CMPT 383 Comparative Programming Languages

Homework 3

This homework is due by 11:59pm PT on Wednesday Feb 9, 2022. No late submission is accepted. Please save your Haskell code in a single file called `h3_firstname_lastname.hs` (`h` in lower case, firstname and lastname replaced with your first and last name) and submit it to Canvas.

Requirements of this homework:

- Write type signatures for all functions using the :: operator.
- Do not use the `if-then-else` expression unless specified in the question.

1. (20 points) Define a `List` type (or type constructor) with data constructors `Empty` and `Cons` to represent lists. For example, the standard notation `[1]` should be represented as `Cons 1 Empty`. Write a function `listZip` that simulates the standard `zip` on two `List`s.

Sample input and output:

```
ghci> listZip (Cons 1 (Cons 2 Empty)) (Cons 3 Empty)
Cons (1,3) Empty
ghci> listZip (Cons 1 (Cons 2 Empty)) (Cons 'a' (Cons 'b' Empty))
Cons (1,'a') (Cons (2,'b') Empty)
```

2. (20 points) A binary search tree is a binary tree where each node has a value that is greater than all values in the left subtree and less than all values in the right subtree. Define a `Tree` type (or type constructor) with data constructors `EmptyTree` and `Node` to represent binary search trees. Write a function `insert` that takes an element v and a binary search tree t and produces a binary search tree with v inserted into t. You can assume the elements (in the binary search tree and the one to insert) are unique.

Sample input and output:

```
ghci> insert 'a' (Node 'b' EmptyTree EmptyTree)
Node 'b' (Node 'a' EmptyTree EmptyTree) EmptyTree
ghci> insert 5 (Node 3 (Node 1 EmptyTree EmptyTree) (Node 6 EmptyTree EmptyTree))
Node 3 (Node 1 EmptyTree EmptyTree) (Node 6 (Node 5 EmptyTree EmptyTree) EmptyTree)
```

3. (20 points) Define a `Nat` type with data constructors `Zero` and `Succ` to represent natural numbers by zero and its successors. For example, 1 should be represented as `Succ Zero`. As another example, 3 should be represented as `Succ (Succ (Succ Zero))`. Write two functions `natPlus` and `natMult` that perform addition and multiplication of `Nat`'s, respectively. Hint: $(m+1) \cdot n = m \cdot n + n$.

Sample input and output:

```
ghci> natPlus (Succ Zero) (Succ Zero)
Succ (Succ Zero)
ghci> natPlus (Succ (Succ Zero)) (Succ Zero)
Succ (Succ (Succ Zero))
ghci> natMult (Succ Zero) Zero
Zero
ghci> natMult (Succ (Succ Zero)) (Succ (Succ Zero))
Succ (Succ (Succ (Succ Zero)))
```

4. (20 points) Consider the `Tree` in Question 2 again. Make `Tree a` an instance of the `Eq` type class without using `deriving (Eq)`. You can assume the values in the tree to compare always have the same type.

Sample input and output:

```
ghci> let t1 = (Node 2 (Node 1 EmptyTree EmptyTree) (Node 3 EmptyTree EmptyTree))
ghci> let t2 = (Node 2 (Node 1 EmptyTree EmptyTree) (Node 3 EmptyTree EmptyTree))
ghci> let t3 = (Node 2 (Node 1 EmptyTree EmptyTree) EmptyTree)
ghci> t1 == t2
True
ghci> t1 == t3
False
```

5. (20 points) Suppose we have an association list defined in the following way

```
data AssocList k v = ALEmpty | ALCons k v (AssocList k v) deriving (Show)
```

which conceptually represents a list of key-value pairs. Here, `k` is the key type and `v` is the value type. It has two data constructors: `ALEmpty` denotes the empty list, and `ALCons` denotes the list cons. For example, the standard notation `[(1, 2), (3, 4)]` becomes `ALCons 1 2 (ALCons 3 4 ALEmpty)` in `AssocList`.

In this question, you need to make `AssocList k` a functor, where the `fmap` function applies the given function to all values in the association list. Please note that you also need to explicitly write down the type signature of `fmap` for `AssocList k`.

Sample input and output:

```
ghci> fmap (+1) (ALCons 1 2 (ALCons 3 4 ALEmpty))
ALCons 1 3 (ALCons 3 5 ALEmpty)
ghci> fmap (*2) (ALCons 'a' 1 (ALCons 'b' 2 ALEmpty))
ALCons 'a' 2 (ALCons 'b' 4 ALEmpty)
```