Assignment 4

due at 23:59 on Wed Oct 4 (80 points)

For this assignment, you will create a set of **total** functions to implement a bounded stack, and then write some additional helper functions for the `Maybe` and `Either` types. Here are some additional considerations:

- You should not be exchanging entire Haskell files with other students. There are too many people submitting exactly the same files, even down to spacing. If you consult with other students, **make sure you type your code separately.** That's the least you can do to at least ensure the code has passed through your brain! Even better would be to discard any notes after consulting with other students, and then reproduce the code on your own. Then you ensure you know how it works.
- Use a comment at the top of the program to include your name, date, and assignment number. Also, any commentary about your work can be helpful in comments: for example, what parts were the trickiest to understand?
- Save all your functions into one file called `a04.hs` and submit it to this dropbox.

Bounded stack

For this section, we're going to produce a **variation** on the bounded stack data type that we studied in class. The main difference is that now we want our functions to be **total** – that is, we can't use `error` when something goes wrong, such as trying to push into a full stack or pop from an empty one.

Instead of `error`, we will alter the types of our functions to use the Haskell `Maybe` type. For example, attempting to pop from an empty stack will produce `Nothing` rather than an exception. And when we successfully pop from a non-empty stack, the result is wrapped in the `Just` constructor, to signify that it worked.

The data definition itself can be the same, although you should derive both `Show` and `Eq` type classes. We need `Eq` in the test code so that we can compare stacks to see if we're getting the desired results.

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

And here are signatures for the six functions you should implement:

```haskell
new :: Int -> BoundedStack a
push :: a -> BoundedStack a -> Maybe (BoundedStack a)
pop :: BoundedStack a -> Maybe (BoundedStack a)
```

```
top :: BoundedStack a -> Maybe a
isFull :: BoundedStack a -> Bool
isEmpty :: BoundedStack a -> Bool
```

Notice that push, pop, and top all return Maybe values now (unlike in class). A stack isFull when it has reached its capacity (and further push operations would produce Nothing). A stack isEmpty when it has no elements (and the pop/top operations would produce Nothing).

Below is a transcript to show how we can test all these functions interactively:

```
ghci> Just s = (push 5 >=> push 6 >=> push 7) (new 4)
ghci> s
BoundedStack {capacity = 1, elements = [7,6,5]}
```

Because push now returns a Maybe, it's not quite as simple to sequence together multiple pushes like we did in class. So there are two concessions here to the Maybe type. First, we bind the variable using `Just s = ...` rather than `s = ....` This is just a form of pattern-matching used in a variable definition. If the right side of the = produces Nothing, it will be a non-exhaustive match failure.

The second concession to Maybe is the >=> operator. When you use >=> to compose functions that produce Maybe values, each step in the sequence will match the Just constructor in order to send the value within it to the next step. If any step produces Nothing, then the whole sequence produces Nothing. Here's an example where that happens, because we apply the sequence to a new stack with insufficient capacity:

```
ghci> (push 5 >=> push 6 >=> push 7) (new 2)
Nothing
```

Pushing the 5 and the 6 succeeds, but when we try to push 7 it fails. We get the same result even if it fails in the middle:

```
ghci> (push 5 >=> push 6 >=> push 7) (new 1)
Nothing
```

Recall that s is still a stack containing [7,6,5]. Here are some further operations on it:

```
ghci> top s
Just 7
ghci> pop s
Just (BoundedStack {capacity = 2, elements = [6,5]})
ghci> Just t = push 8 s
ghci> t
BoundedStack {capacity = 0, elements = [8,7,6,5]}
```

So `t` is full, but `s` is not. Neither one is empty.

```
ghci> isFull s
False
ghci> isFull t
True
ghci> isEmpty s
False
ghci> isEmpty t
False
```

But a brand new stack would be empty, regardless of its capacity.

```
ghci> isEmpty (new 4)
True
```

Here we try to push into a full stack, and pop from an empty stack:

```
ghci> push 9 t
Nothing
ghci> pop (new 4)
Nothing
```

Notice we got `Nothing` instead of exceptions. Here are sequences of `pop` operations using >=>:

```
ghci> (pop >=> pop) t
Just (BoundedStack {capacity = 2, elements = [6,5]})
ghci> (pop >=> pop >=> pop) t
Just (BoundedStack {capacity = 3, elements = [5]})
ghci> (pop >=> pop >=> pop >=> pop) t
Just (BoundedStack {capacity = 4, elements = []})
ghci> (pop >=> pop >=> pop >=> pop >=> top) t
Nothing
ghci> (pop >=> pop >=> pop >=> pop >=> pop) t
Nothing
ghci> top t
Just 8
```

The `Maybe` type

In this section, we'll write a couple helper functions for the `Maybe` type. Recall that `Maybe` is built-in, but equivalent to this:

```
data Maybe a
  = Just a
  | Nothing
```

So the Nothing can be used (like NULL in other languages) to indicate the absence of a value.

mapMaybe

The first one is mapMaybe, which is like map over a list, but the function returns a Maybe. So if the function produces Nothing, we just exclude that element from the list. The signature is:

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
```

To show some examples, let's first define a function that produces a Maybe. This one halves an integer if it's even, but produces Nothing if it's odd:

```
half :: Int -> Maybe Int
half x | even x = Just (x `div` 2)
       | otherwise = Nothing
```

So now the example usage:

```
ghci> half 10
Just 5
ghci> half 11
Nothing
ghci> mapMaybe half [10..15]
[5,6,7]
```

andThen

Sometimes when we have a Maybe value, we want to sequence it with a function that takes the embedded value (if Just) and then also returns Maybe.

For example, push 7 (new 2) produces a Maybe (BoundedStack Int). If we want to apply top to that result, it also produces Maybe Int. So we need to sequence them together in a way that handles the Maybe. (This is related to how the >=> operator works.)

Define a function andThen that can be used for this purpose:

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Here's the example with the BoundedStack

```
ghci> andThen  (push 7 (new 2))  top  -- Both push and top succeed
Just 7
ghci> andThen  (push 7 (new 0))  top  -- push fails, so top isn't executed
Nothing
ghci> andThen  (Just (new 0)) top     -- top fails
Nothing
```

An function like this would often be used "infix" – between its operands. We can surround any function name with the backticks to turn it into an infix operator. We've seen this before with `div` and `mod`. Here are the same examples as the previous transcript, but using infix notation:

```
ghci> push 7 (new 2) `andThen` top
Just 7
ghci> push 7 (new 0) `andThen` top
Nothing
ghci> Just (new 0) `andThen` top
Nothing
```

and here we use it in a chain:

```
ghci> push 7 (new 4) `andThen` push 8 `andThen` push 9 `andThen` pop `andThen` top
Just 8
```

Finally, here are some simpler examples using `half`:

```
ghci> half 20 `andThen` half    -- both succeed
Just 5
ghci> half 10 `andThen` half    -- half 10 succeeds, then half 5 fails
Nothing
ghci> half 5 `andThen` half     -- half 5 fails right away
Nothing
```

The `Either` type

In this last section, we'll define a few simple functions on the `Either` type. Recall that `Either` is built-in to the standard Haskell prelude, but equivalent to this:

```
data Either a b
  = Left a
  | Right b
```

exchange

This simple function should just turn a `Right` value into a `Left`, and vice-versa.

```
exchange :: Either a b -> Either b a
```

```
ghci> exchange (Left 4)
Right 4
ghci> exchange (Right 5)
Left 5
ghci> exchange (Left "Alice")
Right "Alice"
ghci> exchange (Right "Bob")
Left "Bob"
```

mapLeft

Recall that a functor is a type that can hold values of some type, and `fmap` is a generic version of `map` that applies a function to the values *inside* the functor:

```
ghci> :t map                     -- map is specific to lists
map :: (a -> b) -> [a] -> [b]
ghci> :t fmap                    -- fmap works for any functor
fmap :: Functor f => (a -> b) -> f a -> f b
ghci> map (+4) [1..5]
[5,6,7,8,9]
ghci> fmap (+4) [1..5]           -- on lists, fmap same as map
[5,6,7,8,9]
ghci> fmap (+4) (Just 5)         -- Maybe is a functor
Just 9
ghci> fmap (+4) Nothing
Nothing
ghci> fmap (+4) (Right 5)        -- Either is a functor,
Right 9
ghci> fmap (+4) (Left 5)         -- but only on a Right value
Left 5
```

We might like to have something similar to `fmap` but that works on `Left` values instead of `Right` values. Its type would be:

```
mapLeft :: (a -> b) -> Either a c -> Either b c
```

Implement this function. It should behave like so:

```
ghci> mapLeft (+4) (Right 5)
Right 5
ghci> mapLeft (+4) (Left 5)
Left 9
```

coalesce

If both alternatives in the Either type are the same (such as Either Int Int or Either Char Char) then we can eliminate the distinction between left and right. Define this function:

```
coalesce :: Either a a -> a

ghci> coalesce (Left 5)
5
ghci> coalesce (Right 5)
5
ghci> coalesce (Left 'C')
'C'
ghci> coalesce (Right 'D')
'D'
```

Test code

```
import Control.Monad.RWS
import Control.Monad.State
import System.IO
main = do
  flip execStateT (0,0) $ do
    -- Bounded stack
    let s4 = new 4 :: BoundedStack Int
        s0 = new 0 :: BoundedStack Int
    verify "1.01 capacity new" 4 $ capacity s4
    verify "1.02 elements new" [] $ elements s4
    assert "1.03 isEmpty new" $ isEmpty s4
    assert "1.04 isFull new" $ not $ isFull s4
    assert "1.05 isFull new" $ isFull s0
    verify "1.06 capacity push" (Just 3) $ capacity <$> push 6 s4
    verify "1.07 elements push" (Just [9]) $ elements <$> push 9 s4
    verify "1.08 elements push^2" (Just [9,7]) $ elements <$> (push 7 >=> push 9) s4
    verify "1.09 push full" Nothing $ push 9 s0
    verify "1.10 top push" (Just 7) $ (push 7 >=> top) s4
    verify "1.11 top empty" Nothing $ top s0
    verify "1.12 isEmpty pop push" (Just True) $ isEmpty <$> (push 7 >=> pop) s4
    verify "1.13 isEmpty push" (Just False) $ isEmpty <$> push 7 s4
    -- The Maybe type
    let half x | even x = Just (x `div` 2)
               | otherwise = Nothing
        upper x | x `elem` ['a'..'z'] = Just (succ x)
                | otherwise = Nothing
        noChar = Nothing :: Maybe Char
    verify "2.01 mapMaybe []" [] $ mapMaybe half []
    verify "2.02 mapMaybe half" [3..6] $ mapMaybe half [6..12]
    verify "2.03 mapMaybe upper" "fmmppsme" $ mapMaybe upper "Hello World"
    verify "2.04 mapMaybe Nothing" [] $ mapMaybe (const noChar) "goodbye"
    verify "2.05 andThen Nothing" Nothing $ Nothing `andThen` half
    verify "2.06 andThen Just odd" Nothing $ Just 5 `andThen` half
    verify "2.07 andThen Just even" (Just 4) $ Just 8 `andThen` half
    -- The Either type
    let l5 = Left 5 :: Either Int Char
        r5 = Right 5 :: Either Char Int
    verify "3.01 exchange left" r5 $ exchange l5
    verify "3.02 exchange right" l5 $ exchange r5
    verify "3.03 mapLeft left" (Left 10) $ mapLeft (*2) l5
    verify "3.04 mapLeft right" (Right 5) $ mapLeft succ r5
    verify "3.05 coalesce left" 10 $ coalesce (Left 10)
    verify "3.06 coalesce right" 10 $ coalesce (Right 10)
    verify "3.07 mapLeft same as exchange/fmap" (mapLeft (*2) l5) $
      (exchange $ fmap (*2) $ exchange l5)
  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
```