This course is partly based on a course by Christian Lüth (University of Bremen) and was developed within the MMiSS project (MultiMedia Instruction in Safe Systems) with financial support by the German Ministry for Education and Research.
The aims of this course

To understand the main ideas and concepts of functional programming.

To learn how to use functional programming techniques to solve computational problems and write well-structured and readable programs.

To learn techniques for proving the correctness of programs.
What is functional programming?

A functional program is an expression.
Example: \((2 \times 3) + 4\)

To *run* a functional program means to *evaluate* it.
For example, the expression above evaluates to 10.

There are no assignments (such as \(x := x+1\)).

Functional programming is very similar to using a calculator.

Functional programming is as powerful as imperative and object-oriented programming.
Some areas where functional programming is applied

Artificial Intelligence
Scientific computation
Theorem proving
Program verification
Safety critical systems
Web programming
Network toolkits and applications
XML parser
Natural Language processing and speech recognition
Data bases
Telecommunication
Graphic programming
Finance and trading
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.
In this course we use the functional programming language *Haskell*. Haskell is named after *Haskell B Curry* (1900 – 1982), an American Mathematician, whose work provided the theoretical foundation of functional programming (\(\lambda\)-calculus).
The main current functional programming languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Types</th>
<th>Evaluation</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Lisp, Scheme</em></td>
<td>type free</td>
<td>eager</td>
<td>via side effects</td>
</tr>
<tr>
<td><em>ML, CAML</em></td>
<td>polymorphic</td>
<td>eager</td>
<td>via side effects</td>
</tr>
<tr>
<td><em>Haskell, Clean</em></td>
<td>polymorphic</td>
<td>lazy</td>
<td>via monads</td>
</tr>
</tbody>
</table>

The technical terms occurring in that table will be explained during in the course.
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
Functional programming - now and in the future

▶ Functional programming becomes more and more widespread.

▶ Functional concepts are being integrated into other programming languages (for example Java).

▶ Extensions by dependent types (Chayenne, Agda)

▶ Big companies begin to adopt functional programming (Ericsson, Microsoft, Deutsche Bank, ...).

▶ Microsoft initiative: F# into .net
Recommended Books

Graham Hutton.
Functional Programming in Haskell.

Simon Thompson.
Addison-Wesley, 1999.

Richard Bird.
Introduction to Functional Programming using Haskell,
Recommended Books (ctd.)

Paul Hudak.
The Haskell School of Expression – Learning Functional Programming through Multimedia.

John O’Donnell, Cordelia Hall and Rex Page.
Discrete Mathematics Using a Computer, 2nd Edition,

Kees Doets and Jan van Eijck.
The Haskell Road to Logic, Maths an Programming.
Recommended Books (ctd.)

Antony J. T. Davie.  

Jeremy Gibbons and Oege de Moor.  
The Fun of Programming.  

Bryan O’Sullivan, Don Stewart, and John Goerzen.  
http://book.realworldhaskell.org
Information on the web

The course homepage
http://www.cs.swan.ac.uk/~csulrich/fp1.html

The Haskell homepage
http://www.haskell.org

Wikipedia
http://en.wikipedia.org
(use for first information on a topic and pointers to the literature)
Lab exercises and coursework will be done in pairs.

The partners of a pair must be in the same level.

You should have found a partner by Monday, 2nd of February.
Lab classes

Attendance: Every pair has to attend one lab class per week

Allocation: Every pair signs up to one of the slots below.

Lab exercises are compulsory for both levels, but are assessed for Level 1 only (10%).

Place: Linux lab, room 217

Time: Mondays, start 2nd of February
  Level 1: 12-1, 1-2
  Level 2: 2-3
  Mop up: 3-4
Coursework

The coursework is the same for Level 1 and Level 2.

Coursework 1 (10%): Monday 16/2 - Monday 2/3.
Coursework 2 (10%): Monday 16/3 - Monday 20/4.

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

**Submission:** put printout with signature of both partners in the wooden box on the second floor. The box will be emptied in the morning after the submission date.
Exams

The exams for Level and Level 2 are weighted differently

Level 1: 70%
Level 2: 80%

There will be two revision lectures towards the end of term.
Communication

Regularly visit the course web page and the notice board on the 2nd floor and read your email for announcements.

You are welcome to ask me questions via email, but please use your student email. I won’t react to emails from other addresses.

I encourage you to use the Student Forum http://cs.fychan.com

Don’t hesitate to ask questions during lectures and lab classes!
Interactive Haskell

We will work with interactive versions of Haskell that make it particularly easy to learn the language and test programs.

- hugs (Haskell User Gofer System) is installed on the Linux and Windows computers in the labs.
- ghci (Glasgow Haskell Compiler Interactive) comes with the GHC distribution and is very similar to hugs.

Information on how to download, install and start hugs and ghci can be found on the course web page.
Using Hugs as a calculator

We start hugs or ghci and evaluate some expressions:

Prelude> 2+3
5
Prelude> (2+3)*5
25
Prelude> max 4 7
7
Defining expressions in a script

We can write definitions of expressions into a text file and load them into hugs.

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

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

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

A Haskell definition consists of

- a type declaration (or signature): `x :: Int`
- the defining equation: `x = 2+3`
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 add 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.

:? lists all available commands.

Commands can be abbreviated by their first letter.
Defining a function

Instead of defining a fixed area we can define a function that computes the area of a circle with a given radius $r$.

```haskell
area :: Float -> Float
area r = r^2 * pi
```

*Main> area 1.5
7.0685835

*Main> area 3
28.274334

The signature `area :: Float -> Float` means that `area` 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.
Functions of more than one argument

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

average3 :: Float -> Float -> Float -> Float
average3 x y z = (x+y+z)/3

*Main> average 3 4
3.5
*Main> average3 3 4 5
4.0

**Exercise:**
Define a function that computes the hypotenuse of a right-angled triangle from its catheti (legs).
A good script contains plenty of comments.

-- That's how we write short comments

{-
Longer comments
can be included like this
-}
Some syntactic issues

- 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)`

- Arguments of functions of several arguments are separated by spaces
  `average 3 4`, but not `average(3,4)`

- Definitions in a script must start at the beginning of a line.
What are types and what are they good for?

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

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

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
Booleans

- Values: True and False
- Predefined functions:
  - `not :: Bool -> Bool` — negation (not)
  - `(&&) :: Bool -> Bool -> Bool` — conjunction (and)
  - `(||) :: Bool -> Bool -> Bool` — disjunction (or)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>not True</code></td>
<td>False</td>
</tr>
<tr>
<td><code>not False</code></td>
<td>True</td>
</tr>
<tr>
<td><code>True &amp;&amp; True</code></td>
<td>True</td>
</tr>
<tr>
<td><code>True &amp;&amp; False</code></td>
<td>False</td>
</tr>
</tbody>
</table>

e.t.c
The signature of a function

The signature \( \text{not} :: \text{Bool} \rightarrow \text{Bool} \) means that \( \text{not} \) is a function

expecting one argument of type \( \text{Bool} \) (input)
and computing a result of type \( \text{Bool} \) (output).

\( \&\& \) :: \( \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \) means that \( \&\& \)
expects two arguments of type \( \text{Bool} \)
and computes a result of type \( \text{Bool} \).

The function \( \&\& \) can be used in infix notation

\[ \text{True} \&\& \text{False} \] (omitting the brackets around \( \&\& \))

but also in prefix notation

\( \&\& \text{True False} \) (with brackets around \( \&\& \))
Example: Exclusive Or

The predefined function \( \texttt{||} \) defines *Inclusive Or*:

\[ x \ \texttt{||} \ y \text{ evaluates to } \text{True} \text{ if } x, \text{ or } y, \text{ or both are } \text{True}. \]

We can define *Exclusive Or* using the predefined functions \( \texttt{||} \), \( \&\& \) and \( \texttt{not} \):

\[
xor :: \text{Bool} \to \text{Bool} \to \text{Bool} \\
xor \ x \ y = (x \ \texttt{||} \ y) \ \&\& \ \texttt{not} (x \ \&\& \ y)
\]

\[
\begin{align*}
xor \ \text{True} \ \text{False} & \Rightarrow \text{True} \\
xor \ \text{True} \ \text{True} & \Rightarrow \text{False} \\
\text{True} \ \texttt{||} \ \text{True} & \Rightarrow \text{True}
\end{align*}
\]
Defining a boolean function by its truth table

Alternatively, we can define the function `xor` by listing its truth table:

\[
\begin{align*}
\text{xor} &: \text{Bool} \to \text{Bool} \to \text{Bool} \\
\text{xor True True} &= \text{False} \\
\text{xor True False} &= \text{True} \\
\text{xor False True} &= \text{True} \\
\text{xor False False} &= \text{False}
\end{align*}
\]
Exercise: Defining other logic gates

Boolean functions are also called *logic gates*.

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

Give two definitions of \texttt{nand}: from predefined gates and by listing its truth table.
Basic numeric types

Computing with numbers

Limited precision $\leftrightarrow$ arbitrary precision
constant cost $\leftrightarrow$ 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
2-(9 + 4) ⇒  -11
-abs -3   ⇒  error
abs (-3)  ⇒  3
div 33 12  ⇒  2
mod 33 12  ⇒  9
gcd 33 12  ⇒  3
Example: The number of logic gates with $n$ inputs

There are $2^n$ different $n$-tuples of boolean values.

Therefore, there are $2^{(2^n)}$ logic gates with $n$ inputs.

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

*Main> gates 2
16
*Main> gates 3
256
*Main> gates 4
65536
*Main> gates 5
0

What’s wrong?
Comparison operators

\[(=), (\ne), (\le), (<), (\ge), (>): \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}\]

\[-9 = 4 \quad \Rightarrow \quad \text{False}\]
\[9 = 9 \quad \Rightarrow \quad \text{True}\]
\[4 \neq 9 \quad \Rightarrow \quad \text{True}\]
\[9 \geq 9 \quad \Rightarrow \quad \text{True}\]
\[9 > 9 \quad \Rightarrow \quad \text{False}\]

But also

\[(=), (\ne), (\le), (<), (\ge), (>): \text{a} \rightarrow \text{a} \rightarrow \text{Bool}\]

where \(a = \text{Bool, Integer, Rational, Float, Double}\)
Prefix vs. infix

Functions of two arguments, for example (+), whose name is composed of special symbols like *, +, /, <, &, e.t.c, can be used

▶ infix without brackets:  x + y
▶ prefix with brackets:  (+) x y

Functions of two arguments, for example div, whose name is composed of letters can be used

▶ infix with inverted commas:  x ‘div‘ y
▶ prefix without inverted commas:  div x y
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
Conversion from and to integers

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

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

Example

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

Does not work because division (/) expects two floating point numbers as arguments, but x has type Int.

Solution:

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

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

2. Define a function that computes the average of three integers, rounded \textit{down} to the nearest integer.
Characters and strings

Notation for characters: ’a’

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

Used infix, (,:) binds stronger than (++)

"Hello " ++ ’W’ : "orld!"  ⇒  "Hello World!"

Is the same as

"Hello " ++ ('W' : "orld!")
Exercise: Repeating a string

Define a function \( \text{rep} \) that repeats a string.

(Don’t forget the signature!)

Define \( \text{rep} \) using concatenation infix. Then modify the definition using concatenation prefix.

Evaluate the expressions

\[
\text{rep} \ "\text{hello!} \"
\]

\[
\text{rep} \ (\text{rep} \ "\text{hello!} \"")
\]

and so on.

Define a function \( \text{rep2} \) that applies the function \( \text{rep} \) to a string twice, that is,

\[
\text{rep2} \ x = \text{rep} \ (\text{rep} \ x).
\]

How many repetitions of the string "hello! " do we get when evaluating the expression \( \text{rep2} \ (\text{rep2} \ (\text{rep2} \ "hello! ")) \)?
If $a$ and $b$ are types, then

$$(a,b)$$

denotes the *cartesian product* of $a$ and $b$ (in mathematics usually denoted $a \times b$).

The elements of $(a,b)$ are pairs $(x,y)$ where $x$ is of type $a$ and $y$ is of type $b$.

$$(3+4,7) :: (\text{Int,Int})$$
$$(342562, ("George","Sand")) :: (\text{Int, (String,String)})$$
Polymorphism and pattern matching

In the Haskell module Prelude (which is loaded automatically when starting hugs or ghci) the projection functions are defined as follows:

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

- \text{fst} and \text{snd} are \textit{polymorphic}, because their signatures contain \textit{type variables} (a and b) which stand for arbitrary types.

- The defining equations have the \textit{pattern} \((x, y)\) on the left side of the equality sign. Therefore, \text{fst} and \text{snd} are defined by \textit{pattern matching}.
Parametric polymorphism

The function \( \text{fst} \) can be used with pairs of any type:

\[
\begin{align*}
\text{fst} (0,1) & \Rightarrow 0 \\
\text{fst} (\text{True},0) & \Rightarrow \text{True} \\
\text{fst} ((0,1),"World") & \Rightarrow (0,1)
\end{align*}
\]

Obviously, the computation does not depend on the types of the inputs.

This kind of polymorphism is called \textit{Parametric polymorphism}:

“One algorithm for all types”
Exercise: swapping and diagonalisation

1. Define a polymorphic function `swap` that swaps the components of a pair, first using pattern matching, then using the functions `fst` and `snd`.

2. What’s wrong with the following definition?

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

   Correct the error.
Wildcards

Formal parameters that are not used can be replaced by the *wildcard* \_.

\[
\text{fst} :: (a, b) \rightarrow a \\
\text{fst} (x, _) = x
\]

\[
\text{snd} :: (a, b) \rightarrow b \\
\text{snd} (_, y) = y
\]

This definition is equivalent to the one given earlier.
Instead of pairs one can have triples, quadruples, e.t.c.

\[
\begin{align*}
\text{first} &:: (a,b,c) \to a \\
\text{first} \ (x,_,_) & = x \\
\text{second} &:: (a,b,c) \to b \\
\text{second} \ (_,y,_) & = y \\
\text{third} &:: (a,b,c) \to c \\
\text{third} \ (_,_,z) & = z
\end{align*}
\]

Note that two wildcards in the same expression stand for different formal parameters:

\[
\text{first} \ (x,_,_) = x \text{ stands for } \text{first} \ (x,y,z) = x
\]
Type synonyms

Suppose we want to define geometric operations on the points in the plane, where a point is given by a pair of floating point numbers.

Instead of repeatedly writing \((\text{Float, Float})\) we can introduce a name for this type:

type Point = (Float, Float)

distance :: Point -> Point -> Float
distance \((x_1, x_2)\) \((y_1, y_2)\) = \(\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}\)

addPoint :: Point -> Point -> Point
addPoint \((x_1, x_2)\) \((y_1, y_2)\) = \((x_1 + y_1, x_2 + y_2)\)

Type synonyms not only shorten the code, they also make programs more readable.
Lists

Haskell has a predefined data type of *polymorphic lists*, \([a]\).

\[
[] :: [a] \quad \text{-- the empty list}
\]

\[
(：) :: a -> [a] -> [a] \quad \text{-- adding an element}
\]

Hence, the elements of \([a]\) are either \([\text{}]\),
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

\[ [1, 3, 5+4] :: [\text{Int}] \]
\[ ['a', 'b', 'c', 'd'] :: [\text{Char}] \]
\[ [[\text{True}, 3<2], []] :: [[[\text{Bool}]]] \]
\[ [(198845, "Cox"), (203187, "Wu") :: [(\text{Int}, \text{String})]] \]
Strings are lists of characters

The type `String` is the same as the type `[Char]` of lists of characters.

However, Haskell has a special way of displaying strings:

\[
\begin{align*}
['H', 'e', 'l', 'l', 'o'] & \Rightarrow "Hello" \\
"Hello" & \Rightarrow "Hello" \\
[0,1,2,3] & \Rightarrow [0,1,2,3] \\
['0','1','2','3'] & \Rightarrow "0123"
\end{align*}
\]
Some predefined polymorphic functions on lists

\[
\begin{align*}
(:) & : a \rightarrow [a] \rightarrow [a] \quad \text{-- adding an element} \\
(++) & : [a] \rightarrow [a] \rightarrow [a] \quad \text{-- concatenating two lists} \\
null & : [a] \rightarrow \text{Bool} \quad \text{-- emptyness test} \\
length & : [a] \rightarrow \text{Int} \\
head & : [a] \rightarrow a \\
tail & : [a] \rightarrow [a] \\
(!!) & : [a] \rightarrow \text{Int} \rightarrow a \quad \text{-- accessing elements by index}
\end{align*}
\]

\[
\begin{align*}
3 & : [4,5] \quad \Rightarrow \quad [3,4,5] \\
[3] & ++ [4,5] \quad \Rightarrow \quad [3,4,5] \\
\text{head} & [3,4,5] \quad \Rightarrow \quad 3 \\
\text{tail} & [3,4,5] \quad \Rightarrow \quad [4,5] \\
[3,4,5] & !! 0 \quad \Rightarrow \quad 3 \\
[3,4,5] & !! 2 \quad \Rightarrow \quad 5
\end{align*}
\]
concat and zip

\[
\text{concat} :: \boxed{[[a]]} \rightarrow \boxed{[a]}
\]

\[
\text{concat} \boxed{[[3,4],[5],[6,7]]} \Rightarrow \boxed{[3,4,5,6,7]}
\]
\[
\text{concat} \boxed{["How ","are ","you?"]} \Rightarrow \boxed{"How are you?"}
\]

\[
\text{zip} :: \boxed{[a]} \rightarrow \boxed{[b]} \rightarrow \boxed{[(a,b)]]}
\]

\[
\text{zip} \boxed{[1,2,3]} \boxed{["A","B","C"]} \Rightarrow \boxed{[\boxed{(1,"A")},\boxed{(2,"B")},\boxed{(3,"C")}]}
\]
\[
\text{zip} \boxed{[1,2,3]} \boxed{["A","B"]} \Rightarrow \boxed{[\boxed{(1,"A")},\boxed{(2,"B")}]}
\]
Pattern matching with lists

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

\[
\begin{align*}
\text{null} & \:: \ [a] \to \text{Bool} \\
\text{null} \ [\] & = \text{True} \\
\text{null} \ _ & = \text{False} \quad --\text{applies if first line does not match} \\
\text{head} & \:: \ [a] \to a \\
\text{head} \ [\] & = \text{error} \ "\text{head of empty list}\" \\
\text{head} \ (x:_\) & = x \quad --\text{brackets required!} \\
\text{tail} & \:: \ [a] \to [a] \\
\text{tail} \ [\] & = \text{error} \ "\text{tail of empty list}\" \\
\text{tail} \ (_\:xs) & = xs
\end{align*}
\]

If you try these definitions out you must change the names of the functions in order to avoid a conflict with the module Prelude.
Example: Nested patterns for lists

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

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

The first equation is equivalent to
exactlytwo (_,_,[]) = True

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

atmosttwo :: [a] -> Bool
atmosttwo (_,_:_:_) = False
atmosttwo _        = True
```
Equidistant numbers

Haskell has a special syntax for lists of equidistant numbers:

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

We may also define functions using this notation:

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

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

\[ \text{interval 3 7} \Rightarrow [3,4,5,6,7] \]
\[ \text{evens 5} \Rightarrow [0,2,4,6,8,10] \]
List comprehension

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

Example: Squaring all odd numbers in a list

\[ x \times x \mid x \leftarrow [1..10], \text{odd } x \]  \Rightarrow  [1, 9, 25, 49, 81]

- \( x \leftarrow [1..10] \) is a *generator*
- \( \text{odd } x \) is a *test*
- \( x \times x \) is the *transformation*

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:

\[
[(x, y) \mid x \leftarrow [1, 2, 3], y \leftarrow [4, 5]] \\
\Rightarrow [(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]
\]

\[
[(x, y) \mid y \leftarrow [4, 5], x \leftarrow [1, 2, 3]] \\
\Rightarrow [(1, 4), (2, 4), (3, 4), (1, 5), (2, 5), (3, 5)]
\]
Example: Pythagorean triples

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

\[ x^2 + y^2 = z^2 \]

\[
\text{pyth} :: \text{Int} \rightarrow [(\text{Int},\text{Int},\text{Int})] \\
\text{pyth} \ n = [(x,y,z) \mid x <- [1..n], \\
             y <- [1..n], \\
             z <- [1..n], \\
             x^2 + y^2 == z^2]
\]

\[
\text{pyth 10} \Rightarrow [(3,4,5),(4,3,5),(6,8,10),(8,6,10)]
\]
Example: Improving the Pythagorean triples

We can avoid essentially repeated solutions by requiring \( x \leq y \) and \( x \) and \( y \) to have no common factor.

Furthermore, we can reduce the search space by generating \( x \geq y \) and \( z \geq y \) only.

pyth1 :: Int -> [(Int,Int,Int)]
pyth1 n = [(x,y,z) | x <- [1..n],
          y <- [x..n],
          gcd x y == 1,
          z <- [y..n],
          x^2 + y^2 == z^2]
Generators with pattern matching

Adding the components of pairs:

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

\[
\text{addPairs} \ ps = [ x + y \mid (x, y) \leftarrow ps ]
\]

\[
\text{addPairs} (\text{zip} [1,2,3] [10,20,30]) \Rightarrow [11,22,33]
\]

Collecting all singletons in a list of lists

\[
\text{singletons} :: [[a]] \rightarrow [[a]]
\]

\[
\text{singletons} \ xss = [[x] \mid [x] \leftarrow xss]
\]

\[
\text{singletons} \ ["take","a","break","!" ] \Rightarrow ["a","!" ]
\]
# Exercises

1. 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.

2. Use list comprehension and the predefined function
   
   \[ \text{sum} :: [\text{Int}] \rightarrow \text{Int} \quad -- \text{the sum of a list of integers} \]

   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

   \[ x_0 \times y_0 + \ldots + x_{n-1} \times y_{n-1} \]

   Hint: Use `zip`. 
We have seen that the operation of addition, (+), is defined on several types (Int, Float, e.t.c.).

We can ask Haskell for the type of (+)

Hugs> :t (+)
(+) :: (Num a) => a -> a -> a

This means that (+) can be used with any type which is a member of the type class Num.

Hence the operation (+) is polymorphic, but its polymorphism is bounded or constrained by the type class Num.

The prefix (Num a) => is called a type constraint.

All overloading in Haskell is implemented via type classes.
The type class **Num**

We can find out more about the type class **Num** by typing:

Hugs> :i Num
-- type class

class (Eq a, Show a) => Num a where
    (+) :: a -> a -> a
    (-) :: a -> a -> a
    (*) :: a -> a -> a
    negate :: a -> a
    abs :: a -> a
    signum :: a -> a
    fromIntegral :: (Integral a, Num b) => a -> b

-- instances:
instance Num Int
instance Num Integer
instance Num Float
instance Num Double
The operations of a type class

The line

```haskell
class (Eq a, Show a) => Num a where
```

says that the class `Num` inherits all operations from the classes `Eq` and `Show` (these will be discussed in a minute). The section

```haskell
class (Eq a, Show a) => Num a where
    (+) :: a -> a -> a
    (-) :: a -> a -> a
    (*) :: a -> a -> a
    negate :: a -> a
    abs :: a -> a
    signum :: a -> a
    fromIntegral :: (Integral a, Num b) => a -> b
```

specifies which operations are available for every type which is a member of the class `Num`. 
The instances of a type class

The section

-- instances:
instance Num Int
instance Num Integer
instance Num Float
instance Num Double

specifies which types are members of the class Num.

Later in the course you will learn how to put new types into a class and how to create your own type class.
Example: adding pairs

Recall the function

\[ \text{addPairs} :: [(\text{Int,Int})] \to [\text{Int}] \]
\[ \text{addPairs} \ \text{ps} = [x + y \mid (x, y) \leftarrow \text{ps}] \]

\[ \text{addPairs} [(1,2),(4,5)] \implies [3,9] \]

Since addition not only works for integers, but for any numeric type, we can generalize the signature:

\[ \text{addPairs} :: (\text{Num } a) \Rightarrow [(a,a)] \to [a] \]

With this signature the function accepts more inputs:

\[ \text{addPairs} [(1.1,2.1),(4.7,5.8)] \implies [3.2,10.5] \]

With the old signature this would have resulted in an error.
Exercise: Scalar product

Recall the exercise of computing for lists of integers $[x_0, \ldots, x_{n-1}]$ and $[y_0, \ldots, y_{n-1}]$, the number

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

scp :: [Int] -> [Int] -> Int
scp xs ys = sum [x*y | (x,y) <- zip xs ys]

Generalise the signature and test it.
Other type classes

Eq  Types with an equality test (==) and an inequality test (/=).

Ord Types with an ordering (<). This includes (>, (<=), (>=), min and max.

Show Types a with a function show :: a -> String. This function is invoked when the result of a computation is shown on the terminal.

All types we have seen so far and many more are members of the classes Eq, Ord and Show.

Membership in these type classes is preserved by pairing and the list operator. For example, if a is in Eq, so is [a].
Example: `elem`

The predefined function `elem` tests whether a given object occurs in a list. Its type is

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

The type constraint is necessary since for testing whether an object occurs in a list we have to be able to compare it with the elements of the list.
Exercise: occurrences in a list

In an earlier exercise we computed the number of occurrences of a character in a string:

\[
\text{occs :: Char -> String -> Int} \\
\text{occs } x \ s = \text{length } [y \mid y \leftarrow s, y == x]
\]

Clearly all that matters here is that we can test whether two characters are equal.

Generalises the signature accordingly.
Exercise: ordered list

Using the predefined function

\texttt{and :: [Bool] -> Bool}

that tests whether all elements of a list of booleans are \texttt{True}, a programmer defined the following function that tests whether a list is ordered:

\texttt{ordered xs = and [xs!!i <= xs!!(i+1) | i <- [0..(length xs - 2)]]}  

Unfortunately, she forgot to write the signature. Can you add it? Try to find the most general signature.

What is good, what is bad about this program?
Ad-hoc polymorphism

The operations provided by a type class are usually implemented for each member of the type class *differently*.

For example, the implementation of (+) is different for integers and floating point numbers.

This kind of polymorphism is usually called *overloading*, or *ad-hoc polymorphism*, or *bounded polymorphism*.

It is available in most current programming languages and is there just called “polymorphism”.
Ad-hoc vs parametric polymorphism

Ad-hoc polymorphism must not be confused with parametric polymorphism which was discussed in earlier lectures.

Parametrically polymorphic functions were introduced in object-oriented programming languages only recently and are there called generics.

In functional programming, genericity refers to yet another, higher-level, form of polymorphism (and will not be discussed in this course).
Exercises

Add the most general signatures to the following functions.

\[
\text{stutterfree } \text{xs} = \text{and } [\text{xs!!i } /= \text{xs!!(i+1)} | i <- [0..k]]
\]
where
\[
k = \text{length } \text{xs} - 2
\]

\[
\text{squares } n = [i^2 | i <- [1..n]]
\]
Case analysis

- **If-then-else**

  \[
  \text{max} :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
  \text{max } x \ y = \text{if } x < y \text{ then } y \text{ else } x
  \]

- **Guarded equations**

  \[
  \text{signum} :: \text{Int} \rightarrow \text{Int} \\
  \text{signum } x \\
  \quad | \ x < 0 \quad = \ -1 \\
  \quad | \ x == 0 \quad = \ 0 \\
  \quad | \ \text{otherwise} \quad = \ 1
  \]
Case analysis with pattern matching

```haskell
null :: [a] -> Bool
null xs = case xs of
  []    -> True
  _     -> False

head :: [a] -> a
head xs = case xs of
  []    -> error "head of empty list"
  (x:_)-> x

tail :: [a] -> [a]
tail xs = case xs of
  []    -> error "tail of empty list"
  (_:xs) -> xs
```
Local definitions: let and where

\[ g :: \text{Float} \to \text{Float} \to \text{Float} \]
\[ g \ x \ y = \frac{\left(x^2 + y^2\right)}{\left(x^2 + y^2 + 1\right)} \]

better

\[ g \ x \ y = \text{let } a = x^2 + y^2 \text{ in } \frac{a}{a + 1} \]

or

\[ g \ x \ y = \frac{a}{a + 1} \text{ where } a = x^2 + y^2 \]
Local definition of a function

The sum of the areas of two circles with radii \( r, s \).

\[
\text{totalArea} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \\
\text{totalArea} \ r \ s = \pi \times r^2 + \pi \times s^2
\]

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

\[
\text{totalArea} :: \text{Float} \rightarrow \text{Float} \rightarrow \text{Float} \\
\text{totalArea} \ r \ s = \\
\quad \text{let} \ \text{circleArea} :: \text{Float} \rightarrow \text{Float} \\
\quad \ \text{circleArea} \ x = \pi \times x^2 \\
\quad \ \text{in} \ \text{circleArea} \ r + \text{circleArea} \ s
\]

Exercise: Use \text{where} instead.
Example: take, drop, sublists

The predefined functions

\[
\text{take}, \text{drop} :: \text{Int} \to \text{[a]} \to \text{[a]}
\]

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

\[
\begin{align*}
take 5 \ "courses" & \Rightarrow \ "cours" \\
drop 5 \ "courses" & \Rightarrow \ "es"
\end{align*}
\]

We use take and drop to define all nonempty sublists of a list:

\[
\text{nesubs} :: \text{[a]} \to \text{[[a]]} \\
\text{nesubs} \ \text{x} = \ [\text{sublist \ i \ j} \ | \\
\hspace{1cm} i \leftarrow [0..(n-1)], \ j \leftarrow [1..(n-i)]]\\n\text{where} \\
\hspace{1cm} n = \text{length \ xs} \\
\hspace{1cm} \text{sublist \ i \ j} = \text{take \ j} \ (\text{drop \ i} \ \text{xs})
\]

Due to an elaborate layout Haskell programs do not need many brackets and are therefore well readable. The layout rules are rather intuitive:

- definitions must start at the beginning of a line,
- the body of a definition must be indented against the function defined,
- lists of equations and other special constructs must properly line up.
Recursion

Recursion = defining a function in terms of itself.

```
fact :: Int -> Int
fact n = if n == 0
    then 1 -- base
    else n * fact (n - 1) -- step (rec. call)
```

Does not terminate if n is negative. Therefore

```
fact :: Int -> Int
fact n
| n < 0   = error "negative argument to fact"
| n == 0  = 1
| n > 0   = n * fact (n - 1)
```
Example: Fibonacci numbers

The Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, …

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

Due to two recursive calls this program has exponential run time.
Example: Fibonacci numbers improved

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

```haskell
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)
```

The function \texttt{fib} is correct by definition (the Fibonacci numbers are defined exactly as in the program).

The faster function \texttt{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 \texttt{fib} \( n \) for \( n > 40 \).

However we can \textit{prove} the equation by induction.
Recall the proof principle of \textit{induction on natural numbers}:

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

\begin{itemize}
  \item \textit{Induction Base: }$P(0)$.
  \item \textit{Induction Step: }$P(n)$\textit{ implies }$P(n + 1)$, for all natural numbers $n$.
\end{itemize}

In order to prove the correctness of the function $\text{fib1}$ we prove something about the auxiliary function $\text{fibpair}$, namely,

\[
\text{fibpair } n = (\text{fib } n, \text{fib} (n + 1))
\]

(then $\text{fib1 } n = \text{fst} (\text{fibpair } n) = \text{fib } n$)
Proving the correctness of \( \text{fib1} \)

\[
P(n) \equiv \text{fibpair } n = (\text{fib } n, \text{fib}(n + 1))
\]

- **Base, \( P(0) \):** \( \text{fibpair } 0 = (1, 1) = (\text{fib } 0, \text{fib } 1) \)

- **Step:** Assume, as induction hypothesis (i.h.) that \( P(n) \) holds. We show \( P(n + 1) \):

\[
\text{fibpair } (n + 1) \\
= (\text{fib}(n + 1), \text{fib } n + \text{fib}(n + 1)) \quad \text{(def. fibpair and i.h.)} \\
= (\text{fib}(n + 1), \text{fib } (n + 2)) \quad \text{(def. fib)}
\]
Recursion on lists

Example: The length of a list (predefined)

\[\text{length} :: [a] \rightarrow \text{Int}\]
\[\text{length} \; [] \; = \; 0 \quad \quad -- \text{base}\]
\[\text{length} \; (x:xs) \; = \; 1 \; + \; \text{length} \; xs \quad -- \text{step}\]

Computation by reduction:

\[\text{length} \; [1,2] \; = \; \text{length} \; (1 : 2 : []) \quad \Rightarrow \quad 1 \; + \; \text{length} \; (2 : [])\]
\[\Rightarrow \quad 1 \; + \; (1 \; + \; \text{length} \; [])\]
\[\Rightarrow \quad 1 \; + \; (1 \; + \; 0)\]
\[\Rightarrow \quad 2\]
Examples: sum and product (predefined)

Given a list of numbers \([x_1, \ldots, x_n]\) we wish to compute their sum, \(x_1 + \ldots, + x_n\) and their product, \(x_1 \times \ldots, \times x_n\).

\[
\text{sum} :: \text{(Num a)} \Rightarrow [a] \rightarrow a \\
\text{sum} [] = 0 \\
\text{sum} (x:xs) = x + \text{sum} \hspace{1em} xs
\]

Exercises:

- Write a program for product.
- Write the programs for sum using the case construct.
**Concatenation (predefined)**

\[ (++): [a] \rightarrow [a] \rightarrow [a] \]

\[
[] \quad ++ \quad ys \quad = \quad ys
\]

\[
(x:xs) \quad ++ \quad ys \quad = \quad x : (xs ++ ys)
\]

Exercise: prove

\[
\text{length}(xs ++ ys) = \text{length} x + \text{length} ys
\]

by induction on \( xs \).

Concatenation of a list of lists:

\[
\text{concat} :: [[a]] \rightarrow [a]
\]

\[
\text{concat} \quad [] \quad = \quad []
\]

\[
\text{concat} \quad (xs : xss) \quad = \quad xs \quad ++ \quad \text{concat} \quad xss
\]
zip (predefined)

\[ \text{zip} :: [a] \rightarrow [b] \rightarrow [(a,b)] \]
\[ \text{zip } [] \_ = [] \]
\[ \text{zip } \_ [] = [] \]
\[ \text{zip } (x:xs) (y:ys) = (x,y) : \text{zip } xs \ ys \]

Shorter (and slightly more efficient):

\[ \text{zip} :: [a] \rightarrow [b] \rightarrow [(a,b)] \]
\[ \text{zip } (x:xs) (y:ys) = (x,y) : \text{zip } xs \ ys \]
\[ \text{zip } \_ \_ = [] \]
Example: Ordered lists, again

Recall the declarative, but inefficient program for testing whether a list is ordered:

```haskell
ordered :: (Ord a) => [a] -> BOOL
ordered xs = and [xs!!i <= xs!!(i+1) |
                   i <- [0..(length xs - 2)]]
```

Here is an efficient recursive program

```haskell
ordered1 (x:y:xs) = x <= y && ordered1 (y:xs)
ordered1 _ = True
```
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.

\[
\text{qsort} :: (\text{Ord } a) \Rightarrow [a] \rightarrow [a] \\
\text{qsort } [] = [] \\
\text{qsort } (x:xs) = \text{qsort } lows \, ++ \, [x] \, ++ \, \text{qsort } highs \\
\text{where} \\
\text{lows } = [y \mid y \leftarrow xs, y < x] \\
\text{highs } = [y \mid y \leftarrow xs, y \geq x]
\]
Exercises

1. 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.

2. Define the factorial function using the predefined function product.

3. Implement insertion sort:

   To sort a list with head x and tail xs, sort xs and then insert x at the right place.
4. Define recursively the list of all nonempty prefixes of a list (for example \([1, 2]\) is a prefix of \([1, 2, 3]\)).

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

6. Use 3. and 4. to define the list of all nonempty sublists of a list.
   
   Hint: A sublist of \(xs\) is a postfix of a prefix of \(xs\).
7. Let us represent a real polynomial by its list of coefficients:

   \[ \text{type } \text{Polynomial} = [\text{Float}] \]

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

\[ c_0 + c_1 \times x^1 + c_2 \times x^2 + \ldots + c_n \times x^n. \]

Define an evaluation function for polynomials,

\[ \text{evalPol} :: \text{Polynomial} \rightarrow \text{Float} \rightarrow \text{Float}, \]

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

\[ c_0 + c_1 \times x^1 + c_2 \times x^2 + \ldots + c_n \times x^n \]

Hints: You may use list comprehension and the predefined Haskell function \text{sum}. Alternatively you may use the \textit{Horner Scheme}, as taught in the course CS-144, to give a simple (and more efficient) recursive definition of evalPol.
Recursion vs. Loops

In Haskell and most other functional languages loops are expressed by recursion.

Here is a simple example:

An imperative way of computing the length of a list is to count how often one can perform the `tail` operation until one arrives at the empty list.

We compare the imperative pseudo-code with the corresponding Haskell program:
From a while-loop to a recursive program

procedure length
    input xs;
    n = 0;
    while not(null xs) do
        n = n+1;
        xs = tail xs;
    od
    return n

The corresponding recursive Haskell program:

length xs = loop xs 0 where
    loop xs n = if not (null xs)
        then let n’ = n+1
            xs’ = tail xs
            in loop xs’ n’
        else n
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:

\[
\text{reverse } [] = [] \\
\text{reverse } (x:xs) = \text{reverse } xs \; ++ \; [x]
\]

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

A better method is to repeatedly add the heads of the input list $xs$ to another list $ys$, which initially is empty, and return $ys$ as result when $xs$ has become empty (that’s what you would do to reverse a deck of cards).

Again, we compare the imperative with the functional solution:
Reversing lists: imperatively and functional

procedure reverse
    input xs
    ys = [];
    while not(null xs) do
        ys = head xs : ys;
        xs = tail xs;
    od
    return ys

reverse xs = loop xs [] where
loop xs ys = if not(null xs)
    then let ys’ = head xs : ys
        xs’ = tail xs
        in loop xs’ ys’
    else ys
Simplifying the recursive solutions

The recursive programs `length` and `reverse` can be easily simplified. For example,

```haskell
reverse xs = loop xs [] where
  loop xs ys = if not(null xs)
    then let ys' = head xs : ys
        xs' = tail xs
        in loop xs' ys'
    else ys

is equivalent to

reverse xs = loop xs [] where
  loop [] ys = ys
  loop (x:xs) ys = loop xs (y:ys)
```
The recursive programs corresponding to a while-loop are of a particular form called *tail recursion*.

The point is that the recursive call is **not** the argument of another function.

Most compilers of functional languages transform a tail recursion directly into a while loop, hence no time and space is wasted for building a stack. In other words: tail recursions are as efficient as imperative while loops.
Persistence

- In the imperative programs `length` and `reverse` the input `xs` is *destroyed* (`xs` becomes the empty list). This is the case with many other programs (for example for sorting lists).

- In a functional programs arguments are never destroyed. This principle is called *persistence of data*. 
Imperatively, order matters

The order of assignments in an imperative program matters (and this often leads to errors). For example, changing the order of the assignments in the while-loop for reverse yields

```plaintext
procedure badreverse
    input xs
    ys = [];
    while not(null xs) do
        xs = tail xs;
        ys = head xs : ys;
    od
    return ys
```

This program returns an error for any input because it attempts to compute the head of the empty list before the while-loop is finished.
In contrast to this, rearranging the order of the equations in a let or where construct never affects the behaviour of a functional program. For example, the functional program

\[
\text{reverse } xs = \text{loop } xs [] \text{ where }
\]
\[
\text{loop } xs \ ys = \text{if not(null } xs) \text{ then let } ys' = \text{head } xs : ys \text{ in loop } xs' \ ys' \text{ else } ys
\]

is equivalent to

\[
\text{reverse } xs = \text{loop } xs [] \text{ where }
\]
\[
\text{loop } xs \ ys = \text{if not(null } xs) \text{ then let } xs' = \text{tail } xs \text{ in loop } xs' \ ys' \text{ else } ys
\]
Induction on lists

In order to prove

for all finite lists \(xs\), property \(P(xs)\) holds

it suffices to to prove

- **induction base**: \(P([])\)
- **induction step**: assuming \(P(xs)\), show \(P(x : xs)\) holds for all \(x\).

Here, \(P(xs)\) is the induction hypothesis.
Example: associativity of (++)

Recall the definition of list concatenation:

\[ (++) :: [a] \rightarrow [a] \rightarrow [a] \]

\[ [] ++ ys = ys \]

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

Show that (++) is associative, that is, the equation

\[ (xs ++ ys) ++ zs = xs ++ (ys ++ zs) \]

holds for all \( xs, ys, zs \).

We prove this by list induction on \( xs \).
Inductive proof of associativity of (++)

- **Base:** 
  
  \[
  (\text{[]} \; ++ \; ys) \; ++ \; zs = ys \; ++ \; zs = \text{[]} \; ++ \; (ys \; ++ \; zs).
  \]

- **Step:** Assume \((xs \; ++ \; ys) \; ++ \; zs = xs \; ++ \; (ys \; ++ \; zs)\) (i.h.).

  We have to show

  \[
  ((x:xs) \; ++ \; ys) \; ++ \; zs = (x:xs) \; ++ \; (ys \; ++ \; zs)
  \]

  for all \(x\).

\[
\begin{aligned}
((x:xs) \; ++ \; ys) \; ++ \; zs \\
= (x : (xs ++ ys)) \; ++ \; zs & \quad \text{(by def. of (++) )} \\
= x : ((xs ++ ys) ++ zs) & \quad \text{(by def. of (++) )} \\
= x : (xs ++ (ys ++ zs)) & \quad \text{(by i.h.)} \\
= (x:xs) ++ (ys ++ zs) & \quad \text{(by def. of (++) )}
\end{aligned}
\]
Exercises

- Prove $xs ++ [] = xs$.

- Prove $\text{length} \ (xs \ ++ \ ys) = \text{length} \ xs + \text{length} \ ys$.
  Recall

  $$\begin{align*}
  \text{length} &: [a] \to \text{Int} \\
  \text{length} \ [] &= 0 \\
  \text{length} \ (x : xs) &= 1 + \text{length} \ xs
  \end{align*}$$
Exercise: reversing and concatenation

Recall the naive reverse function

\[
\text{rev} :: [a] \rightarrow [a] \\
\text{rev} \; [] \; = \; [] \\
\text{rev} \; (x \; : \; xs) \; = \; \text{rev} \; xs \; ++ \; [x]
\]

Prove \( \text{rev} \; (xs \; ++ \; ys) \; = \; \text{rev} \; ys \; ++ \; \text{rev} \; xs \).
Exercise

Prove that the tail-recursive program for reversing lists

\[
\text{reverse } xs = \text{loop } xs \; [] \; \text{ where}
\]
\[
\begin{align*}
\text{loop } [] \; ys &= ys \\
\text{loop } (x:xs) \; ys &= \text{loop } xs \; (y:ys)
\end{align*}
\]

is correct by proving that it computes the same function as the naive program

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

Hint: prove

\[
\text{loop } xs \; ys = \text{rev } xs \; ++ \; ys
\]

by induction on \(xs\).
Exercise: The 3x + 1 Problem

You are at city \( n \) where \( n \) is your student number. Your goal is to reach city 1. The rules for the journey are as follows:

▶ If you are at city \( k \), then
  ▶ if \( k \) is even, go to city \( k/2 \),
  ▶ if \( k \) is odd, go to city \( 3 \times k + 1 \).

Compute the list of all cities visited on your journey. What is the largest city (number) visited?

Remark: It is an open problem whether the journey terminates for every number. This problem is also known as the 3x + 1 Problem, or Collatz’ Problem.
Exercise: Modular exponentiation

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

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

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

$$f = e/2$$
$$p = \text{expmod}(b, f, m)$$

Then one can express $\text{expmod}(b, e, m)$ 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,

$$(x \ast y) \mod m = ((x \mod m) \ast (y \mod m)) \mod m$$

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

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

Eight queens are to be placed on an 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 × m chessboard such that they don’t attack each other.
Motivating higher-order functions: Sorting a data base

Consider a data base of persons’ names and ages:

type Person = (String, String, Int)

db :: [Person]
db = [("Fawlty","Sybil",36),("Fawlty","Basil",43),
       ("Shearman","Polly",28)]

We want a program that is able sort the database according to varying criteria, for example, by surname, or by age.

▶ The polymorphic sorting function

qsort :: (Ord a) => [a] -> [a]

doesn’t help here, because it requires a type with a fixed ordering.

▶ We need a program which accepts an ordering as an input.

▶ Such a program can be written in Haskell (and most other functional programming languages) as a higher-order function.
Sorting with respect to an ordering

sortBy :: (a -> a -> Bool) -> [a] -> [a]
ssortBy ord [] = []
ssortBy ord (x:xs) = sortBy ord low ++ [x] ++ sortBy ord high
    where
        low = [y | y <- xs, y 'ord' x]
        high = [y | y <- xs, not(y 'ord' x)]

We use this function with some orderings on the database

fn, age :: Person -> Person -> Bool
fn (_,f1,_) (_,f2,_) = f1 < f2
age (_,_,a1) (_,_,a2) = a1 < a2

*Main> sortBy fn db
["Fawlty","Basil",43),("Shearman","Polly",28),("Fawlty","Sybil"
*Main> sortBy age db
["Shearman","Polly",28),("Fawlty","Sybil",36),("Fawlty","Basil"
Flexibilty through higher-order functions

The fact that our sorting function accepts an arbitrary ordering as input gives us a degree of flexibility which cannot be achieved by a conventional “first-order” program.

For example, we can order the data base by the length of surname and firstname together, listing longer names first:

\[
\text{sflh} :: \text{Person} \rightarrow \text{Person} \rightarrow \text{Bool} \\
\text{sflh} (s1,f1,\_) (s2,f2,\_) = \text{length} (s1++f1) > \text{length} (s2++f2)
\]

*Main> sortBy sflh db
[('Shearman','Polly',28),('Fawlty','Sybil',36),('Fawlty','Basil',43)]

But we can as well use the default (lexicographic) ordering:

*Main> sortBy (<) db
[('Fawlty','Basil',43),('Fawlty','Sybil',36),('Shearman','Polly')

*Main>
map

In the following, we discuss some useful predefined higher-order functions.

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

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

map can be defined using list comprehension:

\[
\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
\text{map } f \ xs = [f \ x \mid x \leftarrow xs]
\]

*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]
filter

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

filter can be defined as follows:

\[
\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow \text{[a]} \rightarrow \text{[a]}
\]

\[
\text{filter \ test} \ \text{x}s = \{x \mid x \leftarrow \text{x}s, \ \text{test} \ x\}
\]

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

The functions

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

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

*Main> takeWhile even [2,4,6,7,8,10]
[2,4,6]
*Main> dropWhile even [2,4,6,7,8,10]
[7,8,10]

Exercise: Give recursive definitions of takeWhile, dropWhile.
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:** Define zipWith from zip using list comprehension.
Composition ( . )

Two functions \( f :: a \to b \) and \( g :: b \to c \) can be composed

\[
(g \ . \ f) :: a \to c
\]

The definition of composition is:

\[
(sqrt \ . \ abs) (-9) \Rightarrow sqrt (abs (-9)) \Rightarrow sqrt 9 \Rightarrow 3
\]
Sections

- A type \( s \to b \to c \) is shorthand for \( a \to (b \to c) \).

- Therefore, if \( f :: a \to b \to c \) and \( textttx :: a \), then \( f \ x :: b \to c \).

- \( f \ x \) is called a *section* of \( f \).

- For example, \((+) :: Int \to Int \to Int\) has a section \((+) 5 :: Int \to Int\).

- \((+) 5\) is the function that adds 5 to any input \(((+) 5) 3 \Rightarrow 8\).

- \(((+) 5) 3\) is the same as \((+) 5 3\) and \(5 + 3\).

- Instead of \((+) 5\) one may write \((5 +)\).

- \((< 5) :: Int \to Bool\) is the function that tests whether a number is less than 5.
  \((< 5) 7 \Rightarrow 7 < 2 \Rightarrow False\)
Some applications of Sections

map (+5) [1, 2, 3] ⇒ [6, 7, 8]
map (5+) [1, 2, 3] ⇒ [6, 7, 8]
filter (<2) [1, 2, 3] ⇒ [1]
filter (2<) [1, 2, 3] ⇒ [3]

intersect :: (Eq a) => [a] -> [a] -> [a]
intersect xs ys = filter ('elem' xs) ys

*Main> intersect [1..10] [3..20]
[3,4,5,6,7,8,9,10]
Points revisited

We now turn into ‘practice’ what he have learnt so far by doing some simple image processing.

Don’t worry, image processing will not be a topic of the exam. It is just an example illustrating the use of functional concepts.

First, we recall some basic operations on points:

type Pt = (Float,Float)

addPt :: Pt -> Pt -> Pt
addPt (x1,y1) (x2,y2) = (x1+x2,y1+y2)

scalePt :: Float -> Pt -> Pt
scalePt s (x,y) = (s*x,s*y)

distPt :: Pt -> Pt -> Float
distPt (x1,y1) (x2,y2) = sqrt ((x1-x2)^2 + (y1-y2)^2)
How to represent an image?

Some possibilities for representing a black-and-white image:

1. As a list of points (See maths practicals).

   type Image = [Pt]

2. As a list of horizontal lines, where a horizontal line is a list of Booleans, ‘True’ representing ‘black’:

   type Picture = [[[Bool]]]

**Advantage of 1.** Rotation, scaling etc. are easy.

**Advantage of 2.** Printing (for example as a string) is easy.
Showing a picture

We first transform a picture into a string inserting the *newline character ‘\n’* at the beginning of each line:

\[
\text{showPicture} :: \text{Picture} \to \text{String} \\
\text{showPicture } \text{pic} \\
\quad = \text{concat ['\n' : map boolToChar bs | bs <- pic]} \\
\quad \text{where boolToChar True } = '\@' \\
\quad \quad \text{boolToChar False } = '\.'
\]

For displaying the result we invoke the function

\[
\text{putStrLn} :: \text{String} \to \text{IO ()}
\]

that sends a string to the standard output. \text{IO ()} is the type of *pure actions* (side effects). This type will be discussed in greater detail later.

\[
\text{sp} :: \text{Picture} \to \text{IO ()} \\
\text{sp } = \text{putStrLn . showPicture}
\]
Some pictures

pic1, pic2, pic3 :: Picture
pic1 = [[abs(i-j)< 10 | i <- ix] | j <- ix]
pic2 =
    [[(i-20)^2+(j-30)^2 < 120 | i <- ix] | j <- ix]
pic3 =
    [[even(i`div`10+j`div`10)|i <- ix]|j <- ix]
pic4 =
    [[even((i^2+j^2)`div`50)|i <- ix]|j <- ix]

ix :: [Int]
ix = [0..60]

Use a large terminal, or a small font and type

sp pic1
etc.
Bitmaps

The representations by *images* and *pictures* also have disadvantages, for example:

- For *images*, inversion (swapping black and white) is problematic.
- For *pictures*, the boundaries are hard-wired and transformations like rotation are awkward to implement.

Both representations are too concrete and low-level and therefore not flexible enough.

A very natural, abstract and useful representation is given by *bitmaps*, that is, functions from points to the Booleans:

\[
\text{type Bitmap} = \text{Pt} \rightarrow \text{Bool}
\]
From bitmaps to pictures: Canvas and resolution

In order to display a bitmap we transform it into a picture. To carry out this transformation we need to sample the bitmap. The sampling space is specified by a *setting* consisting of

- the *canvas*, that is, a rectangular part of the 2-dimensional plane, given by the bottom left and the top right corner.
- the *resolution*, that is, the number of grid points in $x$- and $y$-direction.

```haskell
type Setting = (Pt, Pt, Int, Int)
```
Taking pictures

The sampling space determined by a setting is a list of horizontal lines where every horizontal line is a list of points.

```haskell
samples :: Setting -> [[[Pt]]]
samples ((xbl,ybl),(xtr,ytr),n,m) = map hline ys
  where
    hline y = [(x,y) | x <- xs]
    xs = [xbl,xbl+dx..xtr]
    ys = [ytr,ytr-dy..ybl]
    dx = (xtr - xbl) / fromIntegral n
    dy = (ytr - ybl) / fromIntegral m
```

takePicture :: Setting -> Bitmap -> Picture
takePicture set f = map (map f) (samples set)
Showing bitmaps

```haskell
sb :: Bitmap -> IO()
sb = sp . takePicture set0

set0 :: Setting
set0 = ((-1.5,-1.5),(1.5,1.5),80,40)

Examples:

bm1, bm2, bm3, bm4, bm5 :: Bitmap
bm1 p = distPt p (0,0) < 1   -- circle with radius 1
bm2 (x,y) = y < sin (pi*x)   -- area below sin
bm3 (x,y) = abs (y - sin (pi*x)) < 0.05
               -- graph of sin
bm4 (x,y) = max (abs x) (abs y) < 0.7   -- square
bm5 (x,y) = abs x + abs y < 1         -- diamond
```
Inversion, union, intersection

invertBm :: Bitmap -> Bitmap
invertBm f = not . f

unionBm :: Bitmap -> Bitmap -> Bitmap
unionBm f g p = f p || g p

interBm :: Bitmap -> Bitmap -> Bitmap
interBm f g p = f p && g p

unionsBm :: [Bitmap] -> Bitmap
unionsBm fs p = or [f p | f <- fs]

intersBm :: [Bitmap] -> Bitmap
intersBm fs p = and [f p | f <- fs]

xorBm :: Bitmap -> Bitmap -> Bitmap
xorBm f g p = f p /= g p
To shift a bitmap by a vector (point) we pre-compose it with the map shifting a point by the opposite vector:

\[
\text{shiftBm} :: \text{Pt} \rightarrow \text{Bitmap} \rightarrow \text{Bitmap} \\
\text{shiftBm} \; p \; f = f \; \cdot \; \text{addPt} \; (\text{scalePt} \; (-1) \; p)
\]
Rotation

To rotate a point, anti clockwise, around the centre of the co-ordinate system, we apply the well-known linear transformation involving $\sin$ and $\cos$ (see maths practicals).

$$\text{rotate} :: \text{Float} \to \text{Pt} \to \text{Pt}$$
$$\text{rotate} \ \alpha \ (x, y) = (a \times x - b \times y, b \times x + a \times y) \ \text{where}$$
$$a = \cos \ \alpha$$
$$b = \sin \ \alpha$$

To rotate a bitmap by an angle, we must pre-compose it with the map rotating a point by the opposite angle:

$$\text{rotateBm} :: \text{Angle} \to \text{Bitmap} \to \text{Bitmap}$$
$$\text{rotateBm} \ \alpha \ f = f \ . \ \text{rotate} \ (-\alpha)$$
Showing images

takeBm :: Image -> Bitmap

\[
\text{takeBm \ ps \ q = or \ [distPt \ p \ q < 0.1 \mid p \leftarrow ps]}
\]

set1 = ((-3,-3),(3,3),80,40)

si :: Image -> IO ()

\[
\text{si = sp \ . \ takePicture \ set1 \ . \ takeBm}
\]

Dr. Beckmann’s car:

acar :: Image

\[
\text{acar = [(0,1),(1,1),(2,1),\
\hspace{1cm} (-2,0),(-1,0),(0,0),(1,0),(2,0),\
\hspace{1cm} (-1,-1),(1,-1)]}
\]

-- si acar
Simple Data

We now study so-called “algebraic data types” (“data types”, for short).

A simple predefined algebraic data type are the booleans:

```haskell
data Bool = False | True

    deriving (Eq, Show)
```

- The data type `Bool` contains exactly the `constructors` `False` and `True`.
- The expression ‘`deriving(Eq, Show)`’ puts `Bool` into the type classes `Eq` and `Show` using default implementations of `==, /=` and `show`.
Example: Days of the week

We may also define our own data type:

```haskell
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
         deriving(Eq, Ord, Enum, Show)
```

- **Ord**: Constructors are *ordered* from left to right
- **Enum**: * Enumeration and numbering

```haskell
fromEnum :: Enum a => a -> Int
toEnum    :: Enum a => Int -> a
```
Wed < Fri ⇒ True
Wed < Wed ⇒ False
Wed <= Wed ⇒ True

max Wed Sat ⇒ Sat

fromEnum Sun ⇒ 0
fromEnum Mon ⇒ 1
fromEnum Sat ⇒ 6

toEnum 2 ⇒ ERROR - Unresolved overloading ...
toEnum 2 :: Day ⇒ Tue
toEnum 7 :: Day ⇒ Program error ...

enumFromTo Mon Thu ⇒ [Mon, Tue, Wed, Thu]
[(Mon)..(Thu)] ⇒ [Mon, Tue, Wed, Thu]
Pattern matching

Example: Testing whether a day is a work day

- Using several equations
  ```haskell
  workDay :: Day -> Bool
  workDay Sun = False
  workDay Sat = False
  workDay _   = True
  ```

- The same function using the case construct:
  ```haskell
  workDay :: Day -> Bool
  workDay d = case d of
    Sun    -> False
    Sat    -> False
    _      -> True
  ```
Using the derived ordering or equality test

workDay :: Day -> Bool
workDay d = Mon <= d && d <= Fri

workDay :: Day -> Bool
workDay d
  |d == Sat || d == Sun = False
  |otherwise = True
Example: Computing the next day

- Using enumeration

  ```haskell
  dayAfter :: Day -> Day
  dayAfter d = toEnum ((fromEnum d + 1) `mod` 7)
  ```

- The same with the composition operator

  ```haskell
  dayAfter :: Day -> Day
  dayAfter = toEnum . (\(\) \(\) `mod` 7) . (+1) . fromEnum
  ```

Exercise: Define `dayAfter` without `mod` using case analysis instead.
data is a keyword for introducing a new *data type* with *constructors*.

Data types defined in this way are called *algebraic data types*.

Names of data types and constructors begin with an *upper case letter*.

*Definition by cases* can be done via *pattern matching* on constructors.
Let us return to the topic image processing.

We use representations of simple geometric shapes as an example for composite data.

data Shape = Circle Pt Float
             | Rectangle Pt Pt
             deriving (Eq, Show)

- Circle p r represents the circle with center p and radius r.
- Rectangle p1 p2 represents the (upright) rectangle with bottom left corner p1 and top right corner p2.
The type of a constructor

In a data type declaration

```
data DataType = Constr Type1 ... Typen | . . .
```

the type of the constructor is

```
Constr :: Type1 -> ... -> Typen -> DataType
```

In our examples:

```
True, False :: Bool
Sun,...,Sat :: Day
Circle :: Pt -> Float -> Shape
Rectangle :: Pt -> Pt -> Shape
```
### Shifting shapes, creating square


definition shiftShape :: Pt -> Shape -> Shape
shiftShape p sh = case sh of
  Circle q r = Circle (addPt p q) r
  Rectangle p1 p2 = Rectangle (addPt p p1) (addPt p p2)

A square with bottom left corner p and side length a:


definition square :: Pt -> Float -> Shape
square p a = Rectangle p (addPt p (a,a))
Centre of a shape

centre :: Shape -> Pt
centre (Circle p _) = p
centre (Rectangle (x1,y1) (x2,y2))
    = ((x1+x2)/2,(y1+y2)/2)

Exercises:

- Use the case construct instead.
- Compute the number of corners of a shape.
- Compute the perimeter of a shape.
- Compute the area of a shape.
- Compute the diameter of a shape (the largest possible distance of two points in the shape).
Showing shapes

\[ \text{inShape :: Shape} \rightarrow \text{Bitmap} \]

\[ \text{inShape} \ (\text{Circle} \ p \ r) \ q = \text{distPt} \ p \ q \leq r \]

\[ \text{inShape} \ (\text{Rectangle} \ (x_1,y_1) \ (x_2,y_2)) \ (x,y) \]
\[ = x_1 \leq x \land x \leq x_2 \land y_1 \leq y \land y \leq y_2 \]

\[ \text{instance Show Shape where} \]
\[ \quad \text{show} = \text{showPicture} \ . \ \text{takePicture set0} \ . \ \text{inShape} \]

Since we have included the data type \text{Shape} into the type class \text{Show} we now can display a circle by simply typing

\[ \text{Circle} \ (0,0) \ 1 \]
Example: A car

ucar :: Bitmap
ucar = (unionsBm . map inShape)
    [Rectangle (-1,-1) (1,-0.5),
    Rectangle (-0.4,-0.5) (0.6,-0.1),
    Circle (-0.6,-1) 0.2,
    Circle (0.6,-1) 0.2]

Try

sb ucar
sb (rotateBm ((1/4)*pi) ucar)
sequence_ [sb (rotateBm ((n/4)*pi) ucar) | n <- [0..8]]

The function sequence_ :: [IO ()] -> IO () executes all elements of a list of actions in sequence.
Summary

- Using the `data` keyword the user can introduce new structured data.

- In a data type declaration
  
  ```haskell
data DataType = Constr Type1 ... Typen | . . .
```
  
  the type of the constructor is

  ```haskell
Constr :: Type1 -> ... -> Typen -> DataType
```

- *Definition by cases* on structured data can be done using the case construct. Wild cards (underscores) are allowed.
Recursive data

The most common recursive data type is the type of lists:

\[
data \ [a] = [] \mid a : [a]
\]

Without using the special infix (and aroundfix) notation the definition of lists would read as follows:

\[
data List \ a = Nil \mid Cons \ a \ (List \ a)
\]

This definition is recursive because the defined type constructor \(List\) occurs on the right hand side of the defining equation. We will discuss some other common examples of recursive data, and will study a recursive data type \(Region\) for image processing.
A binary tree is

- either empty,
- or a node with exactly two subtrees.
- Each node carries an integer as label.

```
data Tree = Null
  | Node Int Tree Tree
```

deriving (Eq, Read, Show)
Generating trees

Leafs, that is, trees with exactly one node:

\[
\text{leaf} :: \text{Int} \rightarrow \text{Tree} \\
\text{leaf } n = \text{Node } n \text{ Null Null}
\]

Example of a tree:

\[
\text{tree1} :: \text{Tree} \\
\text{tree1} = \text{Node } 2 (\text{leaf } 5) \\
\quad (\text{Node } 5 (\text{leaf } 7) (\text{leaf } 1))
\]

Balanced trees of depth \(n\) with label indicating depth of node:

\[
\text{balTree} :: \text{Int} \rightarrow \text{Tree} \\
\text{balTree } n = \text{if } n == 0 \\
\quad \text{then Null} \\
\quad \text{else let } t = \text{balTree } (n-1) \\
\quad \quad \text{in Node } n \ t \ t
\]
Showing trees

tree2Strings :: Tree -> [String]
tree2Strings Null = ["\n"]
tree2Strings (Node n t1 t2) =
    map (" " ++) (tree2Strings t1) ++ [show n] ++
    map (" " ++) (tree2Strings t2)

st :: Tree -> IO ()
st = putStrLn . concat . tree2Strings

st (balTree 3)

        1
       /
      2
     /
    1

3

        1
       /
      2
     /
    1
Testing occurrence in a tree

member :: Tree -> Int -> Bool
member Null _ = False
member (Node x l r) y =
    x == y || (member l y) || (member r y)

Exercise

Predict what happens when the following expressions are evaluated:

member (balTree 10) 1
member (balTree 10) 10
member (balTree 10) 11
member (balTree 100) 1
member (balTree 100) 100
member (balTree 100) 101
Lazy evaluation

When computing \( \text{member (balTree 100) 100} \) the following computation happens:

\[
\begin{align*}
\text{member (balTree 100) 100} & \quad \Rightarrow \\
\text{member (Node 100 (balTree 99) (balTree 99)) 100} & \quad \Rightarrow \\
100 == 100 \text{ || member t 99 || member t 99} & \quad \Rightarrow 100
\end{align*}
\]

where \( t = \text{balTree 99} \)

Because the operator \( (\text{||}) \) looks first at its left argument and Haskell uses lazy evaluation (or call-by-name) the recursive calls of member are not evaluated.

Therefore, the result is obtained instantly.

In languages like Java or C or Lisp which apply eager evaluation (or call-by-value) the computation would not terminate because an attempt would be made to compute the (huge) tree \( \text{balTree 100} \).
Traversing trees: prefix-, infix-, postfix-order

preord :: Tree -> [Int]
preord Null = []
preord (Node l x r) = [x] ++ preord l ++ preord r

inord :: Tree -> [Int]
inord Null = []
inord (Node l x r) = inord l ++ [x] ++ inord r

postord :: Tree -> [Int]
postord Null = []
postord (Node l x r) = postord l ++ postord r ++ [x]

inord (balTree 3) ⇒ [1,2,1,3,1,2,1]
postord (balTree 3) ⇒ [1,1,2,1,1,2,3]
Structural recursion

The definitions of `member`, `preord`, `inord`, `postord` are examples of

*structural recursion on trees*

The pattern is:

- **Base.** Define the function for the empty tree
- **Step.** Define the function for a composite tree using recursive calls to the left and right subtree.
Ordered trees

A tree Node $x \ l \ r$ is ordered if

- $\text{member } l \ y \text{ implies } y < x \text{ and}$
- $\text{member } r \ y \text{ implies } x < y \text{ and}$
- $l \text{ and } r \text{ are ordered.}$

$t = \text{Node } 9 \ (\text{Node } 5 \ (\text{leaf } 1) \ (\text{leaf } 7))$

\[
\begin{array}{c}
\text{1} \\
\text{5} \\
\text{7} \\
\text{9} \\
\text{13} \\
\text{15} \\
\text{29}
\end{array}
\]
Efficient membership test for ordered trees

\[
\text{member1} :: \text{Tree} \rightarrow \text{Int} \rightarrow \text{Bool}
\]
\[
\text{member1 } \text{Null } _ = \text{False}
\]
\[
\text{member1 } (\text{Node } x \ l \ r) \ y
\]
\[
| y < x \ = \ \text{member1 } l \ y
\]
\[
| y == x \ = \ True
\]
\[
| y > x \ = \ \text{member1 } r \ y
\]

*Main> \text{inord } t
[1,5,7,9,13,15,29]
*Main> \text{filter (member1 } t) [1..100]
[1,5,7,9,13,15,29]
*Main> \text{inord } (\text{Node } 50 \ \text{Null } t)
[50,1,5,7,9,13,15,29]
*Main> \text{member1 } (\text{Node } 50 \ \text{Null } t) \ 1
\text{False}
*Main> \text{member } (\text{Node } 50 \ \text{Null } t) \ 1
\text{True}
The complexity of member is $O(n)$ where $n$ is the size, that is, the number of nodes in the tree, since if a number does not occur in the tree this is known only after traversing the whole tree.

For balanced (or nearly balanced) ordered trees the complexity of member1 is $O(\log n)$, since only one path of the tree needs to be inspected, and each path has length $O(\log n)$. 
Order preserving insertion

Order preserving insertion of a number in a tree:

```
insert :: Tree -> Int -> Tree
insert Null y = leaf y
insert (Node x l r) y
  | y < x  = Node x (insert l y) r
  | y == x = Node x l r
  | y > x  = Node x l (insert r y)
```

The problem is that by repeatedly inserting elements one obtains an unbalanced tree (on which membership test is only linear).

*Main> st (Null 'insert' 1 'insert' 2 'insert' 3)

```
1
  2
    3
```
Order preserving deletion

\[
\text{delete} :: \text{Tree} \rightarrow \text{Int} \rightarrow \text{Tree} \\
\text{delete Null} \ y = \text{Null} \\
\text{delete (Node} \ x \ l \ r) \ y \\
| \ y < x = \text{Node} \ x \ (\text{delete} \ l \ y) \ r \\
| y == x = \text{join} \ l \ r \\
| y > x = \text{Node} \ x \ l \ (\text{delete} \ r \ y) \\
\]

\[
\text{join} :: \text{Tree} \rightarrow \text{Tree} \rightarrow \text{Tree} \\
\text{join t} \ \text{Null} = t \\
\text{join t s} = \text{Node} \ \text{leftmost} \ t \ s1 \ \text{where} \\
(\text{leftmost}, s1) = \text{splitTree} \ s \\
\text{splitTree} :: \text{Tree} \rightarrow (\text{Int}, \text{Tree}) \\
\text{splitTree} \ (\text{Node} \ x \ \text{Null} \ t) = (x, t) \\
\text{splitTree} \ (\text{Node} \ x \ l \ r) = (\text{leftmost}, \text{Node} \ x \ l1 \ r) \ \text{where} \\
(\text{leftmost}, l1) = \text{splitTree} \ l \\
\]

\text{join combines two trees} \ l, r \ \text{in an order preserving way provided all labels in} \ l \ \text{are smaller than those in} \ r.
Order preserving deletion

*Main> st t

1
5
7
9
13
15
29

*Main> st (delete t 9)

1
5
7
13
15
29
Let us resume our case study on image processing.

We lift the operations of intersection, union, etc. to an abstract level by introducing them as *constructors* in a recursive algebraic data type Region.

data Region = Sh Shape
                | Bm Bitmap
                | Inter Region Region
                | Union Region Region
                | Invert Region
                | Rotate Float Region
                | Empty
The meaning of a region

By a straightforward structural recursion we can transform a region into a bitmap. Basically, we interpret every constructor for regions by the corresponding operation on bitmaps.

\[
\text{inRegion} :: \text{Region} \rightarrow \text{Bitmap}
\]

\[
\text{inRegion} \; \text{re} = \text{case} \; \text{re} \; \text{of}
\]

\[
\begin{align*}
\text{Sh} \; \text{sh} & \rightarrow \text{inShape} \; \text{sh} \\
\text{Inter} \; \text{re1} \; \text{re2} & \rightarrow \text{interBm} \; (\text{inRegion} \; \text{re1}) \; (\text{inRegion} \; \text{re2}) \\
\text{Union} \; \text{re1} \; \text{re2} & \rightarrow \text{unionBm} \; (\text{inRegion} \; \text{re1}) \; (\text{inRegion} \; \text{re2}) \\
\text{Invert} \; \text{re} & \rightarrow \text{invertBm} \; (\text{inRegion} \; \text{re}) \\
\text{Rotate} \; \alpha \; \text{re} & \rightarrow \text{rotateBm} \; \alpha \; (\text{inRegion} \; \text{re}) \\
\text{Empty} & \rightarrow \text{const} \; \text{False}
\end{align*}
\]

\[
\text{instance Show Region where}
\]

\[
\text{show} = \text{showPicture} \; . \; \text{takePicture} \; \text{set0} \; . \; \text{inRegion}
\]
The union of a list of regions - foldr

unionsReg :: [Region] -> Region
unionsReg [] = Empty
unionsReg (re : res) = Union re (unionsReg res)

A more concise definition is

unionsReg = foldr Union Empty

which uses the predefined higher-order function foldr ("fold-right").

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
Examples of regions

ring :: Region
ring = Inter (Sh (Circle (0,0) 1.5))
    (Invert (Sh (Circle (0,0) 1)))

tree = Union (Sh (Rectangle (-0.2,-1) (0.2,0.6)))
    (Sh (Circle (0,1) 0.5))

car :: Region
car = (unionsReg . map Sh)
    [Rectangle (-1,-1) (1,-0.5),
     Rectangle (-0.4,-0.5) (0.6,-0.1),
     Circle (-0.6,-1) 0.2,
     Circle (0.6,-1) 0.2]
Summary

- Using the `data` keyword the user can introduce new recursive data types.

- Common recursive data types are lists and trees.

- Functions on recursive data types can be defined by *structural recursion*. For example, recursion on lists, trees.

- Maintaining special properties of recursive data can lead to efficient code. Example: membership for ordered trees.
Input/Output in functional languages

Problem: Functions with side effects are not referentially transparent (that is, not all parameters the function depends on are visible).
  - for example readString :: () -> String

Solution:
Side effects are marked by IO — actions
  - Actions can be combined with actions only
  - “once IO, always IO”
The type $\text{IO } a$

- $\text{IO } a$ is the type of *actions* returning a value of type $a$ when executed.
- The value returned by an action can be used in the next action.
- Two actions are combined by the operator
  \[ (\gg\gg)= ) :: \text{IO } a \to (a \to \text{IO } b) \to \text{IO } b \]
Actions as an abstract data type

type IO a

(>>=) :: IO a
    -> (a -> IO b)
    -> IO b

return :: a -> IO a

Environment
Combining actions

If we are given

\[
\begin{align*}
\text{action1} & : \text{IO} \ a \\
\text{action2} & : \text{a} \rightarrow \text{IO} \ b
\end{align*}
\]

then the action

\[
\text{action1} >>= \text{action2} : \text{IO} \ b
\]

first does \text{action1} and then \text{action2} \ x where \ x \ is \ the \ value \ returned \ by \ \text{action1}. \]
Some predefined actions

- Reading a line from stdin (standard input):
  
  ```haskell
  getLine :: IO String
  ```

- Writing string to stdout (standard output):
  
  ```haskell
  putStr :: String -> IO ()
  ```

- Writing string to stdout and inserting a new line:
  
  ```haskell
  putStrLn :: String -> IO ()
  ```
A simple example

- echo :: IO ()
  echo = getLine >>= putStrLn >>= \_ -> echo

- Reversed echo:
  ohce :: IO ()
  ohce = getLine >>= putStrLn . reverse >> ohce

where (>>) is predefined:

(>>) :: IO a -> IO b -> IO b
f >> g = f >>= \_ -> g

The meaning of f >> g is

“First do f, then g, throwing away the value returned by f”.
The do-notation

- Syntactic sugar for IO:

```haskell
echo =
    getLine
    >>= putStrLn
    >> echo

≡

echo =
    do s <- getLine
        putStrLn s
        echo
```

- On the right-hand side `>>=` and `>>` are implicit.

- Layout: Be careful with indentation in a do construction.
Communication with files

- In Prelude predefined:
  - Writing into files (override, append):
    ```haskell
type FilePath = String
writeFile :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```
  - Reading a file (lazy):
    ```haskell
readFile :: FilePath -> IO String
```
Example: Counting characters, words, lines

```haskell
wc :: String -> IO ()
wcc file =
  do c <- readFile file
     putStrLn (show (length (lines c)) ++ " lines ")
     putStrLn (show (length (words c)) ++ " words, and ")
     putStrLn (show (length c) ++ " characters. ")
```
Actions as values

- Actions are normal values.
- This makes the definition of control structures very easy.
- Better than any imperative language.
Predefined control structures

▶ Sequencing a list of actions:

```haskell
sequence :: [IO a] -> IO [a]
sequence (c:cs) = do x <- c
                      xs <- sequence cs
                      return (x:xs)
```

`sequence [act1,...,actn]` is the action that does the actions `act1, ..., actn` (in that order) and returns the list of results.

▶ Special case: `[()]` as `()`

```haskell
sequence_ :: [IO ()] -> IO ()
```
Example of actions as values: Random numbers.
Module Random, class Random

class Random a where
    randomRIO :: (a,a) -> IO a
    randomIO  :: IO a

For example, randomRIO (0,36) is the action that returns a random number between 0 and 36.

Random further contains
  - Random generators for pseudo random numbers
  - Infinite lists of random numbers
Example: Random words

- Repeating an action a random number of times:
  
  ```haskell```
  atmost :: Int -> IO a -> IO [a]
  atmost most a =
      do l <- randomRIO (1, most)
          sequence (replicate l a)
  ```haskell```

- Random string:
  
  ```haskell```
  randomStr :: IO String
  randomStr = atmost 10 (randomRIO ('a','z'))
Roulette

- Start with 100 pounds.
- Choose a number between 1 and 36.
- Say how many rounds you want to play.
- Each round costs 1 pound.
- In each round a number between 0 and 36 is randomly played.
- If it is your number, you win 36 pounds.
- The game stops if you run out of money.
**until and until IO**

We use a variant of `until` that accepts a `next` function with side effects:

\[
\text{untilIO} :: (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow \text{IO } a) \rightarrow a \rightarrow \text{IO } a
\]

\[
\text{untilIO } \text{stop } \text{next } x =
\]

\[
\begin{align*}
\quad & \quad \text{if stop } x \\
\quad & \quad \text{then return } x \\
\quad & \quad \text{else do } x1 \leftarrow \text{next } x \\
\quad & \quad \text{untilIO } \text{stop } \text{next } x1
\end{align*}
\]

Compare this with the predefined function

\[
\text{until} :: (a \rightarrow \text{Bool}) \rightarrow (a \rightarrow a) \rightarrow a \rightarrow a
\]

\[
\text{until } \text{stop } \text{next } x =
\]

\[
\begin{align*}
\quad & \quad \text{if stop } x \\
\quad & \quad \text{then } x \\
\quad & \quad \text{else let } x1 = \text{next } x \\
\quad & \quad \text{in until } \text{stop } \text{next } x1
\end{align*}
\]
Implementing Roulette

import Random (randomRIO)

roulette :: IO ()
roulette =
do putStr "You have 100 pounds. Which number do you choose? "
n <- getLine
putStr "How many rounds do you want to play maximally? "
r <- getLine
let mine = read n :: Int
maxRounds = read r :: Int
stop (k,money) = k >= maxRounds || money <= 0
next (k,money) = do x <- randomRIO (0,36)
    let win = if x == mine then 36 else 0
    return (k+1, money - 1 + win)
(rounds,endMoney) <- untilIO stop next (0,100)
putStr ("\nYou played " ++ (show rounds) ++ " rounds " ++
    "and have now " ++ (show endMoney) ++ " pounds\n")
Summary

- Input/Output in Haskell by *Actions*
  - Actions (type `IO a`) are functions with side effects
  - Combination of actions by
    \[
    (\gg\gg) :: IO a \to (a \to IO b) \to IO b
    \]
    \[
    \text{return} :: a \to IO a
    \]
  - do-Notation

- Various functions from the standard library:
  - Prelude: `getLine`, `putStr`, `putStrLn`, `writeFile`, `readFile`
  - Modules: `IO`, `Random`, `Random`
Summary (ctd.)

- Actions are normal values.

- Useful combinators for actions:
  - `sequence :: [IO a] -> IO [a]`
  - `sequence_ :: [IO ()] -> IO ()`
  - `untilIO :: (a -> Bool) -> (a -> IO a) -> a -> IO a`
Exercises

- Write an interactive translation program, say from English to French and vice versa. Use a dictionary of the form [(apple,pomme),(street,rue),(sun,soleil),...].
- Modify the program `saveCW` such that the statistical data are written to the file as well.
- Modify the program `roulette` such that after each 100 rounds the player is informed about their current account and given the choice whether or not to continue.
- Write a program that allows the user to interactively read a tree from a file, display the tree, insert a node, delete a node, save the tree into a specified file.