You must complete 3 of the 4 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 3 I get to if I cannot 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 cannot 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) \(\text{firstOne} :: [a]\)
\[
\text{firstOne} = []
\]

(b) \(\text{f} :: (a -> Integer) -> a -> Integer\)
\[
\text{f} \ x \ y = ( (*3) . \ x) \ y
\]

(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) \(\text{h} :: a -> b -> c -> (b, b)\)
\[
\text{h} = \ \lambda x -> \ \lambda y -> f \ x \ y
\]
\[
\quad \text{where} \ f \ x = \ \lambda y -> (x, x)
\]

(e) \(\text{lastOne} :: a -> (a -> \text{Maybe Bool}) -> ([a] -> \text{Bool}) -> \text{Maybe Bool}\)
\[
\text{lastOne} \ x = \ \lambda y \ z -> \text{let} \ w = z \ [x]
\]
\[
\quad \in \text{if} \ w \ \text{then} \ y \ x \ \text{else} \ \text{Just} \ w
\]
Problem Two (Lambda, Let, and Scope)

(a) For the following lambda expression, circle each free variable, and for each bound variable, draw a line from the occurrence of the variable in the body to the corresponding lambda binding.

\[(\lambda x \to ((\lambda x \to (\lambda y \to x y)) \ (\lambda z \to (y z) \ x)))\]

(b) For the following, first show the scope of the bound variables x and y in the expression, and then calculate its value.

```
let y = 2 in let x = 3 * y in x + y
```

```
let y = 2 in let x = 3 * y in x + y
------------------------ scope of the binding for y
----- scope of the binding for x
=> let x = 3 * y in x + y substitution: (y,2)
= let x = 3 * 2 in x + 2 apply substitution
=> x + 2 substitution: (x,(3*2))
= (3*2) + 2 apply substitution
= 8 evaluate
```

(c) Reduce the following expression using one step of Beta Reduction and show the substitution that was used:

\[(\lambda x \to ((\lambda x \to y x) \ x) \ x) \ (\lambda x \to x)\]

Solution:

```
=> (((\lambda x \to x) \ (\lambda x \to y x)) \ (\lambda x \to x)) \ (\lambda x \to x)
subst = ( x, (\lambda x \to x))
```
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. Make your functions polymorphic!

(a) Write a function rotate which rotates a list n positions to the left (taking elements from left and inserting on the right). You may assume n >= 0. Here are examples:

```
--  rotate 1 [1,2,3,4,5,6] =>  [2,3,4,5,6,1]
--  rotate 4 “abc” =>  “bca”
--  rotate 0 [True,False] =>  [True,False]
--  rotate 34 [] =>  []
```

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

(b) Define duplicate, which duplicates each member of a list n times; you may assume n >= 0. Examples

```
--  duplicate 3 ['a','b'] =>  ['a','a','a','b','b','b']
--  duplicate 0 [1,2,3] =>  []
--  duplicate 1 [True,False] =>  [True,False]
```

```
duplicate :: Integer -> [a] -> [a]
duplicate 0 _ = []
duplicate _ [] = []
duplicate n (x:xs) = (replicate n x)++(duplicate n xs)
    where replicate :: Integer -> a -> [a]
          replicate 0 _ = []
          replicate n x = x:(replicate (n-1) x)
```

(rePLICATE is a standard Haskell Prelude function and you could just refer to it if you wish).
Problem Four (Haskell Programming)

This problem concerns binary trees in Haskell. In each case, you must write a function, and also provide the type for your definition. Here is the definition of binary trees you should assume:

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

(a) Write a function `leaves` which returns a list of the leaves of the tree from left to right.

\[
\begin{align*}
\text{leaves :: Tree a -> [a]} \\
\text{leaves Nil} &= [] \\
\text{leaves (Node Nil x Nil)} &= [x] \\
\text{leaves (Node left x right)} &= \text{leaves left} ++ \text{leaves right}
\end{align*}
\]

(b) Define `mirror`, which returns True if the trees are mirror images of each other and False otherwise (examples will be given in class):

\[
\begin{align*}
\text{mirror :: Eq a => Tree a -> Tree a -> Bool} \\
\text{mirror Nil Nil} &= True \\
\text{mirror (Node left x right) (Node left2 x2 right2)} \\
&= (x == x2) \&\& (\text{mirror left right2}) \&\& (\text{mirror right left2}) \\
\text{mirror _ _} &= False
\end{align*}
\]