You must complete 4 of the 5 problems on this exam for full credit. Each problem is of equal weight. Please leave blank, or draw an X through, or write “Do Not Grade,” on the problem you are eliminating; I will grade the first 4 I get to if I can not figure out your intention—no exceptions! Use pen if possible, and don’t write on the back of the page, use the blank page at the end of the exam (and tell me you have done so). In composing your answers, remember that your goal is to show me you understand the techniques presented in the course; if you can not completely solve the problem, show me as much as you know and I will attempt to give you partial credit.

Problem One (Types)

Give the type for each of the following functions in the space indicated. Use Integer as the type of any numeric expression. Use a, b, c, … for type variables.

(a) \texttt{firstOne :: Maybe [a]}

\texttt{firstOne = Just [\]}

(b) \texttt{f :: (a, b -> c, b) -> (a, c)}

\texttt{f (x, y, z) = (x, (y z))}

(c) \texttt{g :: (Bool,a) -> Either a a}

\texttt{g (True,x) = Left x}
\texttt{g (False,y) = Right y}

(d) \texttt{h :: (Integer -> a) -> a}

\texttt{h = \x -> ((\y z -> y z) x 3)}

(e) \texttt{lastOne :: a -> (a -> Maybe Bool) -> ([a] -> Bool) -> Maybe Bool}

\texttt{lastOne x = \y z \rightarrow let w = z [x]}
\texttt{\rightarrow in if w then y x else Just w}

There was a typo in (e), I left out the in, so I gave everyone credit who answered it.
Problem Two (Haskell Programming)

(a) Write your own version of Haskell's `zipWith` function, which combines the `zip` and `map` functions according to your textbook. **Be sure to provide the function's type declaration.**

Here are some examples of the `zipWith` function in action:

```
Main> zipWith (++) ["a","b","c"]["d","e","f"] => ["ad","be","cf"]
Main> zipWith max [1,2,3,4] [4,3,2,1] => [4,3,3,4]
Main> zipWith (+) [1,2,3][] => []
Main> zipWith min ["az"]["xbc"] => ["ab"]
```

**Solution:**

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = (f x y):(zipWith f xs ys)
```

**Solution using map and zip (just last line):**

```
zipWith f xs ys = map (%(x,y) -> f x y) (zip xs ys)
```

(b) Use `zipWith` to implement the following function, which checks if a list is a palindrome (same forward and backwards). You must use `zipWith` but may otherwise use any other helper functions defined in Haskell.

```
isPalindrome [1,3,2,2] => False
isPalindrome "aba" => True
isPalindrome [5] => True
isPalindrome [] => True
```

```
isPalindrome :: Eq a => [a] -> Bool
isPalindrome xs = and (zipWith (==) xs (reverse xs))
```
Problem Three  (Haskell Programming)

This problem concerns recursive list processing in Haskell. In each case, you must write a function, and also provide the type for your definition.

(a) Write the function `take` which returns the first n elements of a list, according to the examples at the bottom of the page. You may assume n >= 0.

```haskell
take :: Integer -> [a] -> [a]
take 0 xs = xs
take _ [] = []
take n (x:xs) = take (n-1) xs
```

(b) Write a function `rotate` which rotates a list n positions to the left (taking elements from left and inserting on the right), according to the examples at the bottom of the page. You may assume n >= 0.

```haskell
rotate :: Integer -> [a] -> [a]
rotate _ [] = []
rotate 0 xs = xs
rotate n (x:xs) = rotate (n-1) (xs ++ [x])
```

(c) Define `flatten`, which concatenates every member of a list of lists, according to the examples at the bottom of the page. You must define this using `foldr` (for half credit, do it without `foldr`).

```haskell
flatten :: [[a]] -> [a]
flatten xs = foldr (++) [] xs

-- without foldr:
flatten :: [[a]] -> [a]
flatten [] = []
flatten (x:xs) = x++(flatten xs)
```
Problem Four (Type Classes)

In order for type classes to work properly in the Haskell ecosystem, they have to follow certain algebraic properties, which are called “laws” in the Haskell community. One of these laws is the following:

**Symmetry:** \( \forall x, y. \ x == y \leftrightarrow y == x \)

(a) Write a data type for Trees which can hold any kind of data.

```haskell
data Tree a = Null | Node (Tree a) a (Tree a)
```

(b) Write an instance declaration for this data type which makes it an instance of the Eq type class:

```haskell
instance (Eq a) => Eq (Tree a) where
  Null == Null = True
  (Node left x right) == (Node left' x' right') =
    left == left' && x == x' && right == right'
  _ == _ = False
```

(c) Prove that your definition satisfies the Symmetry law.

A completely formal proof would require structural induction on the terms, but I'll give you full credit if you got the basic outline correct.

(Base Case): Null == Null is obviously symmetric.

(Induction Hypothesis): Assume that == is symmetric on the type a and on all proper subterms of these two data expressions:

\( (\text{Node } \text{left } x \text{ right}), \quad (\text{Node } \text{left'} \ x' \text{ right'}) \).

(Induction Step)

\[
\begin{align*}
\text{(Node } \text{left } x \text{ right}) &= (\text{Node } \text{left'} \ x' \text{ right'}) \\
\iff \text{left} &= \text{left'} \land x &= x' \land \text{right} &= \text{right'} \\
&\quad \text{by the definition of } == \text{ on Trees} \\
&\iff \text{left'} &= \text{left} \land x' &= x \land \text{right'} &= \text{right} \\
&\quad \text{by the induction hypothesis.} \\
&\iff (\text{Node } \text{left'} \ x' \text{ right'}) &= (\text{Node } \text{left } x \text{ right})
\end{align*}
\]
Problem Two  (Functors)

Consider the following data type for BB Lists:

\[
\text{Data List } a = \text{Nil} | \text{ Cons } a \ (\text{List } a)
\]

(a) Make your data type for Lists into an instance of Functor.

\[
\text{instance Functor List where}
\begin{align*}
\text{fmap } f \ 	ext{Nil} & = \text{Nil} \\
\text{fmap } f \ (\text{Cons } x \ xs) & = \text{Cons } (f \ x) \ (\text{fmap } xs)
\end{align*}
\]

(b) Use your definition from (b) to write a function which takes a List of Strings, and capitalizes every member, as shown in the example at the bottom of the page. You should use the function \text{toUpper} :: \text{Char} \rightarrow \text{Char} in your code. Don’t worry about empty Strings in the List. (For half credit, you may do it without using your definition from (b).)

\[
\text{capitalizeList } :: \text{List String} \rightarrow \text{List String}
\]

\[
\text{capitalizeList lst } = \text{fmap capitalizeString lst}
\]

where \text{capitalizeString} \ (x:xs) = (\text{toUpper } x):xs