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) firstOne :: [a]

    firstOne = []

(b) f :: (a -> Integer) -> a -> Integer

    f x y = ( (*3) . x) y

(c) g :: (Bool,a) -> Either a a

    g (True,x) = Left x
    g (False,y) = Right y

(d) h :: a -> b -> c -> (b, b)

    h = \x -> \y -> f x y
    where f x = \x y -> (x, x)

(e) lastOne :: a -> (a -> Maybe Bool) -> ([a] -> Bool) -> Maybe Bool

    lastOne x = \y z -> let w = z [x]
                in if w then y x else Just w
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][4,5]
[5,7]
Main> zipWith min ['az']['xbc']
['az']

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 compares each pair of adjacent elements with `<` in a given list as shown in these examples (hint: use `zipWith` and `tail`):

```
comp [1,3,2,2] => [True,False,False]
comp ['a','b'] => [True]
comp [5] => []
comp [] => []
```

For full credit on (b) you must use `zipWith` in an essential way; you may do it without `zipWith` for half credit.

```haskell
--Best Solution:
comp :: Ord a => [a] -> [Bool]
comp xs = zipWith (<) xs (tail xs)

-- Acceptable solution:
comp' [] = []
comp' (x:xs) = (zipWith (<) [x] xs)++(comp' xs)

-- Direct solution without zipWith:
comp'' (x:y:ys) = (x<y):(comp'' (y:ys))
comp'' _ = []
```
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.

\[
\text{take} :: \text{Integer} \rightarrow \text{[a]} \rightarrow \text{[a]}
\]
\[
\text{take 0 xs} = \text{xs}
\]
\[
\text{take} _ \_ \_ \_ \_ = \_ \_ \_ \_ \_
\]
\[
\text{take} n \ (x:xs) = \text{take} \ (n-1) \ xs
\]

(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.

\[
\text{rotate} :: \text{Integer} \rightarrow \text{[a]} \rightarrow \text{[a]}
\]
\[
\text{rotate} _ \_ \_ \_ \_ \_ = \_ \_ \_ \_ \_
\]
\[
\text{rotate} 0 \ \text{xs} = \text{xs}
\]
\[
\text{rotate} n \ (x:xs) = \text{rotate} \ (n-1) \ (xs ++ [x])
\]

(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).

\[
\text{flatten} :: \text{[[a]]} \rightarrow \text{[a]}
\]
\[
\text{flatten} \ \text{xs} = \text{foldr} \ (++) \ [\] \ \text{xs}
\]

\[
\text{-- without foldr:}
\]
\[
\text{flatten} :: \text{[[a]]} \rightarrow \text{[a]}
\]
\[
\text{flatten} \ [\_] = [\_]
\]
\[
\text{flatten} \ (x:xs) = x++(\text{flatten} \ \text{xs})
\]
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: \( \forall x, y. \; x == y \leftrightarrow y == x \)

In this problem you will make the following data type an instance of Eq and prove that it is symmetric:

\[
\text{data MaybePair } a \; b = \text{NoPair | MP } a \; b
\]

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

```haskell
instance (Eq a, Eq b) => Eq (MaybePair a b) where
  NoPair == NoPair = True
  MP x y == MP x' y' = x == y && x' == y'
  _ == _ = False
```

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

There are two cases:

(1) NoPair == NoPair is obviously symmetric

(2) Let \( x, \ x' \) and \( y, \ y' \) be arbitrary expressions of types \( a \) and \( b \), respectively. Then:

\[
(MP \ x \ y) == (MP \ x' \ y')
\]

\( \leftrightarrow \) \( x == x' \land y == y' \) -- by the definition of == on MaybePairs

\( \leftrightarrow \) \( x' == x \land y' == y \) -- by the assumption that types \( a \) and \( b \) satisfy Symmetry

\( \leftrightarrow \) \( (MP \ x' \ y') == (MP \ x \ y) \) -- by the definition of == on MaybePairs
Problem Five (Functors)

(a) Write data type for a binary tree.

\[
data \text{Tree } a = \text{Null} \mid \text{Node (Tree } a\text{)} \ a \ (\text{Tree } a)\]

(b) Make your data type for Trees into an instance of Functor.

\[
\text{instance Functor Tree where}
\]
\[
\text{fmap } f \ \text{Null} = \text{Null}
\]
\[
\text{fmap } f \ (\text{Node } \text{left } x \ \text{right}) = (\text{Node } (\text{fmap } f \ \text{left}) \ (f \ x) \ (\text{fmap } f \ \text{right}))
\]

(c) Use your definition from (b) to write a function which takes a Tree of Strings, and capitalizes every element in the tree, as shown in the example at the bottom of the page. You should use the function \text{toUpper :: Char} \rightarrow \text{Char} in your code. Don’t worry about empty Strings in the Tree.

\[
\text{capitalizeTree :: Tree String} \rightarrow \text{Tree String}
\]
\[
\text{capitalizeTree } t = \text{fmap capitalize } t
\]
\[
\text{where capitalize :: String} \rightarrow \text{String}
\]
\[
\text{capitalize } [] = []
\]
\[
\text{capitalize } (x:xs) = (\text{toUpper } x): xs
\]