Programming in Haskell – Homework Assignment 7

UNIZG FER, 2013/2014

Handed out: December 15, 2013. Due: December 23, 2013 at 23:59

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. Let us define some data types:

   ```
   data Date = Date Int Int Int
   ```

   ```
   data Animal = Animal {
       species   :: String,
       name      :: String,
       legNum    :: Maybe Int, -- Will be Nothing if animal has no legs
       birthday  :: Date,
       dangerLvl :: Int }
   ```

   ```
   testDog = Animal "Dog" "Fluffy" (Just 4) (Date 16 3 1975) 10
   ```

   Now, let's define some functions for our newly created type.

   (a) Define `avgLegNum` to calculate the average number of legs of given animals:

   ```
   avgLegNum :: [Animal] -> Double
   avgLegNum [testDog] ⇒ 4
   avgLegNum [] ⇒ error "empty list"
   ```

   (b) Define `canDrinkBeer` that returns the names of all animals older than 18 years:

   ```
   canDrinkBeer :: [Animal] -> [String]
   canDrinkBeer ::  [testDog] ⇒ ["Fluffy"]
   canDrinkBeer ::  [] ⇒ []
   ```

   (c) Define `getFakeID` that makes an animal older for a given number of years:

   ```
   getFakeID :: Animal -> Int -> Animal
   getFakeID testDog 75 ⇒ testDog {birthday = Date 16 3 1900}
   ```

2. Define a new data type `BinaryTree` and the following functions upon it:

   ```
   data BinaryTree a = Null | Node a (BinaryTree a) (BinaryTree a)
   ```

   ```
   testTree = Node 1 (Node 2 (Node (-4) Null Null) (Node 3 Null Null))
              (Node 2 (Node 1 Null (Node 10 Null (Node (-2) Null Null))) Null)
   ```

(a) `numNodes :: BinaryTree a -> Int`
`numNodes testTree` ⇒ `8`
`numNodes Null` ⇒ `0`

(b) Define `averageNodeDegree` to compute the degree of a node, where a node degree is defined as a number of children the node has.

`averageNodeDegree :: BinaryTree a -> Double`
`averageNodeDegree testTree` ⇒ `7/8 = 0.875`
`averageNodeDegree Null` ⇒ `0`

(c) Define `treeDepth` that returns the depth of the deepest node (root has depth of 1).

`treeDepth :: BinaryTree a -> Int`
`treeDepth testTree` ⇒ `5`
`treeDepth Null` ⇒ `0`

(d) Define a function `preorder` to convert a binary tree into a list by a preorder traversal.

`preorder:: BinaryTree a -> [a]`
`preorder testTree` ⇒ `[1, 2, -4, 3, 2, 1, 10, -2]`
`preorder Null` ⇒ `[]`

(e) Define a function `inorder` to convert a binary tree into a list by an inorder traversal.

`inorder :: BinaryTree a -> [a]`
`inorder testTree` ⇒ `[-4, 2, 3, 1, 1, 10, -2, 2]`
`inorder Null` ⇒ `[]`

(f) Define a function `postorder` to convert a binary tree into a list by a postorder traversal.

`postorder :: BinaryTree a -> [a]`
`postorder testTree` ⇒ `[-4, 3, 2, -2, 10, 1, 2, 1]`
`postorder Null` ⇒ `[]`

3. Implement a bounded list. Such a list cannot hold more than a specified number of elements.

(a) Define an appropriate data structure.

```
data BList a = ...
type Limit = Int
```

(b) Define a function to create an empty bounded list, explicitly providing an initial limit.

`empty :: Limit -> BList a`

(c) Define a function to create a bounded list from a regular list. Implicitly set the limit to the length of the input list.

`fromList :: [a] -> BList a`

(d) Modify the list's limit. If the resulting list contains more elements than its limit, silently strip the extra elements from the end of the list.

`limited :: Limit -> BList a -> BList a`
`limited 1 $ fromList [1,2,3]` ⇒ `fromList [1]`

(e) Add a single element to the front of the list. Return an error if the resulting list contains more elements than it is allowed to.

```
cons :: a -> BList a -> BList a
cons 1 $ fromList [1,2,3] ⇒ error "too many elements!"
cons 1 $ limited 10 $ fromList [2] ⇒ limited 10 $ fromList [1,2]
```

(f) Concatenate multiple bounded lists into one. The result should be bounded at n, where n is the sum of the bounds of all concatenated bounded lists.

```
concat' :: [BList a] -> BList a
concat' $ map fromList [[1],[2,3]] ⇒ fromList [1,2,3]
```

4. Define a recursive datatype `Expr` that represents an arithmetic expression possibly containing variables. Use data constructors called `Add`, `Sub`, `Mul` and `Div` to represent the corresponding arithmetic operators. Use the `Val` constructor to represent the real numbers (use `Double`) and `Var` to represent the variables (use `String`).

(a) Define the datatype.

```
data Expr = ...
let e = Add (Var "x") (Mul (Val 2.0) (Add (Var "y") (Val 7.5)))
```

(b) Define a function that shows the expression in `String` form. Return as few parentheses as possible (without reshuffling the `Expr` value). For instance, while showing a `Val (-4)` as `"(-4)"` is okay, showing a `Val 4` as `"(4)"` is not.

```
showExpr :: Expr -> String
showExpr e ⇒ "x+2*(y+7.5)"
showExpr (Mul (Mul (Var "x") (Val 2)) (Val 3)) ⇒ "x*2*3"
showExpr (Div (Val 1) (Val 2)) ⇒ "1/2"
showExpr (Div (Val 1) (Sub (Val 2) (Val 3))) ⇒ "1/(2-3)"
```

(c) A function that substitutes each variable of a certain name in an expression with a given expression. If no such variable exists, the function returns an unmodifed expression.

```
subst :: String -> Expr -> Expr -> Expr
showExpr $ subst "x" (Val 100) e ⇒ "100+2*(y+7.5)"
showExpr $ subst "z" (Val 100) e ⇒ "x+2*(y+7.5)"
```

(d) Define a function that evaluates an expression, given a mapping of variables to values. The mapping is represented using the `Data.Map.Map` structure. You can import `Data.Map` in two ways:

i. `import Data.Map`
This lets us use `Data.Map` functions directly, but the problem is that their names clash with the names of `Prelude` functions. To avoid this, we usually don't import `Data.Map` this way.

ii. `import qualified Data.Map as M`
This lets us use all the `Data.Map` functions, but only when we qualify them with the `M` prefix. We will import `Data.Map` this way. For instance:
`M.insert (3,4) $ M.empty => M.fromList [(3,4)]`

See the documentation for a list of `Data.Map` functions we can use. Functions you might find useful are `M.fromList`, `M.lookup` and `(M.!)`.

So, after importing `Data.Map` qualified, this is our mapping:

```
type VarAssignments = M.Map String Double
```

And this is the function you need to define:

```
eval :: VarAssignment -> Expr -> Maybe Double
```

The function should return a `Nothing` if the expression contains a variable with no known assignment.

```
let vars = M.fromList [("x",10),("y",2.5)]
eval vars e ⇒ Just 30
eval M.empty e ⇒ Nothing
```

5. Implement a prefix tree, also known as a *trie*. Use it to store a mapping between a list `[k]` and an arbitrary value `a`. We can define a trie recursively, as follows:

```
data Trie k a = Leaf a | Branch [(Maybe k, Trie k a)]
```

We can represent an empty tree with `Branch []`. Similarly, a tree containing the items `("more", 1) and ("most", 2)` would look like this:

```
Branch [
  (Just 'm', Branch [
    (Just 'o', Branch [
      (Just 'r', Branch [
        (Just 'e', Branch [
          (Nothing, Leaf 1)])]),
      (Just 's', Branch [
        (Just 't', Branch [
          (Nothing, Leaf 2)])])])])]
```

We assume no key can be a prefix of another key. Each node can be either a branch or a leaf, but not both. To ensure this, we terminate each key with a special value (e.g., `Nothing`).

Using such a structure, define the following functions.

(a) An empty trie value.

```
empty' :: Trie k a
```

(b) A function to insert an item into the trie.

```
insert' :: Eq k => [k] -> a -> Trie k a -> Trie k a
insert' "more" 1 $ insert' "most" 2 empty' ⇒ (the trie above)
```

(c) A function to create a trie from a list of items. (Hint: use `insert'` and a fold function.)

```
fromList' :: Eq k => [([k], a)] -> Trie k a
fromList' [("more",1),("most",2)] => (the trie above)
```

(d) A function to retrieve a value for a given key, if it exists. Otherwise return `Nothing`.

```
lookup' :: Eq k => [k] -> Trie k a -> Maybe a
lookup' "more" $ fromList' [("more",1)] ⇒ Just 1
lookup' "meow" $ fromList' [("most",2)] ⇒ Nothing
```

(e) A function to delete a given key (and the value associated with it) from the trie. If the key doesn't exist, return the unmodified trie.

```
delete' :: Eq k => [k] -> Trie k a -> Trie k a
delete' "more" $ fromList' [("more",1)] ⇒ empty'
```

Corrections

```
v2:  5: Replace the special '$' character with the Nothing value.
v3:     Extended deadline to Monday.
v4:     Add Eq constraint to fromList'. Fix typos.
v5: 3d: Remove faulty fromList example.
v6: 3e: Modify faulty fromList example.
```