COMP 141: Haskell — Part 8

Instructions: In this exercise, we are going to review a bunch of Haskell structures.

(1) Data type `Day`:

(a) Define a data type named `Day` consisting of all days in a week.

(b) Define a function `workday` which receives a value of type `Day` as input, and returns `True` if the day is a weekday (i.e., Monday through Friday). Otherwise, it should return `False`.

(2) Data type `Dog`:

(a) Define data type `Dog` with a single value constructor that receives two strings and an integer as the dog's name, breed, and age respectively.

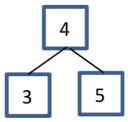
(b) Define a function `breed` that receives a `Dog` and returns the string that represents the breed component.

(3) Define the aforementioned data type `Dog` as a *record*.

(4) Recall the definition for `Maybe` data type:
```
data Maybe a = Nothing | Just a
```
Define your own version of tail function that receives a list (of any type) and returns the tail of the list. Your tail function must return `Nothing` in case the input is an empty list. Therefore, the type of function must be `[a] -> Maybe [a]`.

(5) Recall the definition of binary tree data type:
```
data Btree a = Emptree | Node a (Btree a) (Btree a) deriving (Show, Read, Eq, Ord)
```

(a) Show how to create a binary tree which is equivalent to the tree pictured here.



(b) Define function `multree` that that receives a binary tree of double numbers (`Double`) and multiplies all the numbers within the nodes. (You may assume that for empty trees, the result is 1.0). Be explicit about the typing.

(c) Define function `makeBST :: (Ord a) => [a] -> Btree a` that reads a list of comparable items and puts them in a binary search tree. *Hint 1*: You can use `treeInsert` function introduced in the class. *Hint 2*: You can define it either by recursion on input list, or using folding functions.

(6) Record type for a state:

(a) Define a data type `State` consisting of three components:

- name which is a string,
- population which is potentially large integer, and
- capital which is a string.

(b) Create a record for California according to the data type you defined above.

(7) Define function `preorder :: Btree a -> [a]` that receives a binary tree and returns the pre-order traversal of it. Use pattern matching on binary trees. Do not use any existing library functions associated with trees in Haskell libraries.

Example: If the input is `(Node 5 (Node 6 Emptree (Node 7 Emptree Emptree )) (Node 8 Emptree Emptree))` then the output would be `[5,6,7,8]`

(8) Define function `heightTree :: Btree a -> Int` that computes the height of a binary tree. Use pattern matching on binary trees. Do not use any existing library functions associated with trees in Haskell libraries.

(9) Define function `minimumTree :: (Ord a) => Btree a -> a` that returns the minimum value within a binary tree. Use pattern matching on binary trees. Do not use any existing library functions associated with trees in Haskell libraries.

(10) Define function `maximumTree :: (Ord a) => Btree a -> a` that returns the maximum value within a binary tree. Use pattern matching on binary trees. Do not use any existing library functions associated with trees in Haskell libraries.

(11) Define function `isBST :: (Ord a) => Btree a -> Bool` that returns true if the input binary tree is a binary search tree (BST). Otherwise it returns false. Use pattern matching on binary trees. Do not use any existing library functions associated with trees in Haskell libraries.

Reminder: BST has an ordering property that any node's left subtree keys $\leq$ the node's key, and the right subtree's keys $\geq$ the node's key.

Hints: You can approach the definition in two ways (your choice!):

- Make sure that *every* node is greater than `maximumTree` of its left subtree and less than `minimumTree` of its right subtree.
- Create a list out of the in-order traversal of the binary tree and check if the list is in ascending order.

(12) Tree with arbitrary number of subtrees.

(a) Define a data type for trees with arbitrary number of subtrees. Name the data type `Tree`. Moreover, name the empty tree and the value constructor for internal nodes as `EmptyTree` and `NodeTree`, respectively.

(b) Define the name `mytree` that corresponds to the following tree.