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 \( a, b, c, \ldots \) for type variables.

(a) \( \text{firstOne} :: \text{Maybe } [a] \)
   \[
   \text{firstOne} = \text{Just } []
   \]

(b) \( \text{f} :: (a, b -> c, b) -> (a, c) \)
   \[
   \text{f} (x, y, z) = (x, (y z))
   \]

(c) \( \text{g} :: \text{(Bool,a)} -> \text{Either a a} \)
   \[
   \text{g} (\text{True},x) = \text{Left } x
   \]
   \[
   \text{g} (\text{False},y) = \text{Right } y
   \]

(d) \( \text{h} :: (\text{Integer } -> a) -> a \)
   \[
   \text{h} = \lambda x -> ((\lambda z -> y z) x 3)
   \]

(e) \( \text{lastOne} :: a -> (a -> \text{[Bool]}) -> ([a] -> \text{Bool}) -> \text{[Bool]} \)
   \[
   \text{lastOne } x = \lambda y z -> \text{let w = z } [x]
   \]
   \[
   \text{in } \text{if w then y } x \text{ else } [w]
   \]
Problem Five  (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 (+) [1,2,3,4] [4,3,2,1] => [4,3,3,4]
Main> zipWith (++) "abc" "xyz" => "axzybc"
Main> zipWith min [1,2,3,4] [5,4,3,2] => [1,2,3,2]
Main> zipWith max [1,2,3,4] [5,6,7,8] => [5,6,7,8]
Main> zipWith min [1,2,3,4] [] => []
Main> zipWith max [1,2,3] [] => []

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

For full credit on (b) you must use `zipWith` in an essential way; you may do it without `zipWith` for half credit maximum.
Problem Four  (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 `drop` which deletes the first n elements of a list, according to the examples at the bottom of the page. You may assume n >= 0.

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

(b) Write a function `rotate` which rotates a list n positions to the right (taking elements from right and inserting on the left), 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 xs = rotate (n-1) ((last xs):(init xs))
```

(c) Define `reverse`, which simply reverses the order of the elements in a list, according to the examples at the bottom of the page. You must define this using `foldr` (for half credit, do it without `foldr`).

```haskell
reverse :: [a] -> [a]
rev xs = foldr snoc [] xs
    where snoc x xs = xs ++ [x]
```
Problem Three (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: \( \forall x, y. \ x == y \leftrightarrow y == x \)

In this problem you will create a data type an instance of Eq and prove that it is symmetric:

(a) Write a data declaration 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 type class Eq:

```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:

\( (Node \ left \ x \ right), \ (Node \ left’ \ x’ \ right’) \).

(Induction Step)

\[ (Node \ left \ x \ right) == (Node \ left’ \ x’ \ right’) \]

\[ \leftrightarrow \left\{ \begin{array}{l}
    \text{left} == \text{left’} \land \text{x} == \text{x’} \land \text{right} == \text{right’} \\
    \text{left} == \text{left’} \land \text{x} == \text{x’} \land \text{right} == \text{right’}
\end{array} \right. \]

\[ \leftrightarrow \left\{ \begin{array}{l}
    \text{by the definition of == on Trees} \\
    \text{by the induction hypothesis.}
\end{array} \right. \]

\[ \leftrightarrow (Node \ left’ \ x’ \ right’) == (Node \ left \ x \ right) \]
Problem Two (Functors)

Consider the following data type for BB Lists:

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

(a) Make your data type for Lists into an instance of \text{Functor}.

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

(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).)

\[
\text{capitalizeList :: List String} \rightarrow \text{List String}
\text{capitalizeList lst} = \text{fmap capitalizeString lst}
\text{where capitalizeString} (x:xs) = (\text{toUpper } x):xs
\]

\((\text{Cons “wayne” (Cons “cheng” Nil)})\)
\(\Rightarrow\) \((\text{Cons “Wayne” (Cons “Cheng” Nil)})\)
Overflow and scratch work page