# Assignment 5 solutions

## Test driver

```haskell
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
  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
```

## Higher-order functions

### mapFirstRest

Note that this doesn't need to be recursive! It just distinguishes between an empty list and a non-empty list, then uses `f` on first and `map g` on rest.

```
mapFirstRest :: (a -> b) -> (a -> b) -> [a] -> [b]
mapFirstRest f g [] = []
mapFirstRest f g (x:xs) = f x : map g xs
```

### capitalize

```
capitalize :: String -> String
capitalize = mapFirstRest toUpper id
```

### maybeCapitalize, titleCase

```
exemptions :: [String]
exemptions = ["the", "and", "or", "is", "of", "on"]
```

```haskell
maybeCapitalize :: String -> String
maybeCapitalize word =
  if word `elem` exemptions then word
  else capitalize word

titleCase :: String -> String
titleCase = unwords . mapFirstRest capitalize maybeCapitalize . words
```

## Tree traversal

```haskell
data Tree a
  = Leaf
  | Branch { value :: a, left, right :: Tree a }
  deriving (Show)

sample1 :: Tree String
sample1 =
  Branch "A"
    (Branch "K"
       (Branch "M"
          Leaf
          (Branch "Q" Leaf Leaf))
       (Branch "P" Leaf Leaf))
    Leaf

sample2 :: Tree String
sample2 =
  Branch "B"
    (Branch "S"
       Leaf
       (Branch "C"
          (Branch "D" Leaf Leaf)
          (Branch "F" Leaf Leaf)))
    (Branch "R" Leaf Leaf)

preorder :: Monoid a => Tree a -> a
preorder Leaf = mempty
preorder (Branch v l r) = v <> preorder l <> preorder r

inorder :: Monoid a => Tree a -> a
inorder Leaf = mempty
inorder (Branch v l r) = inorder l <> v <> inorder r

postorder :: Monoid a => Tree a -> a
postorder Leaf = mempty
postorder (Branch v l r) = postorder l <> postorder r <> v
```

## Stack monoid

```haskell
data BoundedStack a
  = BoundedStack { capacity :: Int, elements :: [a] }
  deriving (Show, Eq)

instance Monoid (BoundedStack a) where
  mempty = new 0
  mappend (BoundedStack c1 e1) (BoundedStack c2 e2) =
    BoundedStack (c1+c2) (e1++e2)

new :: Int -> BoundedStack a
new n = BoundedStack { capacity = n, elements = [] }

push :: a -> BoundedStack a -> Maybe (BoundedStack a)
push elem (BoundedStack cap elems)
  | cap > 0 = Just $ BoundedStack (cap-1) (elem:elems)
  | otherwise = Nothing
```