Programming in Haskell – Homework Assignment 6

UNIZG FER, 2017/2018

Handed out: January 11, 2018. Due: January 19, 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 ($\star$) 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 ($\star$) 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)* Define a recursive data type `Pred` that represents a boolean expression.

   Use the `type` constructors `And`, `Or` and `Not` to represent boolean operations and the `Val` constructor to represent a boolean value. Implement the `eval` function that takes a `Pred` and returns its evaluated `Bool` value.

   ```
   expr = And (Or (Val True) (Not (Val True))) (Not (And (Val True) (Val False)))
   ```

   ```
   eval (Val True) ⇒ True
   eval (Val False) ⇒ False
   eval (Or (Val True) (Val False)) ⇒ True
   eval expr ⇒ True
   eval (Not expr) ⇒ False
   ```

2. *(3 point)* In this exercise, you will take a description of a meetup date, and return the actual meetup date. Typically meetups happen on the same day of the week.

   Examples of general descriptions are:

   - the first Monday of January 2017
   - the third Tuesday of January 2017
   - the Wednesteenth of January 2017
   - the last Thursday of January 2017

   Note that "Monteenth", "Tuesteenth", etc are all made up words. There was a meetup whose members realized that there are exactly 7 numbered days in a month that end in "-teenth". Therefore, one is guaranteed that each day of the week (Monday, Tuesday, ...) will have exactly one date that is named with "-teenth" in every month.

   Given examples of a meetup dates, each containing a month, day, year, and descriptor (first, second, teenth, etc), calculate the date of the actual meetup. For example, if given "First Monday of January 2017", the correct meetup date is 2017/1/2.

   Write the following function:

   ```
   dateFromDescription ::  String -> Day
   ```

   Where `Day` is a data type from the `time` package.

   You may want to take a look at `time` package documentation (`Calendar` module in particular) on Hackage. It should come by default with Haskell platform, but in case you don't have it, you can install it with stack or cabal depending on what you use.

   If you are running your scripts with `ghci` than you should run `cabal install time`, and if you are using `stack repl` than you can install it with `stack install time`.

3. *(3 points)* Implement the following functions over a `Tree` data type defined as:

```
data Tree a = Leaf | Node a (Tree a) (Tree a) deriving (Eq, Show)
testTree = Node 1 (Node 2 Leaf Leaf) (Node 3 (Node 4 Leaf Leaf) Leaf)
```

   (a) *(1 point)* Define a function `treeFilter` that takes a predicate and a `Tree` and removes those subtrees that do not satisfy the given predicate (with any children).

```
treeFilter ::  (a -> Bool) -> Tree a -> Tree a
```

   Examples:

```
treeFilter (const True) testTree ⇒ testTree
treeFilter odd testTree ⇒ Node 1 Leaf (Node 3 Leaf Leaf)
treeFilter (<3) testTree ⇒ Node 1 (Node 2 Leaf Leaf) Leaf
```

   (b) *(1 point)* Define a function `levelMap` that takes some binary function and applies it to the tree. The function that is being applied takes the depth of the tree as the first argument. The root element's depth is 0.

```
levelMap ::  (Int -> a -> b) -> Tree a -> Tree b
```

   Examples:

```
levelMap (␣lx -> if odd l then x else x + 1) testTree ⇒ Node 2 (Node
2 Leaf Leaf) (Node 3 (Node 5 Leaf Leaf) Leaf)

levelMap const testTree ⇒ Node 0 (Node 1 Leaf Leaf) (Node 1 (Node
2 Leaf Leaf) Leaf)
```

   (c) *(1 point)* Define a function `isSubtree` that takes two instances of `Tree` and checks whether the first tree appears as part of the second.

```
isSubtree ::  Eq a => Tree a -> Tree a -> Bool
```

   Examples:

```
isSubtree (Node 4 Leaf Leaf) testTree ⇒ True
isSubtree (Node 3 Leaf Leaf) testTree ⇒ False
isSubtree testTree testTree ⇒ True
isSubtree Leaf testTree ⇒ True
```

4. *(4 points(⋆))* We need to import new categories in our web shop. Unfortunately, they aren't very well structured and thus are not very well suited for further processing. Every category is defined as a string representing a full path to that category. For example, string `Category 01 > SubCategory 01 > SubSubCategory 02` represents single category called `SubSubCategory 02` which is 2 levels deep in category hierarchy.

   Define data type `Category` which contains name of that category and list of all of its children categories.

   Define a function which will go through list of category strings and create appropriate hierarchical data structure containing all root categories and their children:

   ```
   parseCategories ::  [String] -> [Category]
   ```

   Define a function which will convert list of categories back to a list of strings representing full path to a certain category:

   ```
   printCategories ::  [Category] -> [String]
   ```

   One thing to note here is that not every possible path will necessarily be defined as input but you should create all categories on that path regardless. What this means is that `paths /= (printCategories $ parseCategories paths)`, but if you execute `printCategories $ parseCategories ["Computers > Equipment > Mouse pads"]` you should get a following result:

   ```
   [ "Computers"
   , "Computers > Equipment"
   , "Computers > Equipment > Mouse pads"
   ]
   ```

   For bonus bonus points *(2 pts)*, define a function which loads initial list of categories from an external text file and than parses them.

Corrections