Programming in Haskell – Homework Assignment 3

UNIZG FER, 2016/2017

Handed out: October 27, 2016. Due: November 3, 2016 at 17:00

Note: Define each function with the exact name specified. You can (and in most cases you should) define each function using a number of simpler functions. 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 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 ($\star$) 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.

1. *(4 pts)* The purpose of this assignment is to create functions to allow conversions of Roman numerals to decimal notation and vice versa. To help you accomplish this you will implement several helper functions. If you are not familiar with Roman numerals, you can find information about them on Wikipedia or at Project Euler. Also, since Roman notation is ambiguous, you will be required to provide the minimal form of the numeral as described in the Project Euler link. We will stick to the convention that you cannot repeat the same literal more than three times thus limiting the range of possible numbers to [1..3999]. Roman numbers will be represented as `type RomanNumeral = String`. The functions you need to implement are:

   (a) *(1 pt)* `isValidRoman :: RomanNumeral -> Bool` Checks whether the given numeral consists only of valid literals (only letters 'I', 'V', 'X', 'L', 'C', 'D', 'M') and whether it follows the rules of writing. You can get more help with the rules if you take a look at Appendix A. Examples:
   ```
   isValidRoman "MCMXLVI" ⇒ True
   isValidRoman "XVIII" ⇒ True
   isValidRoman "AQUILA" ⇒ False
   isValidRoman "MCCMXLVI" ⇒ False
   ```

   (b) *(2 pts)* `toRoman :: Int -> RomanNumeral` Converts the given number to Roman notation. The function should throw an error if the number is out of the valid `[1..3999]` range. Examples:
   ```
   toRoman 1776 ⇒ "MDCCLXXVI"
   toRoman 476 ⇒ "CDLXXVI"
   toRoman 9000 ⇒ error "Number cannot be represented"
   toRoman (-100) ⇒ error "Number cannot be represented"
   ```

(c) *(1 pt)* `fromRoman :: RomanNumeral -> Int` Converts the Roman numeral to a decimal representation. This function should throw an error if the number is not a valid Roman numeral. Examples:

```
fromRoman "XXX" ⇒ 30
fromRoman "MMMDCCCLXXXVIII" ⇒ 3888
fromRoman "AUGUSTUS" ⇒ error "Not a valid Roman numeral"
fromRoman "GEMINA" ⇒ error "Not a valid Roman numeral"
```

2. *(2 pts)* It's friday night and Jack and his friends are planning to visit multiple pubs. Given position of their home and positions of pubs they want to go to, define function shortestDistance which calculates shortest distance they have to take to visit all the pubs. Don't forget to include their way back home. They don't want to get lost taking shortcuts so they decided to walk only on main streets. Because of this, use Manhattan distance to calculate the path. You are not allowed to use functions from Data.List. Examples:

```
shortestDistance :: (Int, Int) -> [(Int, Int)] -> Int

shortestDistance (2,5) [(10,12),(15,17), (8,13)] ⇒ 50
shortestDistance (8,9) [(6,7),(8,12), (4,9)] ⇒ 18
shortestDistance (8,9) [] ⇒ 0
shortestDistance (8,9) [(4, 2)] ⇒ 22
```

3. *(3 pts)* Define the following functions that work with discrete random variables. You can assume the probabilities will always be valid ($0 \leq p_i \leq 1$, $\forall i$ and $\sum_{i=1}^{N} p_i = 1$). The random variables are defined as follows:

```
type Probability = Double
type DiscreteRandVar = [(Int, Probability)]
x :: DiscreteRandVar
x = [(1, 0.2), (2, 0.4), (3, 0.1), (4, 0.2), (5, 0.05), (6, 0.05)]
```

Note: You may need to use the function `fromIntegral` in some of the subtasks to satisfy the type system.

(a) Define an explicitly recursive function `mean` and an accumulator-style recursive function `mean'` that calculate the mean (or expected value) of a discrete random variable, defined as $\sum_{i=1}^{N} x_i \cdot p_i$.

```
mean  :: DiscreteRandVar -> Double
mean' :: DiscreteRandVar -> Double
mean x ⇒ 2.65
mean' x ⇒ 2.65
```

(b) Define an explicitly recursive function `variance` and an accumulator-style recursive function `variance'` that calculate the variance of a discrete random variable. Given the mean of the variable as μ_x, it is defined as $\sum_{i=1}^{N} (x_i - \mu_x)^2 \cdot p_i$.

```
variance  :: DiscreteRandVar -> Double
variance' :: DiscreteRandVar -> Double
variance x ⇒ 1.9275
```

```
variance' x ⇒ 1.9275
```

(c) Define an explicitly recursive function `probabilityFilter` and an accumulator-style recursive function `probabilityFilter'` that take a probability and a random variable and return a list of values that have at least the given probability of appearing, *in the same order* in which they appear in the random variable definition.

```
probabilityFilter  :: Probability -> DiscreteRandVar -> [Int]
probabilityFilter' :: Probability -> DiscreteRandVar -> [Int]
probabilityFilter 0 x ⇒ [1,2,3,4,5,6]
probabilityFilter' 0 x ⇒ [1,2,3,4,5,6]
probabilityFilter 0.2 x ⇒ [1,2,4]
probabilityFilter' 0.2 x ⇒ [1,2,4]
```

4. *(2 pts)* (⋆) Imagine a group of differently–sized frogs living in a swamp with only three lily pads numbered from 1 to 3. Every morning, frogs meet at the first lily pad and spend all day trying to reach the third one, thereby abiding the following rules:

   - A frog cannot jump between the first and the third lily pad as they are too far apart;
   - A frog can jump from a lily pad only if it is the smallest frog on that lily pad;
   - A frog can jump on a lily pad only if it is smaller than all frogs on that lily pad.

   Your job is to define a function `frogJumps` that, given a number of frogs `n`, computes the minimal number of jumps necessary for all `n` frogs to reach the third lily pad.

```
frogJumps :: Int -> Integer
frogJumps 1 ⇒ 2
frogJumps 2 ⇒ 8
frogJumps 5 ⇒ 242
frogJumps 20 ⇒ 3486784400
```

Appendix A

Rules for Roman numerals

Rule 1 - Repetition A single letter may be repeated up to three times consecutively with each occurrence of the value being additive. This means that I is one, II means two and III is three. However, IIII is incorrect for four.

Rule 2 - Additive Combination Larger numerals must be placed to the left of the smaller numerals to continue the additive combination. So VI equals six and MDCLXI is 1,661.

Rule 3 - Subtractive Combination A small-value numeral may be placed to the left of a larger value. Where this occurs, for example IX, the smaller numeral is subtracted from the larger. This means that IX is nine and IV is four. The subtracted digit must be at least one tenth of the value of the larger numeral and must be either I, X or C. Accordingly, ninety-nine is not IC but rather XCIX. The XC part represents ninety and the IX adds the nine. In addition, once a value has been subtracted from another, no further numeral or pair may match or exceed the subtracted value. This disallows values such as MCMD or CMC.

Rule 4 - Repeated Use of V, L and D The numerals that represent numbers beginning with a '5' (V, L and D) may only appear once in each Roman numeral. This rule permits XVI but not VIV.

Rule 5 - Reducing Values The fourth rule compares the size of value of each the numeral as read from left to right. The value must never increase from one letter to the next. Where there is a subtractive numeral, this rule applies to the combined value of the two numerals involved in the subtraction when compared to the previous letter. This means that XIX is acceptable but XIM and IIV are not.

Rule 6 - Multiplication To represent numbers of four thousand or greater, lines are added to each letter. For example, a line above a letter multiplies its value by one thousand. To represent 15,015 the Roman numerals are $\overline{VVVVV}$. This rule is not implemented in the algorithm so the code is limited to values up to but not including four thousand.

Rule 7 - Zero There is very little information that suggests that the system originally had a notation for zero. However, the letter N has been used to represent zero in a text from around 725AD. If you encounter an N, just report an error.

Corrections

To iterate is human. To recurse, divine.