Programming in Haskell – Homework Assignment 5

UNIZG FER, 2017/2018

Handed out: December 8, 2017. Due: December 15, 2017 at 23:00

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 problems 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

   You can download the homework template file from the FER web repository.

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 sub-problem (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 Problems

1. *(3 points)* A robot factory's test facility needs a program to verify robot movements. The robots have three possible movements:

   - turn right
   - turn left
   - advance

   Robots are placed on a hypothetical infinite grid, facing a particular direction (north, east, south, or west) at a set of {`x,y`} coordinates, e.g., {`3,8`}, with coordinates increasing to the north and east.

   The robot then receives a number of instructions, at which point the testing facility verifies the robot's new position, and in which direction it is pointing.

   The letter-string "RAALAL" means:

   - Turn right
   - Advance twice
   - Turn left
   - Advance once
   - Turn left yet again

   Say a robot starts at {`7, 3`} facing north. Then running this stream of instructions should leave it at {`9, 4`} facing west.

   To complete this exercise, you need to create the data type Robot, and implement the following functions and data types:

```
data Bearing
data Robot
bearing :: Robot -> Bearing
coordinates :: Robot -> (Integer, Integer)
mkRobot :: Bearing -> (Integer, Integer) -> Robot

simulate :: Robot -> String -> Robot
-- simulate should be implemented in terms of fold

advance :: Bearing -> (Integer, Integer) -> (Integer, Integer)
turnLeft :: Bearing -> Bearing
turnRight :: Bearing -> Bearing
```

2. *(2 point)* Determine if a triangle is equilateral, isosceles, scalene, or degenerate.

   An `Equilateral` triangle has all three sides the same length.

   An `Isosceles` triangle has at least two sides the same length. (It is sometimes specified as having exactly two sides the same length, but for the purposes of this exercise we'll say at least two.)

   A `Scalene` triangle has all sides of different lengths.

   A `Degenerate` triangle has zero area and looks like a single line. The sum of the lengths of two sides equals that of the third.

   For a shape to be a triangle at all, all sides have to be of length ¿ 0, and the sum of the lengths of any two sides must be greater than or equal to the length of the third side. See Triangle Inequality.

   All triangles that aren't any of the above are `Illegal` triangles.

   Create a new data type Triangle and write a function that determines it's type.

   Function should have following type signature:

   ```
   triangleType :: (Ord a, Num a) => a -> a -> a -> TriangleType
   ```

3. *(3 points)* In this problem you will have to implement a utility for splitting lists on sub lists. This can be quite useful for splitting delimited strings (and will be useful in the next homework so make sure you solve this exercise :).

   It's not very hard to solve this problem with standard recursions but this time you **have** to use folds. Also, keep in mind that not all folds are equal. There is clearly a better and worse choice so make sure you pick the right one for the job.

   In case you need to return more than one value from the fold function, remember that you can use tuples for that (even though it is not necessary in this case).

   You should implement a function with a following type signature:

   ```
   splitter ::  Eq a => [a] -> [a] -> [[a]]
   ```

   The First argument is a sub list (which is used as a delimiter) and the second argument is the delimited list that we want to split up.

   Here are some test examples (you don't have to cover the case of splitting on an empty list):

   ```
   splitter " > " " > " => ["", ""]
   splitter " > " "123 > " => ["123", ""]
   splitter " > " "123 > 456 > 789" => ["123", "456", "789"]
   ```

   As a final tip, this is not a one liner but it can be solved in about 5 short lines of code (give or take), you also might find some functions from 'Data.List' useful (one in particular).

4. *(4 points(⋆))* While playing with the third problem we have concluded it is not possible to make the `splitter` function work on infinite lists (unless we implement it with standard recursions instead of fold).

   This conclusion was based more on a "feeling" rather than hard science and so, because our time is precious, we have decided to let you try to prove us wrong (or right).

   If you think we are wrong, then try to make `splitter` work on infinite lists. If you think we are right, then try to explain why.

   If you need an infinite list to test your function with you can try this :

   ```
   cycle "123 > 456 > "
   ```

Corrections