Programming in Haskell – Homework Assignment 3

UNIZG FER, 2017/2018

Handed out: October 30, 2017. Due: November 6, 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. *(2 points)* A local maximum of a list is an element of the list which is strictly greater than both the elements immediately before and after it. For example, in the list [2,3,4,1,5], the only local maximum is 4, since it is greater than the elements immediately before and after it (3 and 1). The number 5 is not a local maximum since there is no element that comes after it.

   Write a function:

   ```
   localMaxima ::  [Int] -> [Int]
   ```

   which finds all the local maxima in the input list and returns them in order. For example:

   ```
   localMaxima [2,9,5,6,1] ⇒ [9,6]
   localMaxima [2,3,4,1,5] ⇒ [4]
   localMaxima [1,2,3,4,5] ⇒ []
   ```

2. *(2 point)* Scrabble is a word game in which two to four players score points by placing tiles bearing a single letter onto a board divided into a 15x15 grid of squares. The tiles must form words which, in crossword fashion, read left to right in rows or downwards in columns, and be defined in a standard dictionary or lexicon.

   In this problem you will have to extract the scrabble scores from a legacy system and convert them to the new system.

   The old system stored a list of letters per score:

   `[(Int, String)]`

   - 1 point: "AEIOULNRST"
   - 2 points: "DG"
   - 3 points: "BCMP"
   - 4 points: "FHVWY"
   - 5 points: "K"
   - 8 points: "JX"
   - 10 points: "QZ"

   The shiny new scrabble system instead stores the score per letter, which makes it much faster and easier to calculate the score for a word. It also stores the letters in lower-case regardless of the case of the input letters:

   `[(Char, Int)]`

   - "a" is worth 1 point.
   - "b" is worth 3 points.
   - "c" is worth 3 points.
   - "d" is worth 2 points.

   There is no need to implement a function that will print out our list of `(Char, Int)` tuples. This was just given as an example of how such value could be used.

   Your task is to implement a function:

```
transform ::  [(Int,String)] -> [(Char,Int)]
```

3. *(4 points)* Wolfram code is a naming system often used for one-dimensional cellular automaton rules, introduced by Stephen Wolfram.

   The cellular automaton consists of a one-dimensional array of cells which can be either on or off. Each code describes a set of rules which determines the next configuration of cells based on their neighbours.

   You will implement a simplified version of the automaton, which works on a fixed-size array, and only implements the simple Rule 90, so you can hard code the behaviour in instead of having to decode the rules. Please read the linked Wikipedia page for more details.

   Define a function:

   ```
   rule90 :: [Bool] -> [[Bool]]
   ```

   which takes the initial state of the array of cells and returns an infinite list of consecutive states of the array.

   To help with defining the function, define:

   ```
   rule90Step :: [Bool] -> [Bool]
   ```

   which returns the next state of the array. The edges of the array are assumed to be `False`.

   And finally, define:

   ```
   pretty :: [[Bool]] -> String
   ```

   which returns a `String` where each line contains a state of the board, and where `True` cells are represented with a '`#`' and `False` cells are represented with a space.

   (You can find the `xor` function in the `Data.Bits` module. Alternatively, you can use the inequality `(/=)` operator.)

   An example:

   ```
   ghci> putStr $ pretty $ take 8 $
         rule90 (replicate 7 False ++ [True] ++ replicate 7 False)

          #
         # #
        #   #
       # # # #
      #       #
     # #     # #
    #   #   #   #
   # # # # # # # #
   ```

4. *(3 points(⋆))* Write a co-recursive function `f` that will create a list that goes like this:

```
[1, 11, 21, 1211, 111221, ...]
```

It's a popular riddle for kids (and adults). It is advised that you stop reading and think about how this sequence works.

But if you want to make it easier for yourself, continue reading. This sequence is called a "Look-and-say" sequence. To generate a member of the sequence from the previous member, read off the digits of the previous member, counting the number of digits in groups of the same digit. For example: 1 is read off as "one 1" or 11. After that 11 is read off as "two 1s" or 21.

You can read more on this topic on the Wikipedia.

Create additional helper functions if needed.

For this problem it can be useful to use two additional functions from the `Data.List` module.

```
takeWhile ::  (a -> Bool) -> [a] -> [a]

dropWhile ::  (a -> Bool) -> [a] -> [a]
```

They work like take and drop functions respectively, but take an additional function as an argument that acts as a condition for taking and droping. For example:

```
takeWhile (== 'a') "aaabcbaaaba" ⇒ "aaa"
dropWhile (== 'a') "aaabcbaaaba" ⇒ "bcbaaaba"
```

Type signature will be:

```
f ::  [String]
```

Here are some examples:

```
take 4 f ⇒ ["1","11","21","1211"]
f !!  8 ⇒ "31131211131221"
f !!  5 ⇒ "312211"
```