COSC 3015 - Functional Programming September 16, 2021
Finley McIlwaine

Homework #3

Due before class on September 28, 2021

For your submission on WyoCourses, turn in a file named `hw3.hs` containing your Haskell code.

Problem 1. (5 points) Define a function:

```
oscMap :: (a -> b) -> (a -> b) -> [a] -> [b]
```

that takes two functions of the same type and applies each function to every other element of the list in turn. In other words, it *oscillates* between the mapping functions it applies for successive elements. For example:

```
> oscMap (+10) (+100) [1,2,3,4]
[11,102,13,104]
> oscMap (1-) (1+) [1,2,3,4,5]
[0,3,-2,5,-4]
```

Problem 2. (5 points) Copy the following data type definition for polymorphic lists into your Haskell script for this assignment:

```
data List a = Nil | Cons a (List a) deriving Show
```

Now define a function:

```
app :: List a -> List a -> List a
```

that appends two lists in the type we've defined above. For example:

```
> app (Cons 1 (Cons 2 Nil)) (Cons 3 Nil)
Cons 1 (Cons 2 (Cons 3 Nil))
> app Nil Nil
Nil
```

```
> app (Cons 3 Nil) (Cons 4 (Cons 5 (Cons 6 Nil)))
Cons 3 (Cons 4 (Cons 5 (Cons 6 Nil)))
```

Problem 3. (10 points) Using `foldr`, define a function:

```
list2int :: [Int] -> Int
```

that takes a list of non-negative integers and converts it into an integer as in the following examples:

```
> list2int [1,2,3]
123
> list2int [0,1,2,3]
123
> list2int [0,1,2,3,0]
1230
> list2int [0,1,2,3,0,4,5]
123045
> list2int [0,0,0,0]
0
```

Hint: Your solution may require your `foldr` function to be composed with other functions. A function that may be useful for your definition is `iterate :: (a -> a) -> a -> [a]`. By definition, `iterate f x` returns an infinite list of repeated applications of `f` to `x`:

```
iterate f x = [x, f x, f (f x), f (f (f x)), ...]
```

Problem 4. Consider the following data type whose values represent *binary* trees:

```
data Tree a = Leaf | Node a (Tree a) (Tree a) deriving Show
```

A value of type `Tree a` is therefore either a `Leaf` containing no value, or binary node containing some value of type `a` and having two similarly typed children trees. Copy the type declaration into your Haskell script for this assignment and complete the following:

Problem 4.a. (5 points) Define a function:

```
size :: Tree a -> Int
```

that takes a tree and returns the number of values stored in the tree. For example:

```
> size Leaf
0
> size (Node 10 Leaf Leaf)
1
> size (Node 10 (Node 8 Leaf Leaf) (Node 20 (Node 15 Leaf Leaf) Leaf))
4
```

Problem 4.b. (5 points) Define a function:

```
insert :: Ord a => a -> Tree a -> Tree a
```

that takes a value and a tree that is assumed to be "sorted", i.e. a *binary search tree* and returns the same tree but with a new node containing the value added to it in the appropriate spot. Your definition should meet these specifications:

- Inserting a value into a leaf results in a binary node containing the value with two children leaves.
- Inserting a value into a binary node should insert the value into the left child tree if the value is $\leq$ the node's value, and the right child tree otherwise.

For example:

```
> insert 10 (Leaf)
Node 10 Leaf Leaf
> insert 8 (Node 10 Leaf Leaf)
Node 10 (Node 8 Leaf Leaf) Leaf
> insert 15 (Node 10 (Node 8 Leaf Leaf) Leaf)
Node 10 (Node 8 Leaf Leaf) (Node 15 Leaf Leaf)
```

Problem 4.c. (5 points) Define a function:

```
squash :: Tree a -> [a]
```

that takes a tree and returns a list containing the values stored in the tree in *inorder* order. For example:

```
> squash Leaf
[]
> squash (Node 10 Leaf Leaf)
[10]
```

```
> squash (Node 10 (Node 8 Leaf Leaf) Leaf)
[8,10]
> squash (Node 10 (Node 8 Leaf Leaf) (Node 15 Leaf Leaf))
[8,10,15]
```

Problem 4.d. (5 points) Using `foldr`, define a function:

```
unsquash :: Ord a => [a] -> Tree a
```

that constructs a *binary search tree* from a list. For example:

```
> unsquash []
Leaf
> unsquash [10]
Node 10 Leaf Leaf
> unsquash [8,10]
Node 10 Leaf (Node 8 Leaf Leaf)
> unsquash [1,3,2,10]
Node 10 (Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf)) Leaf
```

Problem 4.e. (5 points) Define a function:

```
treesort :: Ord a => [a] -> [a]
```

that sorts a list by building a *binary search tree* out of it and then squashing it back into a list.

Problem 4.f. (10 points) Define a function:

```
foldt :: (a -> b -> b -> b) -> b -> Tree a -> b
```

that computes a fold with values of the `Tree a` type we've been working with. The fold should essentially replace `Leaf`s in the tree with some given "base" or "initial" value, and replace `Node`s in the tree with the given operator.

Problem 4.g. (5 points) Using the `foldt` function from above, define a function:

```
treesum :: Num a => Tree a -> a
```

that computes the sum of all numbers stored in a tree. For example:

```
> treesum Leaf
0
> treesum (Node 3 Leaf Leaf)
3
> treesum (Node 10 (Node 3 Leaf (Node 8 Leaf Leaf)) (Node 7 Leaf Leaf))
28
```

Problem 4.h. (10 points) Make the `Tree a` type we've been working with an instance of the `Eq` type class by declaring it as an instance and defining either the `==` or `/=` functions such that two trees are equal only if their `treesum`s are equal. For example:

```
> Leaf == Leaf
True
> (Node 3 Leaf Leaf) == (Node 2 (Node 1 Leaf Leaf) Leaf)
True
> (Node 3 Leaf Leaf) == (Node 2 (Node 2 Leaf Leaf) Leaf)
False
> (Node 3 Leaf Leaf) /= (Node 2 (Node 2 Leaf Leaf) Leaf)
True
```

EXTRA CREDIT Problem 4.i. (5 points) Using `foldt`, define a function:

```
depth :: Tree a -> Int
```

that computes the depth of a binary tree. For example:

```
> depth Leaf
0
> depth (Node 10 Leaf Leaf)
1
> depth (Node 10 Leaf (Node 10 Leaf Leaf))
2
> depth (Node 10 Leaf (Node 10 Leaf (Node 3 Leaf Leaf)))
3
> depth (Node 10 Leaf (Node 10 (Node 3 Leaf Leaf) (Node 3 Leaf Leaf)))
3
```