universität innsbruck

Functional Programming | WS 2021 | LVA 703025

Exercise Sheet 1, 5 points | Deadline: Wednesday, October 13, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can use a template .hs-file that is provided on the proseminar page.
- Upload your modified .hs-file of Exercise 2 in OLAT.
- Your .hs-file should be compilable with ghci.

Exercise 1 *Haskell setup, no points*

Setup a working Haskell environment on your computer and get familiar with `ghci`. To do this follow these steps:

1. Install Haskell, e.g., via `ghcup`.[1]
2. Run `ghci` in a terminal and evaluate the expression `(5 + 2) * 3`.
3. Find and install a suitable text editor for your system to write and edit .hs files.[2] You can try one of the following free editors:
   - Atom[3] (Windows, macOS, Linux)
   - Notepad++[4] (Windows)
   - Gedit[5] (Windows, macOS, Linux)
4. Copy or enter the following code in your text editor and save it to a file called `myProgram.hs`. Be aware to use standard double-quotes (`"`), but neither two single-quotes (`''`) nor fancy-looking double-quotes (“ or ”).
   ```
   hello :: String -> String
   hello xs = "Hello " ++ xs
   ```
5. Load the file in `ghci` with the command `ghci myProgram.hs`
6. Evaluate the expression `hello "World"`
7. Make yourself familiar with `ghci`, in particular try the following commands:
   - `:?` – help
   - `:load name.hs` or `:l name.hs` – load Haskell script `name.hs`
   - `:reload` or `:r` – reload current Haskell script
   - `:edit` or `:e` – edit current Haskell script
   - `:set editor someEditor` – set `someEditor` as preferred editor

   Further investigate what happens if you type `h` and then the tabulator key, or `hel` and then the tabulator key.

You can find links to introductory material about `ghci`, the command line, etc. on the lecture homepage.[6]

[1] https://www.haskell.org/ghcup/
[2] Word processors like Microsoft Word, Apple pages,. . . are not text editors.
[3] https://atom.io/
[4] https://notepad-plus-plus.org/
[5] https://wiki.gnome.org/Apps/Gedit
[6] http://cl-informatik.uibk.ac.at/teaching/ws21/fp/ghc_setup.php

Exercise 2 **5 p.**

1. Define a function `milesToKilometers m = ...` to convert miles into kilometers. (1 point)
2. Define a function `volume r = ...` to compute the volume of a sphere with radius `r`. (1 point)
3. Define a function `average x y = ...` that computes the average of two numbers `x` and `y`. (1 point)
4. Is `average (average x y) z` the average of three numbers `x`, `y` and `z`? (1 point)
5. Define a function `averageVolume r1 r2 = ...` that computes the average volume of two spheres having radius `r1` and `r2`, respectively. (1 point)

universität innsbruck

Functional Programming	WS 2021	LVA 703025

Exercise Sheet 2, 10 points — Deadline: Wednesday, October 20, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- Upload your .hs-file of Exercise 3 in OLAT.
- Your .hs-file should be compilable with ghci.

Exercise 1 *Typing* — 4 p.

Given the definition

```
plus1 x = x + 1
```

which of the following typing judgments are valid? Justify your answers.

1. `0 :: Bool` (1 point)
2. `head "test" :: Char` (1 point)
3. `'hello' :: String` (1 point)
4. `plus1 :: Integer -> Integer` (1 point)

Exercise 2 *Parsing expressions* — 3 p.

Draw the abstract syntax trees of the following expressions:

1. `7 * (4 - x) + 5 / 3` (1 point)
2. `(x < 10) || (y > 15)` (1 point)
3. `average 5 10 * square 2 + 10` (1 point)

Remark: function applications (e.g., `square 7`) bind stronger than operator applications (e.g., `3 * 4`).

Exercise 3 *Modelling* — 3 p.

In graphical user interfaces (GUIs) a *menu* typically consists of *items* and submenus. One specific application of such menus is website navigation, where items would consist of a label (the text to click on) and a link (the URL of the website to navigate to when the item is clicked).

1. Give a Haskell datatype definition to model items for website navigation as described above. Moreover, define constants that represent the items OLAT (`https://lms.uibk.ac.at`) and FP (`http://cl-informatik.uibk.ac.at/teaching/ws21/fp`) (1 point)
2. Give a Haskell datatype definition to model menus that may contain up to two items. Moreover, define a constant that represents a menu containing the items for OLAT and FP from above. (1 point)
3. Change your definition from the previous exercise such that a menu contain an arbitrary number of items. Moreover, define a constant that represents a menu with at least three items and also represent this constant as a tree as shown on the slides of week 2, page 23. (1 point)

Functional Programming	WS 2021	LVA 703025
Exercise Sheet 3, 10 points		Deadline: Wednesday, October 27, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can use a template .hs-file that is provided on the proseminar page.
- Upload your .hs-file(s) of Exercises 2 and 3 in OLAT.
- Your .hs-file should be compilable with ghci.

Exercise 1 *Pattern Matching* **2 p.**

Consider the following datatype definitions:

```haskell
data Subject = CS | Math | Physics | Biology
data Programme =
    Bachelor Subject
  | Master Subject
  | Teaching Subject Subject -- teachers need two subjects
data Student = Student
  String -- name
  Integer -- matriculation number
  Bool -- active inscription
  Programme
```

Determine which of the expressions 1.-3. match the patterns in (i) and (ii). For each match give the corresponding substitution. (2 points)

1. `Student "Jane Doe" 243781 True (Teaching Math Physics)`
2. `Student "Max Meyer" 221341 False (Teaching CS Math)`
3. `Student "Mary Smith" 234145 False (Master CS)`

(i) `Student name n _ (Teaching Math _)`

(ii) `Student name n False p@(Master _)`

Exercise 2 *Function Definitions* **3 p.**

1. Define a function `disj :: Bool -> Bool -> Bool` for computing the disjunction of two Booleans. (1 point)
2. Define a function `sumList :: List -> Integer` that takes a list of integers (as defined in the lecture) and returns the sum of its elements. The sum over an empty list should be 0. (1 point)
3. Define a function `double2nd :: List -> List` that doubles every second element in a given list of integers, i.e., $[1, 7, 9, 3] \leadsto [1, 14, 9, 6]$. (1 point)

Exercise 3 *A Recursive Function* **5 p.**

In this exercise, we will extend the `Expr` datatype from the lecture with variables:

```
data Expr =
    Number Integer
  | Var String
  | Plus Expr Expr
  | Negate Expr
```

We will also need the following datatype to store variable assignments:

```
data Assignment = EmptyA | Assign String Integer Assignment
```

Here, the `EmptyA` constructor corresponds to an empty assignment in which all variables have value 0. The `Assign` constructor takes an assignment and changes the value of one variable to the given integer (see examples below).

1. Write a function `ite :: Bool -> Integer -> Integer -> Integer` such that `ite b x y` returns `x` if `b` is true and `y` otherwise. Use pattern matching on Booleans only. ("`ite`" stands for "if–then–else") (1 point)
2. Write a function `lookupA :: Assignment -> String -> Integer` that returns the value corresponding to the given variable in the given assignment. (1 point)

   Example: Let `myAssn = Assign "x" 1 (Assign "x" 2 (Assign "y" 3 EmptyA))`. Then:

   ```
   lookupA myAssn "x" == 1
   lookupA myAssn "y" == 3
   lookupA myAssn "z" == 0
   ```
3. Write a function `eval :: Assignment -> Expr -> Integer` that evaluates the given arithmetic expression under the given variable assignment. (2 points)

   Example: Let `myAssn` be as before. Then:

   ```
   eval myAssn (Plus (Negate (Var "y")) (Number 45)) == 42  -- corresponds to (-y) + 45
   ```
4. In order to store auxiliary results and avoid computing the same things twice, extend the `Expr` type with a "**let** $x = e_1$ **in** e_2" construct, i.e. the result of e_1 is assigned to the variable x when evaluating e_2, the result of which is then returned. Extend your "`eval`" function accordingly as well. (1 point)

   Example:
   You should be able to write an expression that encapsulates something like this:

   $$\textbf{let } x = 2 + 3 \textbf{ in } x + x$$

   How you represent this in your datatype is up to you.

universität innsbruck

Functional Programming | WS 2021 | LVA 703025

Exercise Sheet 4, 10 points | Deadline: Wednesday, November 3, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can use a template .hs-file that is provided on the proseminar page.
- Upload your .hs-file(s) of Exercises 1 and 2 in OLAT.
- Your .hs-file should be compilable with ghci.

Exercise 1 *Nested Lists and Either* **5 p.**

1. Study the slides of week 4, pages 19 & 20 to understand the consequences of the definition of the predefined string type.

```
type String = [Char]
(++) :: [a] -> [a] -> [a]    -- and not: String -> String -> String
head :: [a] -> a             -- and not: String -> Char
```

   Given a function `concat :: [[a]] -> [a]`, briefly explain the type of the following six Haskell expressions or give a reason why these expressions result in a type error.

```
e1 = concat [1 :: Int, 2, 3]
e2 = concat ["one", "two", "three"]
e3 = concat [[1 :: Int, 2], [], [3]]
e4 = concat [["one", "two"], [], ["three"]]
e5 = concat e3
e6 = concat e4
```

(1 point)

2. Define a function `suffixes` that computes the list of all suffixes of a list. Particularly, the following identities should hold:

```
suffixes [1, 2, 3] = [[1,2,3], [2,3], [3], []]
suffixes "hello"   = ["hello", "ello", "llo", "lo", "o", ""]
```

   Hint: structural recursion suffices.

(1 point)

3. Define a function `prefixes` that computes the list of all prefixes of a list. Particularly, the following identities should hold:

```
prefixes [1, 2, 3] = [[1,2,3], [1,2], [1], []]
prefixes "hello"   = ["hello", "hell", "hel", "he", "h", ""]
```

   Hint: you might need an auxiliary function; structural recursion is not recommended for `prefixes`.

(2 points)

4. Utilize the `Either` type to create a menu that generates the list of prefixes, suffixes or a meaningful error message depending on its input. Particularly, the following identities should hold:

```
menu 'p' [1,2,3]  = Right [[1,2,3],[1,2],[1],[]]
menu 'p' "hello"  = Right ["hello","hell","hel","he","h",""]
menu 's' [1,2,3]  = Right [[1,2,3],[2,3],[3],[]]
menu 's' "hello"  = Right ["hello","ello","llo","lo","o",""]
menu 'c' "hello"  = Left  "(c) is not supported, use (p)refix or (s)uffix"
```

(1 point)

Exercise 2 *Polymorphic Expressions* **5 p.**

1. Define a polymorphic datatype to represent expressions involving addition, multiplication and numbers. In particular `expr1` and `expr2` should be accepted.

   ```
   expr1 = Times (Plus (Number (5.2 :: Double)) (Number 4)) (Number 2)
   expr2 = Plus (Number (2 :: Int)) (Times (Number 3) (Number 4))
   expr3 = Times (Number "hello") (Number "world")
   ```

   Is `expr3` type correct as well? Provide a brief explanation. (1 point)

2. Write a polymorphic function `numbers` that given an expression constructs a list of numbers that occur in the expression. For example `numbers expr1 = [5.2,4,2]` and `numbers expr2 = [2,3,4]`.

   Also provide a type for your function that is as general as possible. (1 point)

3. Write a polymorphic function `eval` to evaluate an expression. For example `eval expr1 = 18.4` and `eval expr2 = 14`.

   Also provide a type for your function that is as general as possible. (1 point)

4. Write a polymorphic function `exprToString` that converts an expression into a string that represents the expression. The string should insert parentheses only if they are required. For example:

   - `exprToString expr1 = "(5.2 + 4.0) * 2.0"`
   - `exprToString expr2 = "2 + 3 * 4"`

   Also provide a type for your function that is as general as possible. (2 points)

Exercise Sheet 5, 10 points Deadline: Wednesday, November 10, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_05.hs` provided on the proseminar page.
- Upload your .hs-file(s) of Exercises 1 and 2 in OLAT.
- Your .hs-file(s) should be compilable with ghci.

Exercise 1 *Lists and Tuples* **5 p.**

1. Implement a Haskell function `mergeLists` that takes two lists and combines them into a list of pairs. If the lists do not have the same length, the rest of the longer list is ignored. Make sure that the type of your function is as general as possible. (1 point)

   Example: `mergeLists [1,2,3,4] ['a','b','c'] == [(1,'a'),(2,'b'),(3,'c')]`

2. Implement a Haskell function `calculateAge :: (Int, Int, Int) -> Int` that, given a birthday as a triple of day, month and year, computes the current age in years (as of November 10, 2021). (1 point)

   Example: `calculateAge (10, 11, 2021) == 0`

3. Implement a Haskell function

   ```
   convertDatesToAges :: [(String, (Int, Int, Int))] ->  [(String, Int)]
   ```

   that converts a list of names and birthdays into a list of names and ages. That is, it takes a list of pairs with names and dates and computes a list of pairs with names and ages. (1 point)

4. Implement a Haskell function `getOtherPairValue` that takes a pair of name and age as well as a second argument that can be either a name *or* an age and returns the part of the input that wasn't the second argument. If the second argument does not match either element of the input-pair the function should return nothing. (2 points)

   Example: When invoking `getOtherPairValue` on the pair `("Hans", 45)` together with the second argument `45` (wrapped in an appropriate type), the result should be `"Hans"` (wrapped in an appropriate type).

Hint: You might find the `ite` (if-then-else) function of sheet 3 useful for some of the above exercises.

```
ite :: Bool -> a -> a -> a
ite True x y = x
ite False x y = y
```

Exercise 2 *Equational Reasoning, Lists, and Tuples* **5 p.**

Consider the following Haskell functions

```
addPair (x, y)    =  x + y
addList []        =  []
addList (x : xs) =  addPair x : addList xs
```

1. Write down the types of the functions above and explain how you were able to derive them. (1 point)

2. Evaluate the result of the following expression step-wise (that is, only using one of the above equations at a time) by hand. (2 points)

   ```
   addList ((1,2):(2,1):[])
   ```

   Remark: For clarification here is an example of another step-wise evaluation (this process is called *equational reasoning*; the definition of `lastElem` can be found on page 3 of the slides of week 4):

   ```
   lastElem (Cons 1 (Cons 2 Empty)) = lastElem (Cons 2 Empty)
                                    = 2
   ```

3. Implement a Haskell function `fstList :: [(a, b)] -> [a]` that collects all the first elements of a list of pairs. (1 point)

   Examples:
   ```
   fstList [(1,'a'),(2,'b'),(3,'c')] == [1,2,3]
   fstList [("hello","world")] == ["hello"]
   fstList [] == []
   ```

4. Implement a Haskell function `lengthSumMax :: (Num a, Ord a) => [a] -> (Int, a, a)` that, given a list of non-negative numbers, computes its length, the sum of all its elements and the maximum of all its elements and returns those three values as a triple. (1 point)

   Examples:
   ```
   lengthSumMax [] == (0,0,0)
   lengthSumMax [0,1,0,2,0] == (5,3,2)
   ```

universität innsbruck

Functional Programming | WS 2021 | LVA 703025

Exercise Sheet 6, 10 points | Deadline: Wednesday, November 17, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_06.hs` provided on the proseminar page.
- Your .hs-file(s) should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Functions on Numbers* **4 p.**

1. Implement a Haskell function `dividesRange :: Integer -> Integer -> Integer -> Bool` that checks whether there is any divisor of a number within a given range: `dividesRange n l u` should be true iff there is some `x` such that `l` ≤ `x` ≤ `u` and `x` divides `n`. (1 point)

   Example: `dividesRange 629 15 25 == True` since $15 \leq 17 \leq 25$ and 17 divides 629.

   Hint: You can use the built-in functions `div` or `mod` for checking divisibility of two numbers: `div x y` and `mod x y` compute the quotient and the remainder of the integral division of `x` by `y`, respectively. E.g., `div 25 4 = 6` and `mod 25 4 = 1`, since $25 = 6 \cdot 4 + 1$.

2. Implement a Haskell function `prime :: Integer -> Bool` to determine whether a number is prime. Recall: n is a prime number if $n \geq 2$ and n has exactly two divisors. (1 point)

   Example: `prime 7793 == True` and `prime 7797 == False`.

3. Implement a Haskell function `generatePrime :: Integer -> Integer` which takes a number `d` as input and computes a prime number with at least `d` digits. (1 point)

   Valid examples: `generatePrime 4 == 1009` and `generatePrime 8 == 10000019`.

4. How far can you increase `d` such that `generatePrime d` is computed within 1 minute? If this value is below 10, then improve your algorithm `prime`. (1 point)

   Hint: Implement a square root function directly on integers, i.e., without using `sqrt :: Double -> Double`. It does not matter if you implement $\lfloor\sqrt{n}\rfloor$ or $\lceil\sqrt{n}\rceil$. For instance, the integer square root of 27 can either be 5 or 6.

Exercise 2 *Heron's method* **2 p.**

Heron's method is an ancient (but very efficient) method to approximate the square root of a given non-negative real number x. It works like this: We recursively define the following sequence of numbers:

$$y_0 = x \qquad\qquad y_{n+1} = \begin{cases} 0 & \text{if } y_n = 0 \\ \frac{1}{2}(y_n + \frac{x}{y_n}) & \text{otherwise} \end{cases}$$

Mathematically, this sequence converges monotonically to $\sqrt{x}$ but never actually reaches it (unless $x = 0$ or $x = 1$), giving successively better and better approximations to $\sqrt{x}$.

However, due to the finite precision of the `Double` type, doing this computation in Haskell, you will always find that at some point $y_{n+1} == y_n$.

Your task is to write a function `heron :: Double -> [Double]` that outputs the sequence of numbers $[y_0, \dots, y_n]$, where n is the smallest number such that $y_{n+1} == y_n$. (2 points)

Examples:

```
heron 0 == [0.0]
heron 1 == [1.0]
heron 2 == [2.0, 1.5, 1.4166666666666665, 1.4142156862745097,
            1.4142135623746899, 1.414213562373095]
```

Exercise 3 *Fibonacci numbers* **4 p.**

The Fibonacci numbers $(a_n)_{n\in\mathbb{N}} = 0, 1, 1, 2, 3, 5, 8, \dots$ are defined by the recurrence

$$a_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ a_{n-1} + a_{n-2} & \text{otherwise} \end{cases}$$

They can also be computed more efficiently by the following recurrence:

$$a_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \text{ or } n = 2 \\ a^2_{\lfloor \frac{n}{2} \rfloor} + a^2_{\lfloor \frac{n}{2} \rfloor + 1} & \text{if } n \text{ is odd} \\ (2a_{\lfloor \frac{n}{2} \rfloor + 1} - a_{\lfloor \frac{n}{2} \rfloor})a_{\lfloor \frac{n}{2} \rfloor} & \text{if } n \text{ is even} \end{cases}$$

You will implement the computation of the a_n given a non-negative integer n as an input in three different ways:

1. Write a function `fib :: Integer -> Integer` that computes a_n using the naïve recurrence $a_{n+2} = a_{n+1} + a_n$ and a function `fib' :: Integer -> Integer` that does the same using the more complicated recurrence given above.

   Test your functions on increasingly large values and see how much time they take.
   Explain the results. 2 points

   Hint:

   - If you run the command `:set +s` in GHCi, it will print how long each evaluation took.[1]
   - If an evaluation takes too long, you can abort it using the key combination `Ctrl + C`.
   - Recall that in Haskell, $\lfloor \frac{n}{2} \rfloor$ is written as `div n 2`. Also note that the following pre-defined functions exist: `even :: Integer -> Bool` and `odd :: Integer -> Bool`

2. Write a function `fibFast :: Integer -> Integer` that does the same as `fib'` but internally remembers all values that have already been computed in a lookup table. Again check how long it takes on increasingly large inputs. 2 points

   Hint:

   - You will need an auxiliary function

     `fibFastAux :: Integer -> [(Integer, Integer)] -> (Integer, [(Integer, Integer)])`

     that takes a number n and a lookup table (consisting of pairs (i, a_i)) and returns both the result a_n and a (possibly bigger) lookup table. If the pair for n is already in the table, the stored value of a_n should be returned – otherwise the recurrence should be used to recursively compute the value of a_n, and the new pair (n, a_n) is then stored in the table.
   - The pre-defined function `lookup :: Eq a => a -> [(a, b)] -> Maybe b` will be useful to lookup values in the table.
   - The numbers involved grow *very* fast, so even the printing takes a lot of time. For more consistent results, try showing the number of digits of the result instead of the actual result, e.g. `length (show (fib' 10000))` or `length (show (fibFast 10000))`.

[1] Note that benchmarking functions in GHCi like this is not particularly accurate for a number of reasons: the code is not compiled but only interpreted, and many optimisations that GHC normally does are not performed. But for the scope of this exercise, this is fine.

universität innsbruck

Exercise Sheet 7, 10 points Deadline: Wednesday, November 24, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_07.hs` provided on the proseminar page.
- Your .hs-file(s) should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Rational Numbers* **5 p.**

Implement rational numbers in Haskell. Here, rational numbers are represented by two integers, the numerator and the denominator. For instance the rational number $\frac{-3}{5}$ is represented as `Rat (-3) 5` when using the following data type definition.

```haskell
data Rat = Rat Integer Integer
```

1. Implement a normalisation function for rational numbers, so that all of `Rat 2 4`, `Rat (-1) (-2)` and `Rat 1 2` get transformed into the same internal representation. (1 point)

   Hint: the Prelude already contains a function `gcd`.

2. Make `Rat` an instance of `Eq` and `Ord`. Of course, `Rat 2 4 == Rat 1 2` should be valid. (1 point)

3. Make `Rat` an instance of `Show`. Make sure that `show r1 == show r2` whenever `r1 == r2` for two rational numbers `r1` and `r2`. In particular, `show (Rat 1 2) == show (Rat 2 4)` should evaluate to true. Moreover, integers should be represented without division operator. (1 point)

   Examples: `show (Rat (-4) (-1)) == "4"` and both `show (Rat (-3) 2)` and `show (Rat 3 (-2))` result in `"-3/2"`.

4. Make `Rat` an instance of `Num`. See `https://hackage.haskell.org/package/base-4.16.0.0/docs/Prelude.html#t:Num` for a detailed description of this type class. (2 points)

Exercise 2 *Class and Data* **5 p.**

1. Create a new data type `Ingredient`, which should hold the name, the amount and the unit of the ingredient. For the unit there exists ML (milliliters), G (gram) and PC (pieces). Also make `Ingredient` an instance of `Show` to show an ingredient like this `"200 ML of Milk"`. (1 point)

2. Create a new `class Price` with a function `getPrice`, which should return a price in euro as a `Float`. Create now an instance for your `Ingredient` data type. The price of an ingredient is defined independently of the name of the ingredient as follows: 1 ML costs 0.12 cent, 1 PC costs 75 cent and 1 G costs 0.095 cent.

   Modify the `Show` instance of `Ingredient` of the previous part in a way that additionally the price of the ingredient is displayed in Euro, e.g., `"200 ML of Milk, cost: 0.24 EUR"` (2 points)

3. Create a data type `Recipe`, where you can store an arbitrary amount of ingredients. Make `Recipe` an instance of `Price` which calculates the total costs of the ingredients. Moreover, make `Recipe` an instance of `Show` such that a recipe is displayed in a form like this: "200.0 ML of Milk, cost: 0.24 EUR - 200.0 G of Sugar, cost: 0.19 EUR - 3.0 PC of Egg, cost: 2.25 EUR - Price of the Recipe: 2.68 EUR". (2 points)

Functional Programming — WS 2021 — LVA 703025

Exercise Sheet 8, 10 points — Deadline: Wednesday, December 1, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_08.hs` provided on the proseminar page.
- Your .hs-file(s) should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Partial Application and Sections* — 3 p.

Consider the following functions:

```
div1 = (/)
div2 = (2 /)
div3 = (/ 2)
eqTuple f = (\(x, y) -> f x == f y)
eqTuple' f (x, y) = f x == f y
```

1. Explain what these functions do and give the most general type signature for each function (do not use GHCi to find the type signatures). Give an example that shows the difference between `div2` and `div3` and explain why they are different. (1 point)
2. We say that a Haskell function `f` is equal to a Haskell function `g`, whenever `f x1 .. xN = g x1 .. xN` for all inputs `x1`, . . . , `xN`. Based on this definition, are the functions `eqTuple` and `eqTuple'` equal? Justify your answer. (1 point)
3. Which of the functions `foo1 x y = y / x` and `foo2 x y = (\u v -> v / u) y x` are equal to `div1` from above? Justify your answer. (1 point)

Exercise 2 *Higher-Order Functions and Lambdas* — 5 p.

1. Implement a recursive higher-order function `fan :: (a -> Bool) -> [a] -> [[a]]` that takes a predicate[1] and a list, and "fans out" the list into a list of sublists such that for each sublist either *each* element satisfies the predicate or *none* does. Moreover, your implementation should satisfy the equation

   `concat (fan p xs) == xs` (3 points)

   that is, concatenating the result of a call to `fan` results in the original list.

   Examples:
   ```
   fan undefined [] == []
   fan even [1..5] == [[1],[2],[3],[4],[5]]
   fan (== 'T') "This is a Test" == ["T","his is a ","T","est"]
   ```

2. Use `fan` from exercise 1 together with some lambda expression to implement a function

   ```
   splitOnNumbers = fan (\x -> ... x ...)
   ```

   that splits a given text into numbers and non-numbers. (1 point)

   Example: `splitOnNumbers "8 out of 10 cats" == ["8"," out of ","10"," cats"]`

   Hint: Recall that `Char` is an instance of `Ord`.

[1] Functions that return `Bool`s are sometimes called *predicates*, since they "decide" whether their input satisfies some property. For example `even` from the `Prelude` is a predicate on integers.

3. Use `fan` from exercise 1 to implement a function `splitBy :: (a -> Bool) -> [a] -> [[a]]` that splits a given list into sublists such that only parts remain that do not satisfy the given predicate. (1 point)

   Example: `splitBy (== '\n') "Just\nsome\nlines\n" == ["Just","some","lines"]`

Hint: If you did not manage to implement `fan` of exercise 1, you may use the following implementation in exercises 2 and 3:

```
import Data.List
fan p = groupBy (\x y -> p x == p y)
```

Exercise 3 *The* `foldr` *Function* 2 p.

Implement the following functions using `foldr` instead of recursion. In the process, you may find lambda expressions useful.

1. Consider a function that converts a list of digits (represented as `Integer`s) into an `Integer`:

   ```
   dig2int :: [Integer] -> Integer
   dig2int [] = 0
   dig2int (x:xs) = x + 10 * dig2int xs
   ```

   Examples: `dig2int [2,1,5] == 512`

   Implement a variant `dig2intFold` of `dig2int` using `foldr`. (1 point)

2. Consider a function `suffs` that computes all suffixes of a list, from longest to shortest:

   ```
   suffs :: [a] -> [[a]]
   suffs [] = [[]]
   suffs (y @ (_ : xs)) = y : suffs xs
   ```

   Examples:

   ```
   suffs [1,2] = [[1,2], [2], []]
   suffs "hello" = ["hello", "ello", "llo", "lo", "o", ""]
   ```

   Implement a variant `suffsFold` of `suffs` using `foldr`. (1 point)

universität innsbruck

Functional Programming — WS 2021 — LVA 703025

Exercise Sheet 9, 10 points — Deadline: Wednesday, December 15, 2021, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_09.hs` provided on the proseminar page.
- Your .hs-file(s) should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *The Caesar cipher* — 6 p.

The well known Caesar cipher[1] encodes a string by shifting each character n times (for some n).

1. Write a function `shift :: Int -> Char -> Char` that applies a shift factor to a lower-case letter (letters `'a'` – `'z'`). Other characters should be ignored by your function.

   Hint: `fromEnum` and `toEnum` can be used to convert between characters and `Int` values (corresponding to Unicode).

   Examples: `shift 5 'a' = 'f'`, `shift 21 'f' = 'a'`, `shift 5 '.' = '.'`

   Using `shift` and list-comprehensions define a function `encode :: Int -> String -> String` shifting each lower-case letter in a given string.

   Example: `encode 5 "here is an example." = "mjwj nx fs jcfruqj."` (1 point)

2. The key to having a program crack the Caesar cipher is the observation that some letters appear more frequently than others in English text. Below is a list of approximate frequencies (in percent) of the 26 letters of our alphabet (source: https://en.wikipedia.org/wiki/Letter_frequency):

   ```
   freqList = [8.2, 1.5, 2.8, 4.3, 13, 2.2, 2, 6.1, 7, 0.15, 0.77, 4, 2.4, 6.7,
               7.5, 1.9, 0.095, 6, 6.3, 9.1, 2.8, 0.98, 2.4, 0.15, 2, 0.074]
   ```

   If we measure how well a given frequency distribution matches up with the expected distribution, e.g. by using the chi-squared statistic, we can choose the shift factor that produces the best match for our decoding. Implement the following functions to help you achieve this task in the next item.

   (a) `count :: Char -> String -> Int` which returns the number of occurrences of a particular character in a string and `percent :: Int -> Int -> Float` that calculates the percentage of one integer with respect to another. (1 point)

   Examples: `count 'e' "example" = 2`, `percent 1 3 = 33.333336`

   (b) `freqs :: String -> [Float]` which computes the list of frequencies for a given string.

   Hint: `['a'..'z']` produces a list of the 26 lower-case letters. (1 point)

   Example: `freqs "abbcccdddd" = [10.0,20.0,30.000002,40.0,0.0,...,0.0]`

   (c) `chisqr :: [Float] -> [Float] -> Float` which, given a list of observed frequencies os and expected frequencies es, computes the *chi-square statistic*: (1 point)

   $$\sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

   Note that smaller results of the chi-square statistic indicate better matches between observed and expected frequencies.

[1] https://en.wikipedia.org/wiki/Caesar_cipher

(d) `rotate :: Int -> [a] -> [a]` which rotates the elements of a list n places to the left, wrapping around the end of the list, and assuming that $0 \leqslant n \leqslant$ length of the list and the function `positions :: Eq a => a -> [a] -> [Int]` which returns the list of all positions at which a value occurs in a list. (1 point)

Hint: For `positions` first pair all elements in the list with their position using `zip`.

Examples: `rotate 3 [1,2,3,4,5] = [4,5,1,2,3]`, `positions 3 [3,1,3,3] = [0,2,3]`

3. Write a function `crack :: String -> String` which attempts to decode a Caesar cipher-encoded string by first computing the frequency list of of the string, then calculating the chi-square statistic of each possible rotation of the frequency list with respect to the frequencies given in `freqList`, and finally taking the position of the minimum chi-square value as the shift factor for decoding. (In the unlikely case that there are multiple minimum positions, simply pick one.) (1 point)

   Use your cracking function to decode the following text:

   rkcuovv sc pex

Exercise 2 *Bernoulli numbers* **4 p.**

The Bernoulli numbers are a sequence of rational numbers defined like this:

$$B_0 = 1 \qquad\qquad B_n = \sum_{k=0}^{n-1} \binom{n}{k} \frac{B_k}{k-n-1} \text{ if } n > 0$$

Here, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ denotes the binomial coefficient, where $n! = 1 \cdot \ldots \cdot n = \prod_{i=1}^{n} i$ denotes the factorial. The following table lists the first few values of B_n:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	...
B_n	1	$-\frac{1}{2}$	$\frac{1}{6}$	0	$-\frac{1}{30}$	0	$\frac{1}{42}$	0	$-\frac{1}{30}$	0	$\frac{5}{66}$	0	$-\frac{691}{2730}$	...

Note that `Rational` is the type of rational numbers from the Haskell standard library. It has all the type class instances you would expect from it, including most notably a division operator `(/)` and a conversion from integers to rationals `fromInteger :: Integer -> Rational`. There is also the operator

```
(%) :: Integer -> Integer -> Rational
```

that turns a numerator a and denominator b into the rational number $\frac{a}{b}$, i.e. `1 % 2` corresponds to $\frac{1}{2}$.

1. Write functions `fact :: Integer -> Integer` and `binom :: Integer -> Integer -> Integer` that compute the factorial (resp. binomial coefficients) for non-negative inputs. (1 point)

2. Write a function `bernoulli :: Integer -> Rational` such that `bernoulli n` $= B_n$. What is the largest n for which your function still finishes in a reasonable amount of time? (1 point)

   Example: `map bernoulli [0..6] == [1%1, (-1)%2, 1%6, 0%1, (-1)%30, 0%1, 1%42]`

3. Write a function `bernoullis :: Integer -> [Rational]` that, given an integer $n \geq 0$, computes the list of the Bernoulli numbers B_0 to B_n, i.e. `bernoullis n == map bernoulli [0..n]`. Implement it in a more efficient way than just calling `bernoulli` n times! Avoid recomputing results that you have already computed! (1 point)

4. Looking at the sequence of Bernoulli numbers, it seems that starting with B_3, every B_i with i odd is 0. It also seems that in the sequence of the remaining ones, the sign keeps alternating in every step (i.e. B_2, B_6, B_{10}, etc. are positive, B_4, B_8, B_{12}, etc. are negative).

   Write two Haskell functions `check1 :: Integer -> Bool` and `check2 :: Integer -> Bool` that check whether these conjectures are true for all B_n up to a given number n! (1 point)

Trivia: Ada Lovelace is widely credited with having written the first non-trivial computer program in 1842 – and it was an algorithm for computing Bernoulli numbers!

Functional Programming	WS 2021	LVA 703025
Exercise Sheet 10, 10 points		Deadline: Wednesday, January 12, 2022, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_10.tgz` provided on the proseminar page.
- Your .hs-files should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Connect Four* **10 p.**

In this exercise we want to extend the implementation of Connect Four from the lecture in various ways. Note that all sub-tasks can be solved independently.

1. The user-interface does not check whether input moves are valid: it is not checked whether the input from the user really is a number, and whether this number is a valid move. Both cases may lead to unintended behavior or crashes of the programs. Therefore, you should modify the user-interface in a way that it repeatedly asks for input until a valid move has been entered, e.g. as follows:

```
Choose one of [0,1,2,3,4,5,6]: five
five is not a valid move, try again: 8
8 is not a valid move, try again: 3
... accept and continue ...
```

(2 points)

2. Modify the user interface so that after a match has been completed, it asks whether another round should be played. If so, the starting player should be switched. Clearly, this also requires a change in the type of `initialState`. (2 points)

3. Extend the implementation so that it can save and load games, e.g., via file `connect4.txt`. The user interface might look like this:

```
Welcome to Connect Four
(n)ew game or (l)oad game: l
... game starts by loading state from connect4.txt ...
Choose one of [0,2,3,5,6] or (s)ave: s
... game is saved in file connect4.txt and program quits ...
```

For the implementation, note that `read . show = id` and that one can automatically derive `Read`-instances in datatype definitions. (2 points)

4. Modify the function `winningPlayer` in the game logic, so that also diagonals are taken into account.

(2 points)

5. Extend the implementation so that it can give hints. To be more precise, the user interface should inform the current player, whenever she can win within 1 or 2 moves by providing a hint. Winning in 2 moves means that after following the move from the hint, you will win the game no matter how the opponent moves in between your moves. In that case the player can type "`h`" to see a first move that leads to success.

```
Choose one of [0,2,3,5,6] or see (h)int to win within 2 moves: h
Hint: Drop a piece in column 2
Choose one of [0,2,3,5,6]: 3
... the game continues since the user is not forced to follow hints
```

(2 points)

universität innsbruck

Functional Programming | WS 2021 | LVA 703025

Exercise Sheet 11, 10 points | Deadline: Wednesday, January 19, 2022, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_11.hs` provided on the proseminar page.
- Your .hs-file(s) should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Evaluation Strategies and Kinds of Recursion* **5 p.**

1. Given the four functions:

```
double x = x * 2
square x = x * x
add2times x y = x + double y
func x y = square x + add2times y x
```

   Evaluate each of the following expressions step-by-step under the three evaluation strategies call-by-value, call-by-name, and call-by-need. (3 points)

   (a) `add2times (5+2) 8`

   (b) `double (square 5)`

   (c) `func (2+2) 4`

2. For each of the following functions, specify which kind of recursion they use: (1 point)

   (a)
```
squareList [] = []
squareList (x:xs) = x*x : squareList xs
```

   (b)
```
doubleTimes x 0 = x
doubleTimes x y = doubleTimes (x+x) (y-1)
```

   (c)
```
add2List [] = 0
add2List (x:xs) = x + add2List xs
```

   (d)
```
average :: [Double] -> Double
average xs = aux xs 0 (fromIntegral (length xs))
  where aux [] s c = s / c
        aux (x:xs) s c = aux xs (s+x) c
```

3. Implement two variants of a function that takes a string and produces an upper case version of it: `stringToUpperTail` using tail recursion and `stringToUpperGuarded` using guarded recursion. For example `stringToUpperTail "Hello" = stringToUpperGuarded "Hello" = "HELLO"`. (1 point)

Exercise 2 *Lazyness and Infinite Data Structures* **5 p.**

A rooted graph consists of a set of edges between nodes – of the form (source, target) – and additionally has a distinguished node called root. For instance, Figure 1a contains a rooted graph with distinguished node 1 and edges $\{(1,1),(1,2),(1,3),(1,4),(2,1),(3,1),(4,1)\}$.
One way of representing (possibly infinite) rooted graphs is to use (possibly infinite) trees, the so-called *unwinding* of a graph. For example the rooted graph of Figure 1a can be represented by the unwinding shown in Figure 1b.

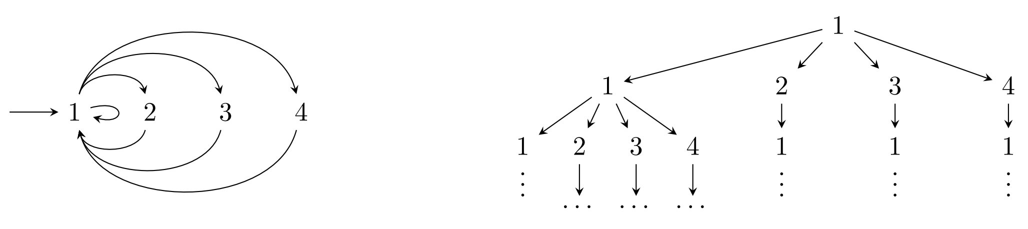


Figure 1: A graph and its unwinding

In this exercise graphs and (infinite) trees are represented by the following Haskell type definitions:

```
type Graph a = [(a, a)]
type RootedGraph a = (a, Graph a)
data Tree a = Node a [Tree a] deriving (Eq, Show)
```

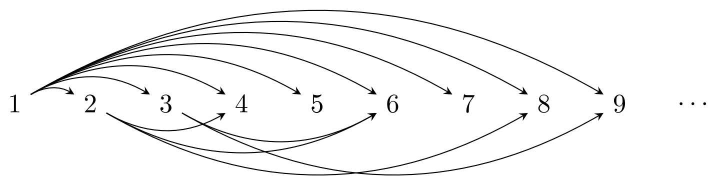
1. Implement a function `unwind :: Eq a => RootedGraph a -> Tree a` that converts a rooted graph into its tree representation. (1 point)

2. Implement a function `prune :: Int -> Tree a -> Tree a` such that `prune n t` results in a pruned tree where only the first `n` layers of the input tree are present. For example invoking `prune 2` on the infinite tree in Figure 1b drops all parts that are depicted by $\dots$ and $\vdots$, and `prune 0` would return a tree that just contains the root node 1.

   Consider the tree that results from unwinding the rooted graph `(z, [(x,z), (z,x), (x,y), (y,x)])`, a figure of eight: $\begin{tikzcd} {} \arrow[r] & z \arrow[r, bend right] & x \arrow[l, bend right] \arrow[r, bend left] & y \arrow[l, bend left] \end{tikzcd}$. What is the result of `prune 4` on this tree? (1 point)

3. Implement a function `narrow :: Int -> Tree a -> Tree a` that restricts the number of successors for each node of a tree to a given maximum (by dropping any surplus successors). For example, when calling the function `narrow 1` on the tree $\begin{tikzcd} & 1 \arrow[dl] \arrow[dr] & \\ 2 & & 3 \end{tikzcd}$, the result would be the tree $\begin{tikzcd} 1 \arrow[d] \\ 2 \end{tikzcd}$. (1 point)

4. Define an infinite tree `mults :: Tree Integer` that represents the graph where every natural number, starting from 1 points to all its multiples: (1 point)



5. Describe the results of evaluating each of the following three expressions: `narrow 4 $ prune 2 mults`, `narrow 1 mults`, and `prune 1 mults`. (1 point)

Functional Programming — WS 2021 — LVA 703025

Exercise Sheet 12, 10 points — Deadline: Wednesday, January 26, 2022, 6am

- Mark your completed exercises in the OLAT course of the PS.
- You can start from `template_12.tgz` provided on the proseminar page.
- Your .hs-file(s) should be compilable with ghci and be uploaded in OLAT.

Exercise 1 *Cyclic Lists* **5 p.**

We say that a number n is *special* if and only if it satisfies one of the following two conditions:

- $n = 1$, or
- there is some special number m such that $n = 3m$ or $n = 7m$ or $n = 11m$

The aim of this exercise is to compute the infinite list of all special numbers in ascending order.

1. Write a function `merge` that merges two lists into one. `merge xs ys` should fulfill the following conditions:
   - All elements in `merge xs ys` are also elements from `xs` or `ys`.
   - If `xs` and `ys` are in ascending order and contain no duplicates, then `merge xs ys` is in ascending order and contains no duplicates.

   Example: `merge [1,18,200] [19,150,200,300] = [1,18,19,150,200,300]` (1 point)

2. Define the infinite list `sNumbers` that computes the infinite list of special numbers in ascending order without duplicates as a cyclic list.

   Hint: Use the function `merge` and functions like `map (3*)`. Also have a look at the definition of `fibs` on slide 7 of lecture 12.

   Example: `take 10 sNumbers = [1,3,7,9,11,21,27,33,49,63]` (2 points)

3. Convince yourself that the computation of special numbers is not that easy and also not that efficient without infinite lists: implement a function `sNum :: Int -> Integer` where `sNum i` computes the `i`-th special number, i.e., `sNum i == sNumbers !! i`, where the implementation of `sNum` must not use lists, and compare the execution times of `sNum 200` and `sNumbers !! 200`.

   Hint: Try to define a predicate that tests whether a number is special; a special number has a prime factorization of a very specific shape. (2 points)

Exercise 2 *Partitions* **5 p.**

In this exercise, you will develop an abstract datatype to represent *partitions*. Given a set A with n elements, a partition is a set of disjoint sets whose union is A. In other words: a partition distributes the n elements of A into different groups, where each element is a member of exactly one group.
In Haskell, we want to have a type `Partition a` that represents a partition of some (finite) set of elements of type A. It must support at least the following operations:

Discrete partition Given a set A, return the *discrete partition* over that set, i.e. the partition in which every element is its own group. (e.g. the discrete partition of $\{1,2,3\}$ is $\{\{1\},\{2\},\{3\}\}$.

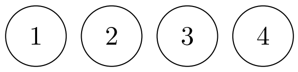


Figure 1: The discrete partition of the set $\{1, 2, 3, 4\}$.

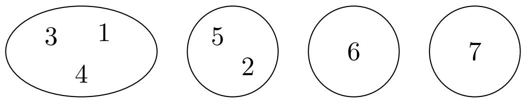


Figure 2: The partition $\{\{1, 3, 4\}, \{2, 5\}, \{6\}, \{7\}\}$, which partitions the set $\{1, 2, 3, 4, 5, 6, 7\}$.

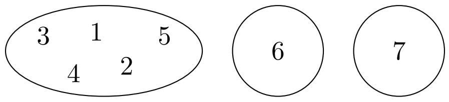


Figure 3: The partition from Figure 2 after elements 4 and 5 have been joined.

Relatedness Given a partition and two elements, determine whether two elements are *related* (i.e. they are in the same group).

Representative Given a partition and one element that is in it, return a canonical representative of that element's group. "Canonical" means that if two elements are related, they must be assigned the same representative.

Joining Given a partition and two elements, merge the two elements' groups into one single group (see Figure 3).

To make your life easier, you may assume an `Ord` instance on the value type `a`.

1. Create an abstract datatype for partitions. If your type has any invariants, document them. For example, if you were to write an abstract datatype for natural numbers, a reasonable representation would be `data Nat = Nat Integer` with the invariant that the integer be non-negative.

   Also provide an instance of `Eq` and a function `toLists :: Ord a => Partition a -> [[a]]` that returns a list of all the groups (the order is irrelevant, but no duplicate elements are allowed).

   Be careful to ensure that your implementation of `==` actually returns `True` for any two partitions that are equal in the mathematical sense.

   The code template already provides a function `pretty :: Ord a => Partition a -> String` that allows you to print your partitions in mathematical notation. (2 points)

2. Implement the functions

   ```
   discrete :: Ord a => [a] -> Partition a
   representative :: Ord a => a -> Partition a -> a
   related :: Ord a => a -> a -> Partition a -> Bool
   join :: Ord a => a -> a -> Partition a -> Partition a
   ```

   For the `discrete` operation, you may assume that the input list contains no duplicate elements. If any of the other functions are given an element that is not in the partition, the result may be whatever you want (including a crash). (2 points)

   Example:

   ```
   testPartition = join 1 3 $ join 2 5 $ join 1 4 $ discrete [1..7]

   toLists $ discrete [1..4] = [[1],[2],[3],[4]]
   toLists testPartition == [[1,3,4],[2,5],[6],[7]]
   related 1 3 testPartition == True
   related 1 5 testPartition == False
   representative 3 testPartition == 1   -- or 3 or 4
   toLists $ join 4 5 testPartition == [[1,2,3,4,5],[6],[7]]
   ```

   Note: The order of the lists returned by `toLists` is up to you and may be different from the one in the above examples.

3. As an application of your abstract datatype, the code template contains an implementation of *Kruskal's algorithm* for computing the minimal spanning forest of a weighted undirected graph. This algorithm works like this:

   - Start with the discrete partition of the graph's vertices
   - Traverse the edges of the graph by order of ascending weights
   - For every edge, check if the two vertices are related (i.e. in the same group of the partition). If not, add the edge to the result and join the two vertices in the partition.

   In other words, the algorithm uses your `Partition` datatype to keep track of which of the vertices are connected already. Every group in the partition corresponds to a connected component of the forest that has been built up so far.

   Use the tests in the code template to check that the algorithm works correctly with your abstract datatype. (1 point)