# Programming in Haskell – Homework Assignment 4

UNIZG FER, 2014/2015

Handed out: October 27, 2014. Due: November 2, 2014 at 23:59

*Note:* Define each function with the exact name and the type specified. You can (and in most cases you should) define each function using a number of simpler functions. Provide a type signature above each function definition and comment the function above the type signature. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message. Problems marked with a star ($\star$) are optional.

Each problem is worth a certain number of points. The points are given at the beginning of each problem or subtask (if they are scored independently). These points are scaled, together with a score for the in-class exercises, if any, to 10. Problems marked with a star ($\star$) are scored on top of the mandatory problems, before scaling. The score is capped at 10, but this allows for a perfect score even with problems remaining unsolved.

1. (1 point) Define a function `leftFactorial` that calculates the left factorial of a number. The left factorial can be defined as $!n = \sum_{i=0}^{n-1} i!$, $!0 = 0$.

   ```
   leftFactorial :: Integer -> Integer
   leftFactorial 0 ⇒ 0
   leftFactorial 4 ⇒ 10
   leftFactorial 20 ⇒ 128425485935180314
   ```

2. (1 point) Implement a function `factorialZeroes` that computes the number of zeroes $n!$ ends with.

   ```
   factorialZeroes :: Int -> Int
   factorialZeroes 0 ⇒ 0
   factorialZeroes 5 ⇒ 1
   factorialZeroes 34324321 ⇒ 8581073
   ```

3. (1 point) Define a function `interleave` that, given a list in format `[L1,L2,L3,R1,R2,R3]`, returns a list formatted as `[L1,R1,L2,R2,L3,R3]`. If the list has an odd number of elements, consider them in the format such as `[L1,L2,R1]` and return a list formatted as `[L1,R1,L2]`. The list can contain any number of elements.

   ```
   interleave :: [a] -> [a]
   interleave [1,2,3] ⇒ [1,3,2]
   interleave [1,3,2,4] ⇒ [1,2,3,4]
   interleave [1,2,3,4,5] ⇒ [1,4,2,5,3]
   interleave [1,1,1,1,1] ⇒ [1,1,1,1,1]
   interleave [] ⇒ []
   ```

4. (1 point) Define a function `pairs` that, for a given list containing no duplicates, returns a list of pairs `(x,y)` such that `x` differs from `y` and contains no symmetric pairs `(y,x)`. The function type must not define any typeclass constraints.

   ```
   pairs :: [a] -> [(a, a)]
   pairs [] ⇒ []
   pairs [1] ⇒ []
   pairs [1,2] ⇒ [(1,2)]
   pairs [1..3] ⇒ [(1,2),(1,3),(2,3)]
   pairs [1..4] ⇒ [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
   ```

5. (1 point) Define a function `shortestSub` that finds the shortest repeating sublist of elements within a given list. Hint: `Data.List.inits` or `Data.List.tails` might be helpful.

   ```
   shortestSub :: Eq a => [a] -> [a]
   shortestSub [] ⇒ []
   shortestSub "meow" ⇒ "meow"
   shortestSub "woofwoof" ⇒ "woof"
   shortestSub (replicate 10 True) ⇒ [True]
   shortestSub (concat $ replicate 100 [1..5]) ⇒ [1,2,3,4,5]
   ```

6. (1 point) In this task we will be implementing functions that manipulate timestamps. Timestamps are represented by a list that can be in three different formats:

   - `[Seconds]`
   - `[Minutes,Seconds]`
   - `[Hours,Minutes,Seconds]`

   Everything else is an invalid timestamp. Solve the task using pattern matching.

   ```
   type Timestamp = [Int]
   ```

   (a) Define `isVaildTimestamp` that checks if a timestamp contains valid values. (Hour must be within the interval `[0..23]`, etc.)

   ```
   isValidTimestamp :: Timestamp -> Bool
   isVaildTimestamp [5] ⇒ True
   isValidTimestamp [23,45,19] ⇒ True
   isValidTimestamp [23,70,19] ⇒ False
   isValidtimestamp [] ⇒ False
   ```

   (b) Define `timestampToSec` that converts a given timestamp to seconds.

   ```
   timestampToSec :: Timestamp -> Int
   timestampToSec [5] ⇒ 5
   timestampToSec [30,40] ⇒ 1840
   timestampToSec [23,45,19] ⇒ 85519
   timestampToSec [] ⇒ error "Invalid timestamp"
   timestampToSec [23,45,19,30] ⇒ error "Invalid timestamp"
   ```

   (c) Define `timeDiff` that calculates a temporal difference, in seconds, between two timestamps.

   ```
   timeDiff :: Timestamp -> Timestamp -> Int
   timeDiff [40] [20] ⇒ [20]
   ```

```
timeDiff [2,20] [0,15] ⇒ [125]
timeDiff [23,59,59] [0] ⇒ [84599]
timeDiff [15,30] [20,0] ⇒ [270]
timeDiff [200,12] [20,0] ⇒ error "Invalid timestamp"
```

7. Try out the function `Data.List.group`. It groups a list of elements into sublists containing equal neighbouring elements. For example:

   ```
   group :: Eq a => [a] -> [[a]]
   group [4,4,1,1,2,4,3,3,3] ⇒ [[4,4],[1,1],[2],[4],[3,3,3]]
   ```

   (a) (0.5 points) Using `group` and list comprehensions, define the function counts that, given a list of elements, returns a list of pairs of (element, number of occurences of the element). Hint: sort the list before giving it to `group`.

   ```
   counts :: Ord a => [a] -> [(a, Int)]
   counts [7] ⇒ [(7,1)]
   counts "igloo" ⇒ [('g',1),('i',1),('l',1),('o',2)]
   counts "kikiriki" ⇒ [('i',4),('k',3),('r',1)]
   counts [1,9,9,3,2,9,1] ⇒ [(1,2),(2,1),(3,1),(9,3)]
   ```

   (b) (1 point) Define a function `group'` that does the same as `Data.List.group`, but doesn't require the equal elements to be neighbours in order to be grouped together. Make sure it's of the same type as `group`, but don't use `group` internally. Hint: count the number of an element's occurences in the list.

   ```
   group' :: Eq a => [a] -> [[a]]
   group' [1,2,4,3,4,1] ⇒ [[1,1],[2],[4,4],[3]]
   ```

   (c) (0.5 points) The function `counts` has a type constraint narrower than it could have been. We'd like to be able to count elements of a type belonging only to the `Eq` class, and not necessarily also the `Ord` class. Using `group'` and list comprehensions, define a more general version of the `counts` function (note the usage of `Eq`, and not `Ord`):

   ```
   counts' :: Eq a => [a] -> [(a, Int)]
   counts' xs ⇒ counts xs
   ```

8. Let's play a game of *Lights Out!*. In case you didn't know, the game consists of a rectangular grid where each unit square contains a tiny light bulb that can be either on or off. In one move, a player can select a single light bulb and change the state (from off to on, or from on to off) of that bulb and all of its adjacent bulbs (squares are considered adjacent if they share a side). The goal of the game is to turn the lights out in a minimal number of moves.

   We will represent the grid as a matrix of binary strings where `'1'` denotes that the corresponding light bulb is on, and `'0'` denotes that the light bulb is off.

   ```
   type Grid = [String]
   ```

   (a) (1 point) Define a function `lightsOutLite` that takes a `Grid` and computes the minimal number of moves to complete a simplified version of *Lights Out!*. In this version, a player can only change the state of one light bulb in a single move.

```
lightsOutLite :: Grid -> Int
lightsOutLite ["101","11"] ⇒ error "Broken grid!"
lightsOutLite ["000","000"] ⇒ 0
lightsOutLite ["10101","11011","00000","11111"] ⇒ 12
```

(b)⋆ (2 points) Define a function `lightsOut` that computes the minimal number of moves necessary to complete the original version of the game.

```
lightsOut :: Grid -> Int
lightsOut ["101","11"] ⇒ error "Broken grid!"
lightsOut ["01101","00000","00111","00101","00100"]
⇒ error "Impossible game!"
lightsOut ["00011"] ⇒ 1
lightsOut ["11011","10101","01110","10101","11011"] ⇒ 5
```

9. (a) (1 point) Implement a function `oneEdits` that, given a `String`, returns all possible strings that are one edit–distance away from it. A string `x` is considered to be one edit–distance away from a string `y` if one of the following applies:

   - `x` is the same as `y`, but contains an extra inserted character (for example: `"home"` and `"homer"`, `"okay"` and `"okaly", etc.`
   - `x` is the same as `y` but contains one less character (for example: `"seven"` and `"even"`, `"john"` and `"jon"`, etc.)
   - `x` is the same as `y`, but has one of its characters replaced with a different one (for example: `"geek"` and `"meek"`, `"reef"` and `"reek"`, etc.)
   - `x` is the same as `y`, but has one of the characters swapped with a neighbouring one (for example: `"test"` and `"tets"`, `"meow"` and `"mewo"`, etc.)

   Limit yourself to strings of only lowercase English alphabet characters (i.e. `['a'..'z']`). No other strings will be given as input, nor should be given as output. Sort the resulting list in ascending order. The list should not contain duplicates. Try to divide your solution into multiple smaller functions, each dealing with a specific subtask.

   ```
   oneEdits :: String -> [String]
   "ih" `elem` oneEdits "hi" ⇒ True
   "hai" `elem` oneEdits "hi" ⇒ True
   ```

   (b) (1 point) Now define a function `twoEdits`, that does the same as `oneEdits`, but edit–distances of 2 instead: a chain of two possible modifications of the string (as described above). Hint: this shouldn't be a lot of work.

   ```
   twoEdits :: String -> [String]
   "hiya" `elem` twoEdits "hi" ⇒ True
   "god" `elem` twoEdits "do" ⇒ True
   ```

   You can perform further testing on your functions by using the code provided below and the files provided with the homework on the PUH official site.

   ```
   compareToFile :: (String -> [String]) -> String -> FilePath -> IO Bool


   compareToFile f s file = do
      list <- readFile file
      return $ f s == (read list :: [String])
   ```

```
testOneEdits :: IO Bool
testOneEdits = compareToFile oneEdits "hi" "oneEdits.txt"

testTwoEdits :: IO Bool
testTwoEdits = compareToFile twoEdits "hi" "twoEdits.txt"
```

10. (1 point) Often, when working with a general-purpose programming language, we need to generate pseudorandom values. Define a set of pure (side-effect free) functions that help us generate an infinite list of pseudorandom integers.

    In order to accomplish this, use the linear congruential generator algorithm combined with a simplification: skip the step where a number is reduced to a subset of its bits. A pseudorandom number is therefore equal to (a∗X + c) % m, where X is a given seed value and the other values are constants which you may pick from the linked wiki page or choose yourself.

    (a) Define **fromSeed** that generates a pseudorandom integer given a seed value (use the formula above). Note: this function's behaviour depends on the constants chosen above, so your results may differ from the ones presented below, in task 10b.

    ```
    type Seed = Int
    fromSeed :: Seed -> Int
    ```

    (b) Define `guess`, a game that, given a starting seed value and number limit, asks the player to guess a number between 0 and the limit (inclusively). The number is generated pseudorandomly. In case the player's guess is too low, the function should return `LT`. If it is too high, it should return `GT`. Finally, if the player is really lucky and manages to guess the number, it should return `EQ`.

    ```
    guess :: Seed -> Int -> IO Ordering
    ghci> guess 6548 10
    guess:  3
    LT
    ghci> guess 6548 10
    guess:  8
    GT
    ghci> guess 6548 10
    guess:  6
    EQ
    ```