Assignment 4 solutions

Test driver

```
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
            ++ " got " ++ show actual
-- End of test driver
```

Bounded stack

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

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

pop :: BoundedStack a -> Maybe (BoundedStack a)
pop (BoundedStack cap (_:elems)) =
  Just (BoundedStack (cap+1) elems)
pop _ = Nothing
```

```haskell
top :: BoundedStack a -> Maybe a
top (BoundedStack _ (elem:_)) = Just elem
top _ = Nothing

isFull :: BoundedStack a -> Bool
isFull = (<= 0) . capacity

isEmpty :: BoundedStack a -> Bool
isEmpty = null . elements
```

The `Maybe` type

```haskell
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ [] = []
mapMaybe f (x:xs) =
  case f x of
    Nothing -> mapMaybe f xs
    Just y -> y : mapMaybe f xs

andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
andThen Nothing _ = Nothing
andThen (Just a) f = f a
```

The `Either` type

```haskell
exchange :: Either a b -> Either b a
exchange (Left a) = Right a
exchange (Right b) = Left b

mapLeft :: (a -> b) -> Either a c -> Either b c
mapLeft f (Left a) = Left (f a)
mapLeft _ (Right c) = Right c
```

Here's an alternative definition of `mapLeft`, using the functor `fmap` (which for the `Either` type is essentially a `mapRight`):

```haskell
mapLeft2 f = exchange . fmap f . exchange

coalesce :: Either a a -> a
coalesce (Left a) = a
coalesce (Right a) = a
```