Programming in Haskell – Homework Assignment 4

UNIZG FER, 2013/2014

Handed out: November 8, 2013. Due: November 14, 2013 at 17:00

Note: Define each function with the exact name and type specified. You can (and in most cases 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, and return the exact error message specified. Problems marked with a star ($\star$) are optional.

1. Define your own implementations of some commonly used list functions by using explicit recursion and pattern matching.

   (a) Define `map'` that applies a function to all elements of a list:
   ```
   map' :: (a -> b) -> [a] -> [b]
   map' (+1) [1,4,2,5] ⇒ [2,5,3,6]
   map' not [True,False] ⇒ [False,True]
   map' not [] ⇒ []
   map' odd [1..3] ⇒ [True,False,True]
   ```

   (b) Define `filter'` that returns all list elements that satisfy a given predicate:
   ```
   filter' :: (a -> Bool) -> [a] -> [a]
   filter' odd [0,4,2,5,6,1,22,4] ⇒ [5,1]
   filter' isDigit "abc34cd2d3" ⇒ "3423"
   filter' isDigit "" ⇒ ""
   ```

   (c) Define `iterate'` that returns `[x, f x, f (f x), f (f (f x)), ...]`.
   ```
   iterate' :: (a -> a) -> a -> [a]
   iterate' succ 0 ⇒ [0..]
   iterate' (drop 1) "eatme" ⇒ ["eatme","atme",tme"..]
   ```

   (d) Define `intercalate'` that places a list between all elements of another list:
   ```
   intercalate' :: [a] -> [[a]] -> [a]
   intercalate' "cd" ["b","aa","ab"] ⇒ "bcdaacdab"
   intercalate' "+" ["1","12","7"] ⇒ "1+12+7"
   intercalate' _ [] ⇒ []
   intercalate' _ ["bc"] ⇒ "bc"
   ```

2. Define an explicitly recursive `sumEvens` that sums elements at even list positions:

```
sumEvens :: Num a => [a] -> a
sumEvens [1..5] ⇒ 1 + 3 + 5 = 9
sumEvens [7] ⇒ 7
sumEvens [] ⇒ 0
```

3. Define `dup` that duplicates list elements. It duplicates the first element once, the second element twice, and the nth element n times. Define it using explicit recursion.

```
dup :: [a] -> [a]
dup [1,2,3] ⇒ [1,1,2,2,2,3,3,3,3]
dup [5..] ⇒ [5,5,6,6,6,7,7,7,7,8,8,8,8,8,..]
dup [] ⇒ []
```

4. Define `prime` that finds out whether a number is prime or not. Define it using explicit recursion.

```
prime :: Integral a => a -> Bool
prime 7 ⇒ True
prime 21 ⇒ False
```

5. Define an explicitly recursive `words'` that splits a string into words. (Hint: use `Data.Char.isSpace`.)

```
words' :: String -> [String]
words' "this \tis a\n breeze!" ⇒ ["this","is","a","breeze!"]
words' " ignore surrounding space " ⇒ ["ignore", "surrounding", "space"]
```

6. Define `prefixCalculator` that takes an expression in prefix (Polish) notation and returns the calculated result. The input is a string that contains the operators and operands separated with whitespaces. The operators are `+`, `*`, `-`, and `/`, while the operands are real numbers. You can assume that the input is a valid expression in prefix notation, i.e., you don't have to handle the cases when the input is malformed. (Hint: you can use a list to mimic a stack, cf. this example.)

```
prefixCalculator :: String -> Double
prefixCalculator "+ * 2 3 / 4.3 - 5 0.7" ⇒ 2 * 3 + 4.3 / (5-0.7) = 7
prefixCalculator "+ + + 2 0.9 5 0" ⇒ 7.9
prefixCalculator "- + 5.8 * -10 - 3 / 15 * 2.3 3 -6" ⇒ 3.53913
prefixCalculator "/ 5 0" ⇒ error "Division with zero"
```

7. Define `findSubsequence xs ys` that (cf. subsequence) that returns the ascending list of indices at which subsequence `xs` occurs in list `ys`. If there are multiple solutions, return one with the smallest sum of all indices (there will always be just one such solution and you can obtain it easily by using a greedy approach).

```
findSubsequence :: Eq a => [a] -> [a] -> [Int]
findSubsequence "abcde" "abcbcdef" ⇒ [0, 1, 2, 5, 6]
findSubsequence [9,9,7] [9,7,7,9,9,7] ⇒ [0,3,5]
findSubsequence "abc" "caccdcbdca" ⇒ [1,6,8]
findSubsequence "312" "1212313" ⇒ error "subsequence does not exist"
findSubsequence [] _ ⇒ []
```

8. Define two functions that remove duplicate elements from a list. The first function, `nubRight`, considers an element to be a duplicate if it appears to the right of the first occurrence of an identical element. Conversely, `nubLeft` considers an element to be a duplicate if it appears to the left of the first occurrence of an identical element. Do not use `reverse`!

   (a) `nubRight :: Eq a => [a] -> [a]`
   `nubRight "kikiriki"` ⇒ `"kir"`

   (b) `nubLeft :: Eq a => [a] -> [a]`
   `nubLeft "kikiriki"` ⇒ `"rki"`

9. Define `median` that returns the median of a sorted list of integral elements. (The input list is considered to be sorted – you do not need to sort it yourself.) Do this as a recursive function that only traverses the list *once*. Think about how you would accomplish this. (Hint: you can traverse two copies of a list at the *same* time; this counts as only one traversal.) You can use `realToFrac` to convert integrals to fractionals.

```
median :: (Integral a, Fractional b) => [a] -> b
median [1,3,5,8,10,100] ⇒ 6.5
median [] ⇒ 0.0
```

10. An undirected graph is represented as a list of edges. An edge is represented as a pair of vertices (a, b), where $a \leq b$. A vertex is represented as an integer.

```
type Vertex = Integer
type Graph = [(Vertex,Vertex)]
```

    (a) Define `isNbrWith` that checks if two vertices are neighbours. Vertices are neighbours if there exists an edge between them.

```
isNbrWith :: Graph -> Vertex -> Vertex -> Bool
isNbrWith [(0,1),(1,2),(2,4)] 1 2 ⇒ True
isNbrWith [(0,1),(1,2),(2,4)] 0 4 ⇒ False
```

    (b) Define `areConnected` that checks if two vertices are connected. Vertices are connected if there is a path between them.

```
areConnected :: Graph -> Vertex -> Vertex -> Bool
areConnected [(0,1),(1,2),(2,4)] 0 4 ⇒ True
areConnected [(0,1),(1,2),(2,4),(5,6)] 1 5 ⇒ False
```

    Hint: Vertex a is connected to vertex b if a is a neighbour of b or any neighbour of a is connected to b. Notice a recursion here!

    Be careful not to make the recursive call on vertices you have already visited, because that would result in an infinite loop. To avoid this, you will have to maintain a list of visited vertices.

Corrections

```
v2:  2: [0..5] => [1..5]
v3: 1c: [f x,  => [x, f x,
        [1..]  => [0..]
v4: 1b: 3424 => 3423
```