: Int -> Eqtype ([Bool] ->  Bool)
eqfun n f g = map f bss == map g bss
  where bss = blists n
```

Note that the function `blists` has two features we didn't come across far:

- `blists` uses pattern matching on 0 and $n + 1$.
- `blists` is recursive, since in the definition of `blists (n+1)` it call itself. Recursively defined functions will be the topic of the next chapter.

8 Recursion and induction

In Chapter 5 we used recursion to implement repeated actions. Now we study how to define *functions* recursively and how to prove their correctness using the principle of *induction*. For recursively defined functions correctness proofs are particularly important because, when programmed carelessly, recursive function may fail to terminate or have unacceptably large run time, and in many cases it is not obvious that they compute the correct result.

8.1 Recursion and induction on natural numbers

Recursion

- *Recursion* = defining a function in terms of itself.

```
fact :: Integer -> Integer
fact n = if n == 0
         then 1                -- base
         else n * fact (n - 1) -- recursive call
```

  Does not terminate if `n` is negative. Therefore

```
fact :: Integer -> Integer
fact n
   | n < 0   = error "negative argument to fact"
   | n == 0  = 1
   | n > 0   = n * fact (n - 1)
```

The correctness of the function `fact`, that is, the fact that it correctly computes the factorial function, can be proved *induction*.

We set $\mathbf{N} := \{0, 1, 2, 3, \ldots\}$ = the set of natural numbers.

Induction

To prove that $P(n)$ is true for all $n \in \mathbf{N}$ it suffices to prove the following:

- *Induction Base*: $P(0)$.
- *Induction Step*: $P(n)$ implies $P(n + 1)$, for all $n \in \mathbf{N}$.

*R*emarks.

1. To prove the induction step on assumes the $P(n)$ holds and tries to show $P(n+1)$. The assumption $P(n)$ is often called "induction hypothesis" (abbreviated "ih").

2. If the property $P(n)$ is clear from the context one often indicates the induction base by writing "$n = 0$" and the induction step by writing "$n \to n+1$".

3. If one chooses a fixed natural number m and one only wants to show that $P(n)$ holds for all number $n \geq m$, then one has to show $P(m)$ in the base case of the inductive proof.

Correctness proof for `fact`

We prove by induction that `fact` $n = n!$ for every $n \in \mathbf{N}$.

Recall that $n! = 1 * 2 * \ldots * n$.

Therefore: $0! = 1, \quad (n+1)! = n! * (n+1)$.

Induction base ($n = 0$): `fact` $0 = 1 = 0!$.

Induction step ($n \to n+1$):

The **induction hypothesis** is `fact` $n = n!$.

We have to show `fact` $(n+1) = (n+1)!$:

$$\begin{aligned} \texttt{fact}\ (n+1) &\overset{\text{def. }\texttt{fact}}{=} (n+1) * \texttt{fact}\ n \\ &\overset{\text{ih}}{=} (n+1) * n! \\ &= (n+1)! \end{aligned}$$

Remark. The correctness proof also shows that the function `fact` terminates because `fact` $n = n!$ and $n!$ is a defined number. A fundamental result about the semantics of Haskell states that whenever e is a purely functional Haskell expression (no `IO`) such that we can prove that $e = n$ holds, where n is a number, then the evaluation of e will terminate with the result n. This is known as the "Computational Adequacy Theorem". In this theorem n can be any kind of numbers, but also a Boolean, string, or character.

Computing recursive functions

How does Haskell compute a recursive function?

Answer: roughly in the same way you would do with pencil and paper, namely by *reduction*.

```
fact 3  ⇒  3 * fact 2  ⇒  3 * 2 * fact 1
  ⇒  3 * 2 * 1 * fact 0  ⇒  3 * 2 * 1 * 1  ⇒  6
```

Exercise 70 Define the factorial function using the predefined function

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

that computes the product of a list of numbers.

Fibonacci numbers: 1,1,2,3,5,8,13,21,. . .

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

Due to *two recursive calls* this program has *exponential run time*.

In order to prove that `fib n` terminates and has exponential run time one needs a more general form of induction.

Complete induction

To prove that $P(n)$ is true for all natural numbers it suffices to prove the following for all natural numbers n:

- The assumption that $P(k)$ holds for all $k < n$ implies $P(n)$.

In other words: to prove $P(n)$ we may assume that $P(k)$ holds for all $k < n$.

The assumption that $P(k)$ holds for all $k < n$ is called *strong induction hypothesis*.

Remark. Complete induction is sometimes also called “wellfounded induction” or “<-induction”.

Proof that `fib` n terminates

Let n be a natural number. We show that `fib` n terminates by strong induction.

The strong induction hypothesis is that `fib` k terminates for all $k < n$.

Case $n = 0$ **or** $n = 1$**:** `fib` 0 and `fib` 1 obviously terminate (in one step). Note that the strong i.h. has not been used.

Case $n > 1$**:** Then `fib` $n =$ `fib`$(n-1) +$ `fib`$(n-2)$.

By the strong i.h. `fib`$(n-1)$ and `fib`$(n-2)$ terminate. Therefore, `fib` n terminates.

In order to show that `fib` n has exponential run time `run` n we first note that `run` $0 =$ `run` $1 = 1$ and for $n > 1$

$$\texttt{run}\ n = \texttt{run}(n-1) + \texttt{run}(n-2) + 1$$

assuming that addition of natural numbers is performed in one step.

It suffices to show

(a) `fib` $n \leq$ `run` n

(b) `run` $n \leq 2^n$

(c) $(\frac{3}{2})^{n-1} \leq$ `fib` n

Exercise 71 Prove statement (a) by complete induction.

Exercise 72 Prove statement (b) by complete induction.

Exercise 73 Prove statement (c) by complete induction.

Solution

Cases $n = 0$ *and* $n = 1$*:* $(\frac{3}{2})^{-1} < 1 = \texttt{fib}\ 0$, $(\frac{3}{2})^{0} = 1 = \texttt{fib}\ 1$.

Case $n > 1$*:* By the strong i.h. $(\frac{3}{2})^{n-2} \leq \texttt{fib}(n-1)$ and $(\frac{3}{2})^{n-3} \leq \texttt{fib}(n-2)$. Therefore

$$\begin{array}{rcll} \texttt{fib}\ n & = & \texttt{fib}(n-1) + \texttt{fib}(n-2) & \\ & \geq & (\dfrac{3}{2})^{n-2} + (\dfrac{3}{2})^{n-3} & \text{(by the strong i.h.)} \\ & = & (\dfrac{3}{2} + 1) * (\dfrac{3}{2})^{n-3} & \\ & \geq & (\dfrac{3}{2})^{2} * (\dfrac{3}{2})^{n-3} & \text{(because } \frac{3}{2} + 1 \geq (\frac{3}{2})^2\text{)} \\ & = & (\dfrac{3}{2})^{n-1} & \end{array}$$

Fibonacci numbers improved

- A linear Fibonacci program with a subroutine computing pairs of Fibonacci numbers:

```
fib1 :: Integer -> Integer
fib1 n = fst (fibpair n)  where

  fibpair :: Integer -> (Integer,Integer)
  fibpair n
   | n < 0    = error "negative argument to fib"
   | n == 0   = (1,1)
   | n > 0    = let (k,l) = fibpair (n - 1)
                in  (l,k+l)
```

Exercise 74 Prove, using (ordinary) induction, that the `fibpair`n has run time $\leq 2 * n$. You may assume that addition and forming a pair take one time step each.

The faster function `fib1` is supposed to compute the Fibonacci numbers as well, that is, we want the equation

$$\texttt{fib1}\, n = \texttt{fib}\, n$$

to hold for all natural numbers n.

Our ability to confirm this empirically (by testing) is very limited since we cannot compute `fib`$\,n$ for $n > 40$.

Exercise 75 Prove that $\texttt{fib1}\,n = \texttt{fib}\,n$ holds for all $n \in \mathbf{N}$.

Solution

The equation $\texttt{fib1}\,n = \texttt{fib}\,n$ cannot be proven directly by induction. But, obviously, it suffices to prove instead

$$P(n) \quad :\equiv \quad \texttt{fibpair}\,n = (\texttt{fib}\,n\,,\,\texttt{fib}\,(n+1))$$

$P(n)$ can be proven by induction.

- *Base, P(0)*: $\texttt{fibpair}\,0 = (1,1) = (\texttt{fib}\,0\,,\,\texttt{fib}\,1)$
- *Step*: Assume, as induction hypothesis (i.h.) that $P(n)$ holds. We show $P(n+1)$:

$$\begin{array}{rcl} \texttt{fibpair}\,(n+1) & = & (\texttt{fib}\,(n+1)\,,\,\texttt{fib}\,n + \texttt{fib}\,(n+1)) \quad \text{(def. }\texttt{fibpair}\text{ and i.h.)} \\ & = & (\texttt{fib}\,(n+1)\,,\,\texttt{fib}\,(n+2)) \quad \text{(def. }\texttt{fib}\text{)} \end{array}$$

The usual reason for a recursively defined function to terminate is that in the recursive calls that function is called with smaller arguments. If this is the case then termination can be proved by a straightforward complete induction (as we did for the Fibonacci function).

However, there are as well recursive functions where the recursive calls are not always smaller. A famous example is the so-called Collatz Problem (or $3x + 1$ Problem): Starting with a given natural number $n > 1$, move to another number according to the following rules:

- if n is even move to $n/2$,
- if n is odd move to $3n + 1$.

The journey ends if on has arrived at number 1. For example

$$7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

Although experiments indicate that this journey ends for any positive starting number, so far nobody was able to prove this.

Exercise 76 Define a function that computes for every natural number $n \geq 1$ the Collatz journey as a list of natural numbers. For example `collatz 7` should evaluate to

```
[7,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2,1]
```

Compute the Collatz journey for your student number.

8.2 Recursion and induction on lists

Recursion on lists

Example: The length of a list (predefined)

```
length :: [a] -> Int
length []     = 0                -- base
length (x:xs) = 1 + length xs   -- step
```

Computation by reduction:

$$\begin{aligned} \texttt{length}\,[1,2] = \texttt{length}\,(1:2:[]) &\Rightarrow 1 + \texttt{length}\,(2:[]) \\ &\Rightarrow 1 + (1 + \texttt{length}\,[]) \\ &\Rightarrow 1 + (1 + 0) \\ &\Rightarrow 2 \end{aligned}$$

The general scheme for recursion on lists is

- Base: define the result for the empty list `[]`
- Step: define the result for a nonempty list `(x:xs)` using the result for `xs`

The sum of a list of numbers

The predefined function `sum` computes for a list of numbers $[x_1, \ldots, x_n]$ the sum $x_1 + \ldots, +x_n$.

The function `sum` is defined by recursion on lists:

```
sum :: (Num a) => [a] -> a
sum []     = 0
sum (x:xs) = x + sum xs
```

Exercise 77 Give a recursive definition of the predefinded function `product` that computes for a list of numbers $[x_1, \ldots, x_n]$ the product, $x_1 * \ldots, *x_n$.

Exercise 78 Give recursive definitions of the predefined functions

```
and, or :: [Bool] -> Bool
```

which return `True` exactly if *every* respectively at *last one* element of the input list is `True`.

Membership test

The predefined function `elem` tests whether or not an element x occurs in a list xs.

```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:xs) = x == y || elem x xs
```

Exercise 79 Functions like `elem` and other functions defined by recursion on lists become slightly more efficient if the defining equations are swapped

```
elem :: Eq a => a -> [a] -> Bool
elem x (y:xs) = x == y || elem x xs
elem x _      = False
```

Explain why.

Concatenation (predefined)

Concatenation of two lists:

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

Concatenation of a list of lists:

```
concat :: [[a]] -> [a]
concat []         = []
concat (xs : xss) = xs ++ concat xss
```

Exercise 80 Prove by induction on xs

```
length(xs ++ ys) = length xs + length ys
```

Solution

Base: We have to show `length(xs ++ ys) = length xs + length ys`.

Left hand side: `length([] ++ ys) = length ys`.

Right hand side: `length [] + length ys = 0 + length ys = length ys`.

Step: The i.h. is `length(xs ++ ys) = length xs + length ys`.

We have to show `length((x:xs) ++ ys) = length (x:xs) + length ys`.

```
length((x:xs) ++ ys) = length(x : (xs ++ ys))     (by def. of (++))
                     = 1 + length(xs ++ ys)       (by def. of length)
                     = 1 + length xs + length ys  (by i.h.)
                     = length (x:xs) + length ys  (by def. of length)
```

Exercise 81 Prove `xs ++ [] = xs`.

Exercise 82 Prove `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`.

Solution

We prove this by induction on `xs`.

Base: We have to show `([] ++ ys) ++ zs = [] ++ (ys ++ zs)`.

`([] ++ ys) ++ zs =  ys ++ zs = [] ++ (ys ++ zs)`.

Step: The i.h. is `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`.

We have to show `((x:xs) ++ ys) ++ zs = (x:xs) ++ (ys ++ zs)`.

```
((x:xs) ++ ys) ++ zs = (x : (xs ++ ys)) ++ zs    (by def. of (++))
                     = x : ((xs ++ ys) ++ zs)    (by def. of (++))
                     = x : (xs ++ (ys ++ zs))    (by i.h.)
                     = (x:xs) ++ (ys ++ zs)      (by def. of (++))
```

`map`

The predefined higher-order function `map` can be defined by recursion on lists as follows:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x) : (map f xs)
```

Exercise 83 Prove `map f (xs ++ ys) = (map f xs) ++ (map f ys)` by recursion on `xs`.

Exercise 84 Prove `map (f . g) xs = map f (map g xs)` by recursion on `xs`.

Remark. The equation in Exercise 84 can be written more shortly

`map (f . g) = (map f) . (map g).`

Exercise 85 Give a recursive definition of the predefined function `take :: Int -> [a] -> [a]` that computes the first n elements of a list.

Exercise 86 Give a recursive definition of the predefined function `drop :: Int -> [a] -> [a]` that drops the first n elements of a list.

Reversing lists

The naive program for computing the reverse of a list $x : xs$ is to recursively reverse xs and append x at the end:

```
rev [] = []
rev (x:xs) = rev xs ++ [x]
```

This works well for short lists, but becomes very slow if the lists get longer. Why?

Exercise 87 Give an estimate of the time complexity of the naive reverse function.

Reversing lists in linear time

```
reverse xs = loop xs []  where
  loop []     ys = ys
  loop (x:xs) ys = loop xs (y:ys)
```

This program has linear run time.

Moreover, it is **tail recursive** because the recursive call of the subroutine `loop` is *not* an argument of another function.

Tail recursive functions can be implemented by efficient while loops without the need for building a *recursion stack*.

Remark. Most of the recursive functions we discussed so far (`fact`, `fib`, `length`, etc.) are not tail recursive.

Proving correctness of `reverse`

To prove that

```
reverse xs = loop xs []  where
  loop []     ys = ys
  loop (x:xs) ys = loop xs (y:ys)
```

computes the same function as the naive program

```
rev [] = []
rev (x:xs) = rev xs ++ [x]
```

we prove

```
loop xs ys = rev xs ++ ys
```

by induction on `xs`. More precisely, the statement we are proving for all lists xs is $P(xs) :\equiv \forall ys\ (\texttt{loop}\ xs\ ys = \texttt{rev}\ xs \mathbin{++} ys)$

Exercise 88 Complete the proof of correctness of the function `reverse`.

Solution

Base, $P([])$

`loop [] ys = ys = [] ++ ys = rev [] ++ ys.`

Step, $P(xs) \to P(x : xs)$

The i.h. is $P(xs)$, that is, $\forall zs$ (`loop` xs $zs =$ `rev` xs $++$ zs) (note the renaming of ys to zs, this is inessential, bit it will make the proof clearer).

We have to show $P(x : xs)$, that is, $\forall ys$ (`loop` $(x{:}xs)$ $ys =$ `rev` $(x{:}xs)$ $++$ ys).

```
loop (x:xs) ys = loop xs (x:ys)        (by def. of loop)
               = rev xs ++ (x:ys)      (by i.h., with zs = x:ys)
               = rev xs ++ ([x] ++ ys) (by def. of (++))
               = (rev xs ++ [x]) ++ ys (by associativity of (++))
               = rev (x:xs) ++ ys      (by def. of rev)
```

It remains to be shown that indeed `reverse xs = rev xs`:

```
reverse xs = loop xs []
           = rev xs ++ []    (by the inductive proof above)
           = rev xs          (we have shown that in general xs ++ [] = xs)
```

Quicksort

To sort a list with head x and tail xs, compute

- $low =$ the list of all elements in xs that are smaller than x,
- $high =$ the list of all elements in xs that are greater or equal than x.

Then, recursively sort low and $high$ and append the results putting x in the middle.

```
qsort :: (Ord a) => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort lows ++ [x] ++ qsort highs
   where
     lows  = filter (x >) xs
     highs = filter (x <=) xs
```

Note that `qsort` does *not* follow the general pattern of list recursion. However, the recursive calls are with lists that are shorter than the input list xs. Therefore, termination and correctness can be proved by complete induction on the length of lists.

Exercise 89 Modular exponentiation

For integers b, e, m with $b > 0$, $e \geq 0$ and $m > 0$ we define

$$\mathsf{expmod}(b, e, m) = b^e \,\mathsf{mod}\, m$$

Give an efficient recursive definition of expmod using the following idea of *repeated squaring*: Assume e is a positive *even* number. Set

$$\begin{aligned} f &= e/2 \\ p &= \mathsf{expmod}(b, f, m) \end{aligned}$$

Then one can express $\mathsf{expmod}(b, e, m)$ in terms of p^2 and m using the square function and mod .

In order to find the solution you may use the fact that mod commutes with multiplication, that is,

$$(x * y) \,\mathsf{mod}\, m = ((x \,\mathsf{mod}\, m) * (y \,\mathsf{mod}\, m)) \,\mathsf{mod}\, m$$

If e is *odd*, then the computation of $\mathsf{expmod}(b, e, m)$ can be reduced to the computation of $\mathsf{expmod}(b, e - 1, m)$.

Use your fast implementation of expmod to compute the last 6 digits (in decimal notation) of x^x where x is your student number.

Compare your program with a naive implementation of expmod.

Remark: Modular exponentiation plays an important role in *cryptography*.

Exercise 90 Newton's Method

- To approximate $\sqrt{x}$, start with 1 (or any other value) as first approximation.
- If y is an approximation of $\sqrt{x}$, then $(y + x/y)/2$ is a better approximation.
- Stop if the approximation y is good enough, say $|y^2 - x| < 0.00001$.

Implement a variant of Newton's Method where the stopping condition is given by an upper bound for the number of iterations, and the result is the list of all approximations computed.

Exercise 91 Towers of Hanoi

There are three pegs, and a number of discs of different sizes which are stacked in order of size on one peg, the smallest at the top.

The entire stack must be moved to a specified target peg by moving upper discs from one peg to another peg with a larger top disc.

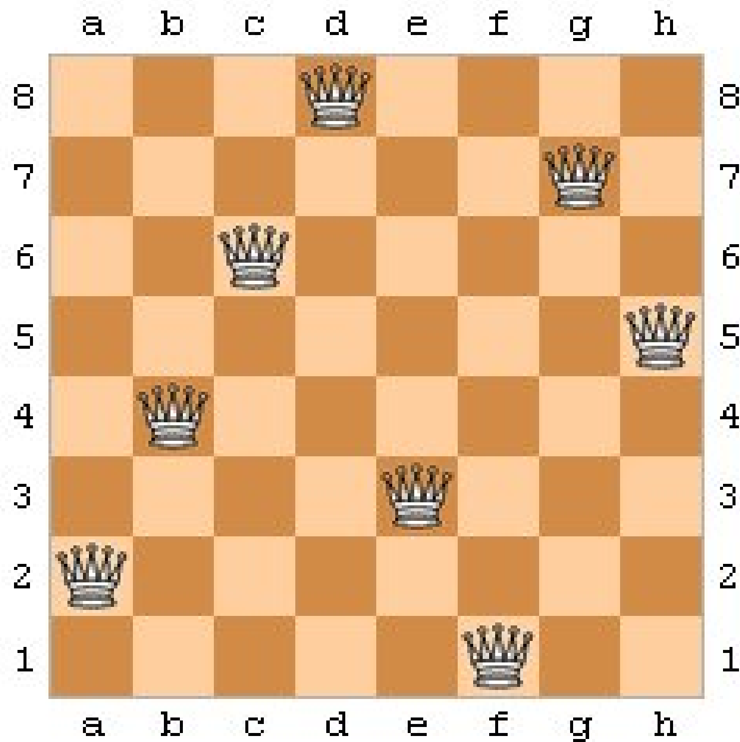
An easy solution proceeds by induction on the number of discs:

- Base: If there is no disc, nothing needs to be done.
- Step: Suppose there are $n + 1$ discs.
  - Move n discs to the auxiliary peg (using the induction hypothesis).
  - Move the remaining disc to the target peg.
  - Move the n discs from the auxiliary peg to the target peg (using the induction hypothesis again).

Represent a move by a triple of pegs specifying source, auxiliary and target peg. Define a function that computes a list of moves moving a stack of n discs from peg 1 to peg 3.

Exercise 92 Eight Queens

Eight queens are to placed on a 8×8 chessboard such that they do not attack each other.



A solution shall be represented by the y-coordinates of the queens listed from left to right. In the example shown this is $[2, 4, 6, 8, 3, 1, 7, 5]$.

Compute a list of *all* solutions to the Eight Queens puzzle.

Hint: Define a function that computes all solutions to the more general problem of placing m queens on an $n \times m$ chessboard such that they don't attack each other.

Exercise 93 Implement *insertion sort*:

To sort a list with head x and tail xs, sort xs and then insert x at the right place.

Exercise 94 Define recursively the list of all nonempty prefixes of a list (for example $[1, 2]$ is a prefix of $[1, 2, 3]$).

Exercise 95 Define recursively the list of all nonempty postfixes of a list (for example $[2, 3, 4, 5]$ is a postfix of $[1, 2, 3, 4, 5]$).

Exercise 96 Use the Exercises 94 and 95 to define the list of all nonempty sublists of a list. (for example $[2, 3]$ and $[1, 2, 3, 4, 5]$ are a sublists of $[1, 2, 3, 4, 5]$).

Hint: A sublist of xs is a postfix of a prefix of xs.

Exercise 97 Let us represent a real polynomial by its list of coefficients:

```
type Polynomial = [Float]
```

Hence a list $[c_0, c_1, c_2, \ldots, c_n]$ represents the polynomial

$$c_0 + c_1 * X^1 + c_2 * X^2 + \ldots + c_n * X^n.$$

Define an *evaluation function* for polynomials,

`evalPol :: Polynomial -> Float -> Float`,

that computes, for every polynomial $[c_0, c_1, c_2, \ldots, c_n]$ and floating point number x, the number

$$c_0 + c_1 * x^1 + c_2 * x^2 + \ldots + c_n * x^n$$

Hints: You may use list comprehension and the predefined Haskell function `sum`. Alternatively you may use the *Horner Scheme*, as taught in the course CS-144, to give a simple (and more efficient) recursive definition of `evalPol`.