Functional and Logic Programming

Home Assignment 4

Due: Saturday, 28.5.2019 - 23:55

Instructions

- Please create a source file called ***hw4.hs*** and put all the answers there.
  The file should start with a comment which contains your **full name** (in English) and **ID**

  -- John Doe

  -- 654321987

- Make sure the file **is valid** by loading it into GHCi. A
  valid file will load without any errors or warnings.

- If you need a function but you don't know how to implement it - just write it's signature (name and type) and put `undefined` in the function's body.
  That way you'll be able to load the file even though it contains references to undefined names.

- When writing a function - write both the **type** and the **body** of the function.

- Be sure to write functions with **exactly the specified name** (and **type signature** - if it is provided) for each exercise.
  You may create additional auxiliary/helper functions with whatever names and type signatures you wish.

- Try to write **small functions** which perform just **a single task**, and then **combine** them to create more complex functions.

Exercises

1. This exercise deals with infinite lists.
   a) Define **naturals-** an infinite sequence of the natural numbers
   b) Define **squares**- an infinite sequence of the natural numbers squared
   c) Define **threes**- an infinite sequence of multiplications of 3.

   Solution:

```haskell
naturals :: [Integer]
naturals = 1 : map (+1) naturals
threes :: [Integer]
threes = map (*3) naturals
squares :: [Integer]
squares = map (^2) naturals
```

   d) Define **res-**
   an infinite sequence which mixes together `naturals, squares` and `threes`.
   The first element of the list should be the first element from `naturals`
   The second element of the list should be the first element from `squares`
   The third element of the list should be the first element from `threes`
   The fourth element of the list should be the second element from `naturals`
   The fifth element of the list should be the second element from `squares`
   The sixth element of the list should be the second element from `threes`
   …

   Solution:

```haskell
interleaveThree :: [t] -> [t] -> [t] -> [t]
interleaveThree [] [] [] = []
interleaveThree (x:xs) (y:ys) (z:zs) = (x:[y,z])++interleaveThree xs ys zs
res :: [Integer]
res = interleaveThree naturals squares threes
```

d) Implement a function:
`switch :: [a] -> [a]`

which takes a list and switches between the i^{th} and the $i+1^{th}$ elements.

Note: You can assume you will be tested only on infinite lists.

<u>Solution</u>:

```
interleave :: [a] -> [a] -> [a]
interleave (x:xs) ys = x : interleave ys xs
interleave [] ys = ys

getEven :: [t] -> [t]
getEven [x] = []
getEven (x:xs:ys) = xs: getEven (ys)

getOdd :: [t] -> [t]
getOdd [x] = [x]
getOdd (x:xs:ys) = x: getOdd (ys)

switch :: [a] -> [a]
switch lst = interleave (getEven lst)  (getOdd lst)
```

2. This question deals with infinite trees.
   For the binary tree:

   data BinaryTree a = Nil | BNode a (BinaryTree a) (BinaryTree a)

   a) Define the function:
      `infTree` **:: a -> BinaryTree a**
      which produces a full, symmetric, infinite, binary tree of a's.

      Solution:

```haskell
infTree :: a -> BinaryTree a
infTree x = BNode x (infTree x) (infTree x)
```

   b) Define the function:
      `treeMap` **:: (a -> b) -> BinaryTree a -> BinaryTree b**

      Which takes a function and a binary tree, and produces a binary tree in which all nodes are the result of applying the function on the given tree.

      Solution:

```haskell
treeMap :: (a -> b) -> BinaryTree a -> BinaryTree b
treeMap _ Nil          = Nil
treeMap f (BNode x l r) = BNode (f x) (treeMap f l) (treeMap f r)
```

   c) Define:
      type Depth = **Int**

      Define the function:
      `treeTake` :: **Depth -> BinaryTree a -> BinaryTree a**

      Which prunes the given tree to produce a tree with at most "depth" levels.

      Solution:

```
treeTake :: Depth -> BinaryTree a -> BinaryTree a
treeTake 0 _      = Nil
treeTake _ Nil = Nil
treeTake n (BNode x l r) = BNode x (treeTake (n-1) l) (treeTake (n-1) r)
```

d) Define the function:

`treeSort` **:: BinaryTree t -> [t]**

Which takes a binary search tree and outputs a sorted list of the tree's values.

Solution:

```
treeSort :: BinaryTree t -> [t]
treeSort Nil = []
treeSort(BNode n Nil Nil) = [n]
treeSort(BNode n tl tr) = (treeSort tl) ++ [n] ++ (treeSort tr)
```