Programming in Haskell – Homework Assignment 4

UNIZG FER, 2016/2017

Handed out: November 3, 2016. Due: November 17, 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. 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 (⋆) 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 (⋆) 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. *(1 pt)* Define your own version of `Data.List.Intercalate`, called `intercalate'`. The function should behave the same as `Data.List.Intercalate` for all inputs.

```
intercalate' :: [a] -> [[a]] -> [a]
intercalate' xs yss == intercalate xs yss
```

2. *(2 pts)* Often we need to split up a list into smaller chunks. Define three functions to do just that.

   (a) Define a function `chunk` that splits up a list `xs` into sublist of length `n`. If the length of `xs` is not a multiple of `n`, the last sublist will be shorter than `n`.

```
chunk :: Int -> [a] -> [[a]]
chunk 2 "shadowless" ⇒ ["sh","ad","ow","le","ss"]
chunk 4 "shadowless" ⇒ ["shad","owle","ss"]
chunk 11 "shadowless" ⇒ ["shadowless"]
chunk 0 "shadowless" ⇒ []
```

   (b) Define a function `chunkBy` that splits up a list `xs` into sublists of lengths given in a list of indices `is`. If the lengths in `is` do not add up to the length of `xs`, the remaining part of `xs` will remain unchunked.

```
chunkBy :: [Int] -> [a] -> [[a]]
chunkBy [1,2,3] "shadowless" ⇒ ["s","ha","dow"]
chunkBy [11,2] "shadowless" ⇒ ["shadowless"]
chunkBy [3,0,3,4] "shadowless" ⇒ ["sha","dow","less"]
chunkBy [] "shadowless" ⇒ []
```

(c) Define a function `chunkInto` that splits up a list `xs` into `n` sublists of equal length. If the length of `xs` is not divisible by `n`, chunk the remainder into the last sublist.

```
chunkInto :: Int -> [a] -> [[a]]
chunkInto 5 "shadowless" ⇒ ["sh","ad","ow","le","ss"]
chunkInto 3 "shadowless" ⇒ ["sha","dow","less"]
chunkInto 1 "shadowless" ⇒ ["shadowless"]
chunkInto 0 "shadowless" ⇒ []
chunkInto 11 "shadowless"
⇒ ["s","h","a","d","o","w","l","e","s","s"]
```

3. *(1 pt)* Define a function `cycleMap fs xs` that maps various functions from `fs` over a list `xs`, depending on the index of an element in the list. The list `fs` of functions to be mapped is cycled over the list `xs`: the first function from `fs` is applied on the first element from `xs`, the second function from `fs` on the second element from `xs`, etc. When the list of functions `fs` is exhausted, mapping restarts from the first function from `fs`.

```
cycleMap :: [a -> b] -> [a] -> [b]
cycleMap [odd, even] [1,2,3,4] ⇒ [True, True, True, True]
cycleMap [(+1), (subtract 1)] [1..10] ⇒ [2,1,4,3,6,5,8,7,10,9]
cycleMap [map (+1),map (`div` 2),filter (>7)] [[1,2,3],[4,5,6],[7,8,9]]
⇒ [[2,3,4],[2,2,3],[8,9]]
cycleMap [] "Whatever" ⇒ []
```

4. *(3 pts)*

(a) Define an explicitly recursive function `reduce` that reduces a list of elements to a single element using a seed value and a binary reduction function. The reduction function is applied to the seed value and the first element of the list to get an intermediate value. That value is then combined with the second element of the list to get the next intermediate value, and so on, until the end of the list is reached.

```
reduce :: (a -> b -> a) -> a -> [b] -> a
reduce (+) 0 [1,2,3] ⇒ 6
reduce (-) 0 [1,2,3] ⇒ -6
reduce (++) "a" ["b", "c"] ⇒ "abc"
reduce (\x s -> length s + x) 0 ["an", "example", "or", "something"]
⇒ 20
reduce (*) 1 [] ⇒ 1
```

(b) Define a variant of `reduce` called `reduce1` that behaves like `reduce`, but assumes the input list contains at least one element and so eschews taking a seed element.

```
reduce1 :: (a -> a -> a) -> [a] -> a
reduce1 (+) [1..3] ⇒ 6
reduce1 (++) $ words "just an example" ⇒ "justanexample"
reduce1 (-) [7] ⇒ 7
```

```
reduce1 (*) [] ⇒ error "reduce1 got an empty list"
```

(c) Define a function `scan` that performs similarly to `reduce`, but returns a list of all the intermediate values with the result at the end instead of just the last result.

```
scan :: (a -> b -> a) -> a -> [b] -> [a]
scan (+) 0 [1,2,3] ⇒ [0,1,3,6]
scan (-) 0 [1,2,3] ⇒ [0,-1,-3,-6]
scan (*) 0 [] ⇒ [0]
```

(d) Define a variant of `reduce` that performs similarly, only does the operations from right to left, instead. Call this function `rreduce`.

```
rreduce :: (a -> b -> b) -> b -> [a] -> b
rreduce (+) 0 [1,2,3] ⇒ 6
rreduce (-) 0 [1,2,3] ⇒ 2
rreduce (++) "a" ["b", "c"] ⇒ "bca"
rreduce (\s x -> length s + x) 0 ["an", "example", "or", "something"]
⇒ 20
rreduce (*) 1 [] ⇒ 1
```

(e) Define a variant of `rreduce` called `rreduce1` that behaves like `rreduce`, but assumes the input list contains at least one element.

```
rreduce1 :: (a -> a -> a) -> [a] -> a
rreduce1 (+) [1..3] ⇒ 6
rreduce1 (++) $ words "just an example" ⇒ "justanexample"
rreduce1 (-) [7] ⇒ 7
rreduce1 (^) [] ⇒ error "rreduce1 got an empty list"
```

(f) Define a variant of the `scan` function that works from right to left, called `rscan`.

```
rscan :: (a -> b -> b) -> b -> [a] -> [b]
rscan (+) 0 [1,2,3] ⇒ [6, 5, 3, 0]
rscan (-) 0 [1,2,3] ⇒ [2,-1,3,0]
rscan (*) 0 [] ⇒ [0]
```

5. *(2 pts)*

   (a) Define `newton`, a function that computes an approximation of the square root of a number using a special case of Newton's method. Assuming that some initial guess `y` is the square root of `x`, we can get a better approximation of the actual square root (`y'`) by averaging `y` and `x/y`. In other words, `y' = (y+x/y)/2`. We repeat this step with the newly found value `y'` to gain better and better approximations. The function should return a result when the difference between two successive approximations is less than some given tolerance. (*Note:* Your results might differ from the ones shown below, but should in general show improvements as the tolerance value gets smaller.)

   ```
   type Tolerance = Double
   newton :: Tolerance -> Double -> Double
   newton 1e+1 1024 ⇒ 32.02142090500024
   newton 1e-2 1024 ⇒ 32.0000071648159
   newton 1e-4 1024 ⇒ 32.0000000000008
   newton 1e-8 0 ⇒ 6.103515625e-5
   newton any (-632) ⇒ error "can't get sqrt of negative number"
   ```

   (b) Define `deriv`, a function that computes the derivative of a given function. Remember that the derivative of `f(x)` is defined as `f'(x) = [f(x+dx)-f(x)]/dx`. We'll cheat and only return an approximation of the derivative: simply assume that dx is equal to a very small number (like `0.00001`). (*Note:* Again, your results might differ slightly.)

   ```
   deriv :: (Double -> Double) -> Double -> Double
   let f = (**3)
   let f' = deriv f
   f' 2 ⇒ 12.000060000261213
   let g' = deriv sin
   g' 3.14159 ⇒ -0.9999999999996315
   ```

6. *(3 pts)* Define a function `rpnCalc` that takes a mathematical expression written in Reverse Polish notation and calculates its result, using the operators provided as a second argument to the function.

   The expression is limited to 1-digit positive integers, while the operators are always binary and of the type `Int -> Int -> Int`. The function should only work for the operators defined and should result in an error otherwise.

   ```
   type Operators = [(Char, Int -> Int -> Int)]
   basic    = [ ('+', (+))
              , ('-', (-)) ]

   standard = [ ('+', (+))
              , ('-', (-))
              , ('*', (*))
              , ('/', div)
              , ('^', (^)) ]
   ```

```
rpnCalc :: String -> Operators -> Int
rpnCalc "32-1+" basic ⇒ 2
rpnCalc "32-1+" standard ⇒ 2
rpnCalc "32*" basic ⇒ error "Invalid symbol *"
rpnCalc "321-+5-" basic ⇒ -1
rpnCalc "321-+5-" standard ⇒ -1
rpnCalc "42^2/" standard ⇒ 8
rpnCalc "35*2/" standard ⇒ 7
rpnCalc "35*7-3^43*/" standard  ⇒ 42
rpnCalc "" standard ⇒ 0
rpnCalc "22%" standard ⇒ error "Invalid symbol %"
rpnCalc "33++" ⇒ error "Invalid RPN expression"
```

7. *(2 pts)* (⋆) 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) 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) 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
   ```

8. *(2 pts)* (⋆) Define a function `permutations'` that, given a list, returns a list of all its permutations. Implement it using explicit recursion. The ordering of the list of permutations is irrelevant!

   ```
   permutations' :: [a] -> [[a]]
   permutations' "hi" ⇒ ["hi", "ih"]
   permutations' [1,2,3]
   ```

```
⇒ [[1,2,3],[2,1,3],[3,2,1],[2,3,1],[3,1,2],[1,3,2]]
permutations' [] ⇒ [[]]
```

Corrections

Practice is the best of all instructors.
Publilius Syrus