Programming in Haskell – Homework Assignment 1

UNIZG FER, 2017/2018

Handed out: October 12, 2017. Due: October 19, 2017 at 17:00

1 Instructions

1. To submit your homework you need to have a folder named after your JMBAG. In that folder there should be two files, Homework.hs for homework tasks and Exercises.hs for all in class exercises (yes you need to submit those as well). You should ZIP that whole folder and submit it through Ferko.

   Example folder structure :

   - 0036461143
     - Homework.hs
     - Exercises.hs

2. If you need some help with your homework or have any questions, ask them on our Google group.
3. Define each function with the exact name and type specified. You can (and in most cases you should) define each function using a number of simpler functions.
4. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values (must be total). Use the `error` function for cases in which a function should terminate with an error message.
5. Problems marked with a star (⋆) are optional.

2 Grading

Each problem is worth a certain number of points. The points are given at the beginning of each problem or subproblem (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 (⋆) 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 some problems remaining unsolved.

3 Tasks

1. *(2 points)* Write a function that checks if a year is a leap year. In Gregorian calendar leap years occur :

   - on every year that is divisible by 4
   - except every year that is evenly divisible by 100
   - unless the year is also evenly divisible by 400

   For example, 1997 is not a leap year, but 1996 is and 1900 is not leap year, but 2000 is.

   The function name and signature is :

   ```
   isLeapYear ::  Int -> Bool
   ```

   After that generate a list of all leap years between 1996 and 2017 and store it in :

   ```
   leapList ::  [Int]
   ```

2. *(3 points)* In this task you will have to implement the exponential function.

   (a) For that you will have to first write a function which evaluates polynomial of single indeterminante `x` where polynomial coefficients are represented by a list of `Double`s. The polynomial function can be expressed as $\sum_{k=0}^{n} a_k x^k$.

   ```
   evaluate :: Double -> [Double] -> Double
   ```

   (b) After that is done, define a function that calculates the nth factorial :

   ```
   factorial :: Double -> Double
   ```

   (c) Next, define an infinite list of coefficients for Maclaurin series which can be expressed as $(\frac{1}{n!})_{n=0}^{\infty} = (\frac{1}{0!}, \frac{1}{1!}, \frac{1}{2!}, ...)$.

   ```
   maclaurin :: [Double]
   ```

   (d) And finally, combine `evaluate`, `factorial` and `maclaurin` to create a function which approximates e^x from the first **170** terms of the series.

   ```
   exp' :: Double -> Double
   ```

   Some useful functions :

   ```
   sum, fst, snd, product, take
   ```

3. *(3 points)* In this task you will have to implement some polymorphic utility functions for working on lists of key value pairs where keys are strings.

   (a) The Function for getting a pair with a certain key which should return a list with a single element as a result (or no elements if key doesn't exist) :

   ```
   findItem :: [(String, a)] -> String -> [(String, a)]
   ```

   (b) Function that checks if a list contains an element with a certain key :

   ```
   contains :: [(String, a)] -> String -> Bool
   ```

   (c) Function that tries to retrieve a value with a certain key or throws an error if the key doesn't exist (example of error function usage : `error "I'm an error message"`) :

   ```
   lookup :: [(String, a)] -> String -> a
   ```

   (d) Function that inserts a new key value pair. If key already exists than do nothing :

   ```
   insert :: [(String, a)] -> (String, a) -> [(String, a)]
   ```

   (e) Function that removes a key value pair with the certain key :

   ```
   remove :: [(String, a)] -> String -> [(String, a)]
   ```

   (f) Function that updates the value of a certain key (if the key doesn't exist, function does nothing) :

   ```
   update :: [(String, a)] -> String -> a -> [(String, a)]
   ```

   Some useful functions :

   ```
   not, null, error
   ```

4. *(3 points(⋆))* In this task you will have to implement a function that is calculating how similar two texts are.

   To do that, we can imagine that each text is a vector in vector space composed of union of words from those two texts.

   Their similarity can than be expressed as cosine of angle between those two vectors, with values ranging from 0 (meaning not similar) to 1 (meaning exactly the same / angle is 0). Values can't be negative because we are relying on word frequency in this case. This is also known as cosine similarity so check the wiki in case you need some help.

   Make sure you remove all non letter characters from text (space is considered non letter character so be careful with that) and convert it to lowercase to get best results.

   Implement function :

   ```
   cosineSimilarity ::  String -> String -> Double
   ```

   Which takes in two texts and returns a value between 0 and 1 that describes how "similar" they are.

   Fell free to define as many "helper" functions as you need. You should always try to break the problem in smaller pieces which you can compose into your final solution.

   For strings

   `"Haskell makes me giddy!  I want to jump around like a little girl."`

   and

   `"Haskell makes me happy!  So much that I want to jump around like a little pony."`

   `cosineSimilarity` should give you `0.76271276980969`

   Some useful functions :

   ```
   sum, fst, snd, zip, fromIntegral, toLower, isLetter, isSpace, length,
                              words, nub, sort
   ```