1. Function Type Declarations (8 points):
For each of the following function definitions, correctly complete the preceding type declaration. Be sure to include any necessary class constraints.

(A)

```
mystery1 :: (Num a, Eq a) => a -> a -> a -> a
mystery1 x y z | x == y = x + z
    | y == z = y + z
    | otherwise = x + y - z
```

(B)

```
mystery2 :: [a] -> (a -> Bool) -> [a]
mystery2 x p | p (head x) = tail x
    | otherwise = head x : mystery2 (tail x) p
```

(C)

```
mystery3 :: (a -> b -> c) -> [a] -> b -> [c]
mystery3 g [] y = []
mystery3 g (x:xs) y = g x y : mystery3 g xs y
```

(D)

```
mystery4 :: (Ord a) => a -> [a] -> a
mystery4 x l = case l of [] -> x
    (y:ys) -> mystery4 (if x < y then x else y) ys
```
2. Basic Recursion (12 points):

Refer to the following function descriptions and sample call(s)/result(s), and implement them on
the following page using explicit recursion. Unless otherwise indicated you should not use any
built-in functions other than those for basic arithmetic and list construction.

(A) **swapAdj**, which takes a list and returns a list containing the original values, but where suc-
    cessive pairs of values are swapped. If the input list has an odd number of values, the last
    value will remain in its original position.

> swapAdj [1..10]
[2,1,4,3,6,5,8,7,10,9]

> swapAdj "abcdefg"
"badcfeg"

(B) **collatz**, which takes a positive integer \( n \) and returns the Collatz sequence starting at \( n \) as
    a list. The next value in the Collatz sequence for an even number \( n \) is \( n/2 \), while the next
    value for an odd number \( n \) (where \( n > 1 \)) is \( 3 \times n + 1 \); the sequence ends on the number 1.

> collatz 6
[6,3,10,5,16,8,4,2,1]

> collatz 32
[32,16,8,4,2,1]

(C) **chunksOf**, which takes a positive integer \( n \) and a list and returns that list split into chunks
    of size \( n \). If \( n \) does not divide the length of the input list evenly, then the last element of
    the result will be short. You may use the **take** and **drop** functions, whose behavior are also
demonstrated below.

> chunksOf 2 [1..10]
[[[1,2],[3,4],[5,6],[7,8],[9,10]]]

> chunksOf 3 "hello world"
["hel","lo ","wor","ld"]

> take 3 "aloha"
"alo"

> drop 3 "aloha"
"ha"

> take 10 [1..5]
[1,2,3,4,5]

> drop 10 [1..5]
[]
swapAdj :: [a] -> [a]
swapAdj [] = []
swapAdj [x] = [x]
swapAdj (x1:x2:xs) = x2:x1:(swapAdj xs)

collatz :: Integer -> [Integer]
collatz 1 = [1]
collatz n | even n = n : collatz (n `div` 2)
| otherwise = n : collatz (3*n+1)

chunksOf :: Int -> [a] -> [[a]]
chunksOf n [] = []
chunksOf n xs = take n xs : chunksOf n (drop n xs)
3. Evaluating Folds (9 points):
Show the result of evaluating each of the following expressions involving either `foldr` or `foldl`.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) <code>foldl (\r x -&gt; [x] ++ r) [] [1..10]</code></td>
<td><code>[10,9,8,7,6,5,4,3,2,1]</code></td>
</tr>
<tr>
<td>(B) <code>foldr iter ([],0) [1..5]</code> where iter x (y,z) = (x+z:y, x+z)</td>
<td><code>([15,14,12,9,5], 15)</code></td>
</tr>
<tr>
<td>(C) <code>foldr iter ([],1) &quot;CS 495 P P P&quot;</code> where iter ' ' (r,n) = (0:r,n+1) iter _ (r,n) = (n:r,n)</td>
<td><code>([5,5,0,4,4,0,3,0,2,0,1], 5)</code></td>
</tr>
</tbody>
</table>
4. Using Folds (12 points):

Refer to the following function descriptions and sample call(s)/result(s), and implement them on the following page using either foldr or foldl. You should not use explicit recursion, and unless otherwise indicated you should not use any built-in functions other than those for basic arithmetic and list construction.

(A) cluster, which takes a list and returns a list of lists where each sublist contains elements that are equal and were adjacent in the original list.

> cluster "Mississippi"
["M","i","s","i","s","i","p","p","i"]

> cluster "aabbcccc"
["aa", "bbb", "cccc"]

(B) intersperse, which takes an element and a list and “intersperses” that element between the elements of the list.

> intersperse ',' "abcde"
"a,b,c,d,e"

> intersperse 0 [1..5]
[1,0,2,0,3,0,4,0,5]

(C) closest, which takes a number n and a list and returns the number from the list closest in value to n. If there is a tie, the number nearest to the head of the list is returned. You may use the function abs, which returns the absolute value of its argument.

> closest 5 [1.5, 2.9, 4.8, 8.9, 6.2]
4.8

> closest 2 [6, 5, 3, 8, 9, 1, 7]
3
cluster :: (Eq a) => [a] -> [[a]]

cluster xs = foldr it [[]] xs
  where it x ([]:rs) = [x]:rs
        it x (ys@(y:_):rs) | x == y = (x:ys):rs
        | otherwise = [x]:ys:rs

intersperse :: a -> [a] -> [a]
intersperse sep ys = foldr it [] ys
  where it x [] = [x]
        it x rs = x:sep:rs

closest :: (Num a, Ord a) => a -> [a] -> a
closest _ [y] = y
closest v (y:ys) = foldl it y ys
  where it r x | abs(r-v) <= abs(x-v) = r
              | otherwise = x
5. Universal Property of Folds (5 points):

Consider the following recursive function:

\[
\text{foo } f \ [\ ] = 0 \\
\text{foo } f \ (x:xs) = \text{foo } f \ xs + (\text{if } f \ x \ \text{then} \ 1 \ \text{else} \ 0)
\]

Using the universal property of fold, redefine \( \text{foo} \) using \( \text{foldr} \). If your final answer is not correct, showing your derivation may earn partial credit.

\[
\text{foo'} \ f = \text{foldr} \ (\lambda x \ xs \rightarrow xs + (\text{if } f \ x \ \text{then} \ 1 \ \text{else} \ 0)) \ 0
\]