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) \text{easy :: Maybe a}

\text{easy = Nothing}

(b) \text{k :: a -> b -> c -> (b,b)}

\text{k = \lambda x -> \lambda y -> f x y}

\text{where f x = \lambda y -> (x, x)}

(c) \text{g :: (a -> Integer) -> a -> Integer}

\text{g x y = ( (*3) . x) y}

(d) \text{f :: (Bool,a) -> Either a a}

\text{f (True,x) = Right x}

\text{f (False,y) = Left y}

(e) \text{weird :: a -> (a -> Maybe Bool) -> ([a] -> Bool) -> Maybe Bool}

\text{weird x = \lambda y z -> let w = z [x]}

\text{in if w then y x else Just w}
Problem Four (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:

```haskell
Main> zipWith (+) [1,2,3] [4,5,6] => [5,7,9]
Main> zipWith min [10,20,30] [2,4,5] => [2,4,5]
Main> zipWith max [10,20,30] [4,5,6] => [10,20,30]
Main> zipWith min ["a","b","c"] ["d","e","f"] => ["ad","be","cf"]
```

**Solution:**

```haskell
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):**

```haskell
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`):

```haskell
comp [1,3,2,2] => [True,False,True]
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 maximum.

--- Best Solution:

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

--- Acceptable solution:

```haskell
comp' [] = []
comp' (x:xs) = (zipWith (<) [x] xs) ++ (comp' xs)
```

--- Direct solution without `zipWith`:

```haskell
comp'' (x:y:ys) = (x<y):(comp'' (y:ys))
comp'' _ = []
```
Problem Five (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 Two (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:

```haskell
data WeirdTriple a b = WT a a b
```

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

```haskell
instance Eq (WeirdTriple a b) where
    WT x y z == WT x' y' z' = x == y && x' == y' && z == z'
    _ == _ = False
```

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

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

\[
(WT \ x \ y \ z) == (WT \ x' \ y' \ z') \leftrightarrow x==x' \&\& y==y' \&\& z==z' \quad \text{-- by the definition of == on WeirdTriples}
\]

\[
x'==x \&\& y==y' \&\& z==z \quad \text{-- by the assumption that types } a \text{ and } b \text{ satisfy Symmetry}
\]

\[
(WT \ x' \ y' \ z') == (WT \ x \ y \ z) \quad \text{-- by the definition of == on WeirdTriples}
\]
**Problem Three  (Functors)**

(a) Write data type for a binary Tree.

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

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

```haskell
instance Functor Tree where
  fmap f Null = Null
  fmap f (Node left x right) = (Node (fmap f left) (f x) (fmap f 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 `toUpper :: Char -> Char` in your code. Don’t worry about empty Strings in the Tree. (For half credit, you may do it without using your definition from (b).)

```haskell
capitalizeTree :: Tree String -> Tree String
capitalizeTree t = fmap capitalize t
  where capitalize :: String -> String
        capitalize [] = []
        capitalize (x:xs) = (toUpper x): xs
```
Overflow and scratch work page