Assignment 5

due at 23:59 on Wed Oct 11 (80 points)

Save all your functions into one file called `a05.hs` and submit it to this dropbox.

Higher-order functions

`mapFirstRest`

Define the function `mapFirstRest`, with this signature:

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

It works like `map`, in that it applies a function to each element of a list. But unlike `map` it takes two separate function parameters, and applies the first function to the first list element, and the second function to the *rest* of the list. Here are some examples:

```
ghci> mapFirstRest (+5) (*5) [7..10]
[12,40,45,50]
ghci> mapFirstRest (+5) (*5) [8,8,8]
[13,40,40]
ghci> mapFirstRest length (const 0) ["Alice", "Bob", "Carol"]
[5,0,0]
ghci> mapFirstRest (+5) id [9,9,9]
[14,9,9]
ghci> mapFirstRest id (+5) [9,9,9]
[9,14,14]
```

In the above, we're using `id` which is the identity function. It just returns its argument exactly as given:

```
ghci> id 9
9
ghci> id "Hello"
"Hello"
```

`capitalize`

Create a function `capitalize` that will convert the first letter of its string argument to upper-case.

```haskell
capitalize :: String -> String
```

To do that, you should import `Data.Char` which contains a function `toUpper` that works on a single character:

```
ghci> import Data.Char
ghci> toUpper 'h'
'H'
ghci> toUpper 'G'
'G'
ghci> toUpper '%'
'%'
```

Use `mapFirstRest` to apply `toUpper` appropriately. Here are some sample inputs and outputs:

```
ghci> capitalize "hello"
"Hello"
ghci> capitalize "nifty"
"Nifty"
ghci> capitalize "NICE"
"NICE"
ghci> capitalize "#wow"
"#wow"
```

`maybeCapitalize`, `titleCase`

Title case is the name for the capitalization rules typically used in titles and headlines, where every word is capitalized except articles (`the`, `a`), conjunctions (`and`, `but`, `for`), and prepositions (`in`, `of`). But the first word of the sentence is capitalized even if it *is* one of these words.

Define these in your solution:

```haskell
exemptions :: [String]
maybeCapitalize :: String -> String
titleCase :: String -> String
```

The `exemptions` should be a list of strings representing words that are exempt from capitalization in titles:

```
ghci> "the" `elem` exemptions
True
ghci> "dog" `elem` exemptions
False
```

The `maybeCapitalize` is a function that capitalizes only if the word is not exempt. If it is exempt, then it just returns the word unchanged. In your definition, you should use both `exemptions` and `capitalize`.

```
ghci> maybeCapitalize "the"
"the"
ghci> maybeCapitalize "dog"
"Dog"
```

Finally, `titleCase` is a function that takes a complete phrase, splits it into words, and applies capitalization appropriately. Your implementation should use `capitalize`, `maybeCapitalize`, `mapFirstRest`, `words`, `unwords`, and function composition.

The built-in functions `words` and `unwords` are great for disassembling a phrase into a list of words, and then putting it back together again:

```
ghci> words "the quick brown fox"
["the","quick","brown","fox"]
ghci> unwords ["jumps", "over", "the", "lazy", "dog."]
"jumps over the lazy dog."
```

Here are some examples for `titleCase`:

```
ghci> titleCase "the quick brown fox jumps over the lazy dog"
"The Quick Brown Fox Jumps Over the Lazy Dog"
ghci> titleCase "the hound of the baskervilles"
"The Hound of the Baskervilles"
ghci> titleCase "harry potter and the chamber of secrets"
"Harry Potter and the Chamber of Secrets"
```

Tree traversal

Using the definition of `Tree` from the notes on 5 October, define traversals that concatenate together all the values in the specified order. Your functions will require that the type of values stored in the tree forms a monoid.

```
preorder :: Monoid a => Tree a -> a
inorder :: Monoid a => Tree a -> a
postorder :: Monoid a => Tree a -> a
```

At each node:

- pre-order outputs the node first, then left, then right
- in-order outputs left, then the current node, then right

- post-order outputs left, then right, then finally the current node

Here are traversals on the tree defined `sample1` in the notes (and a reminder of what that tree contains).

```
ghci> printTree sample1
- "A"
  |- "K"
  |  |- "M"
  |  |  |- *
  |  |  |- "Q"
  |  |- "P"
  |- *
ghci> preorder sample1
"AKMQP"
ghci> inorder sample1
"MQKPA"
ghci> postorder sample1
"QMPKA"
```

Stack monoid

Using the `BoundedStack` data type from the Assignment 4 solutions, define it as an instance of `Monoid`. Unlike the tree, it should **not** require that the element type is also a `Monoid`. Here is the first line of the instance definition:

```
instance Monoid (BoundedStack a) where
```

and then you add the definitions of `mempty` and `mappend`.

To append two stacks, you should combine together their capacities and place all the elements in the left stack before all the elements in the right stack. Here are some examples:

```
ghci> Just s1 = push 7 (new 3)
ghci> Just s2 = (push 8 >=> push 9) (new 2)
ghci> s1 <> s2
BoundedStack {capacity = 2, elements = [7,9,8]}
ghci> s2 <> mempty
BoundedStack {capacity = 0, elements = [9,8]}
ghci> mempty <> s1
BoundedStack {capacity = 2, elements = [7]}
```

Remember that <> is an operator shorthand for `mappend`.

Test code

```
import Data.Monoid
import Data.Char
import Control.Monad.State

main = do
  flip execStateT (0,0) $ do
    -- mapFirstRest
    verify "1.01" [12,40,45,50] $ mapFirstRest (+5) (*5) [7..10]
    verify "1.02" [13,40,40]    $ mapFirstRest (+5) (*5) [8,8,8]
    verify "1.03" [5,0,0]       $ mapFirstRest length (const 0)
                                  ["Alice", "Bob", "Carol"]
    verify "1.04" [14,9,9]      $ mapFirstRest (+5) id [9,9,9]
    verify "1.05" [9,14,14]     $ mapFirstRest id (+5) [9,9,9]
    -- capitalize
    verify "2.01" "Hello" $ capitalize "hello"
    verify "2.02" "Nifty" $ capitalize "nifty"
    verify "2.03" "NICE"  $ capitalize "NICE"
    verify "2.04" "#wow"  $ capitalize "#wow"
    -- maybeCapitalize
    assert "3.01" $ "the" `elem` exemptions
    assert "3.02" $ not $ "dog" `elem` exemptions
    verify "3.03" "the" $ maybeCapitalize "the"
    verify "3.04" "Dog" $ maybeCapitalize "dog"
    verify "3.05" "The Quick Brown Fox Jumps Over the Lazy Dog"
      $ titleCase "the quick brown fox jumps over the lazy dog"
    verify "3.06" "The Hound of the Baskervilles"
      $ titleCase "the hound of the baskervilles"
    verify "3.07" "Harry Potter and the Chamber of Secrets"
      $ titleCase "harry potter and the chamber of secrets"
    -- tree traversal
    verify "4.01" "AKMQP"  $ preorder  sample1
    verify "4.02" "MQKPA"  $ inorder   sample1
    verify "4.03" "QMPKA"  $ postorder sample1
    verify "4.04" "BSCDFR" $ preorder  sample2
    verify "4.05" "SDCFBR" $ inorder   sample2
    verify "4.06" "DFCSRB" $ postorder sample2
    -- stack monoid
    let Just s1 = push 7 (new 3)
    let Just s2 = (push 8 >=> push 9) (new 2)
    verify "5.01" (BoundedStack {capacity = 2, elements = [7,9,8]})
      $ s1 <> s2
    verify "5.02" s2 $ s2 <> mempty
    verify "5.03" s1 $ mempty <> s1
```

```haskell
  where
    say = liftIO . putStrLn
    correct (k, n) = (k+1, n+1)
    incorrect (k, n) = (k, n+1)
    assert s = verify s True
    verify :: (Show a, Eq a) => String -> a -> a -> StateT (Int,Int) IO ()
    verify = verify' (==)
    verifyF = verify' (\x y -> abs(x-y) < 0.00001)
    verify' :: (Show a) => (a -> a -> Bool) -> String -> a -> a ->
              StateT (Int,Int) IO ()
    verify' eq tag expected actual
      | eq expected actual = do
          modify correct
          say $ " OK " ++ tag
      | otherwise = do
          modify incorrect
          say $ "ERR " ++ tag ++ ": expected " ++ show expected
            ++ " got " ++ show actual
-- End of test driver
```