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 try to avoid writing on the back of the page, but if you do so, please tell me!

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. Don’t forget about class constraints if they are necessary. Use \(a, b, c, \ldots\) for type variables.

(a)

\[
\text{apply} :: (a \to b) \to (c \to a) \to c \to b
\]

\[
\text{apply } f \ g \ x = f \ (g \ x)
\]

(b)

\[
\text{try} :: \text{Num} \ a \Rightarrow a \Rightarrow [a] \Rightarrow [a]
\]

\[
\text{try } x \ xs = x:(x+1):xs
\]

(c)

\[
\text{mystery} :: (a \to a) \to (a \to a \to a) \to a \to a \to a
\]

\[
\text{mystery } f \ h \ x \ y = h \ (f \ (f \ x)) \ (h \ y \ y)
\]
Problem Two (Lambda Calculus) Consider the following lambda expression:

\[
\left( \lambda f \rightarrow \left( \lambda x \rightarrow f (\left( x \ x \right) x) \right) \left( \lambda x \rightarrow f (\left( x \ x \right) x) \right) \right) (z)
\]

(a) Draw a line from each occurrence of a bound variable to the appropriate lambda binding. Circle any free variables and label as “free.” You don’t have to recopy the expression, just draw on the formula as given above.

(b) Give the first 2 reductions in the (infinite) sequence of reductions from this term, starting with the outermost reduction, i.e., the substitution of z for the bound variable f.

\[
\begin{align*}
\lambda f . ( \lambda x . f (x(x)) ) (\lambda x . f (x(x))) & \quad (z) \\
\Rightarrow ( \lambda x . z (x(x)) ) (\lambda x . z (x(x))) & \quad (z \ A \ A) \\
\Rightarrow z ( \lambda x . z (x(x))) (\lambda x . z (x(x))) (\lambda x . z (x(x))) & \quad (z ( A \ A \ A)) \\
& \quad = z ( A \ A \ A) \\
& \quad = z ( A \ A \ A)
\end{align*}
\]

(c) There is clearly a pattern here. What do you predict the term will look like after 100 reductions? Diagram this as best you can, using abbreviations—obviously you can’t draw the whole thing—but convince me that you know what it will look like.
Problem Three  (Haskell Programming)

In this problem you will write a basic dictionary (map) in Haskell. The dictionary will be a list of pairs (key, value), where key and value can be arbitrary types (hence the functions you create below will be polymorphic), and where the list of pairs is sorted in ascending lexicographic order on the keys. For example, you might have the following:

```haskell
example :: [(Int, String)]
example = [(2, "hi"), (5, "there"), (9, "folks")]
```

(A) Write the function `insert`, which takes a key and a value and a dictionary, and returns a new dictionary where the pair (key, value) is now in the dictionary; if the key was already there, replace its value with the new value. Be SURE TO GIVE THE TYPE OF `insert` at the top of your solution. Be sure to maintain the list in ascending order of keys.

```haskell
insert :: Ord a => a -> b -> [(a, b)] -> [(a, b)]
insert k v [] = [(k, v)]
insert k v ((k1, v1):rest)
  | k == k1 = (k, v):rest
  | k > k1  = (k1, v1):(insert k v rest)
  | otherwise = (k, v):(k1, v1):rest
```

(B) Write the function `lookup`, which takes a key and a dictionary, and if the key is in the dictionary, returns `Just value`, and if not, returns `Nothing`. Be SURE TO GIVE THE TYPE OF `lookup` at the top of your solution.

```haskell
lookup :: Eq a => a -> [(a, b)] -> Maybe b
lookup k [] = Nothing
lookup k ((k1, v):rest)
  | k == k1 = Just v
  | otherwise = lookup k rest
```

OR, more efficient, given the dictionary is ordered:

```haskell
lookup :: Ord a => a -> [(a, b)] -> Maybe b
lookup k [] = Nothing
lookup k ((k1, v):rest)
  | k == k1 = Just v
  | k < k1  = Nothing
  | otherwise = lookup k rest
```

(C) Write the function `keys`, which takes a dictionary and returns the list of all the keys in the table. Be SURE TO GIVE THE TYPE OF `keys` at the top of your solution.

```haskell
keys :: [(a, b)] -> [a]
keys [] = []
k:keys rest = k:(keys rest)
```
**Problem Four (Haskell Programming)**

This problem is about Trees as given by the following data declaration:

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

You will write three functions, whose behavior is described in each part and illustrated at the bottom right of this page.

(A) Write a function to insert a new element into the tree. **Allow duplicate elements, so if you find that an element is already there, insert it into the left child.** Do not forget about the type declaration, and use the most appropriate class constraint in your type.

```haskell
insert :: Ord a => a -> Tree a -> Tree a
insert x Null = Node Null x Null
insert x (Node left y right) | x <= y = (Node (insert x left) y right)
                             | otherwise = (Node left y (insert x right))
```

(B) Write a function `inOrder` to turn a tree into a list of elements, by doing an inorder traversal. Don’t forget about the type (is a class constraint necessary?).

```haskell
inOrder :: Tree a -> [a]
inOrder Null = []
inOrder (Node left x right) = (inOrder left) ++ [x] ++ (inOrder right)
```

(C) Write a function count to take an inorder listing and return a list of pairs giving the number of occurrences of a given element in the list. Don’t forget about the type, and use the most appropriate class constraint.

```haskell
count :: Eq a => [a] -> [(a,Integer)]
count xs = count' (map \(x -> (x,1)) xs)
  where count' :: Eq a => [(a,Integer)] -> [(a,Integer)]
        count' [] = []
        count' [(x,n)] = [(x,n)]
        count' [(x,n):xs'] x = x' = count' [(x,n+n'):xs]
                          | otherwise = (x,n):count' ((x',n'):xs)
```

Main> a = foldr insert Null [5,3,7,1,8,4,3,8,1]

Main> inOrder a
[1,1,3,3,4,5,7,8,8]

Main> count (inOrder a)
[(1,2),(3,2),(4,1),(5,1),(7,1),(8,2)]
Problem Five  -- Omitted

(This problem from Fall 2018 is not appropriate for Spring 2019)