```
--p1
scale_nums :: [Integer] -> Integer -> [Integer]
scale_nums lst factor = map (\x -> x * factor) lst

only_odds :: [[Integer]] -> [[Integer]]
only_odds lli = filter (all (\x -> x `mod` 2 == 1)) lli

largest :: String -> String -> String
largest first second =
  if length first >= length second then first else second

largest_in_list :: [String] -> String
largest_in_list ls = foldl largest [] ls


--p2
count_if :: (a -> Bool) -> [a] -> Int
count_if _ [] = 0
count_if pfunc (x:xs) =
  if (pfunc x)
    then 1 + (count_if pfunc xs)
    else count_if pfunc xs

count_if_with_filter :: (a -> Bool) -> [a] -> Int
count_if_with_filter pfunc la = length (filter pfunc la)

count_if_with_fold :: (a -> Bool) -> [a] -> Int
count_if_with_fold pfunc la = foldl (\acc x -> if pfunc x then acc + 1 else acc) 0 la

--p3
{-
a)In partial application we take a function and bind some of the variables to create a function with fewer arguments.
  When currying however, we transform a function so that it becomes a sequence of functions that each take a single
argument.

b)ii could be equivalent to a->b->c since it would be curried as a->(b->c).  i, however, would take in a function that
accepts types a and b and return an object of type c.

c) (\x -> (\y -> (\z -> (\t -> map t [x,x+z,..y]))))
-}

--p4
{-
a) a
b) b
c) c and d
d) 4 is bound to a and c.  5 is bound to b and d.  6 is bound to e and 7 to f since the lamda function left after evaluatin
g f 4 5 still needs vals for e and f.
-}

--p5
{-
Given that pointers are essentially just the numbers that point to the address of the function in memory, you can still
do arithmetic with the function pointers themselves.  You cannot however add two closures together.
Closures can take in and return various types without altering the closure where as pointers in c must always have th
```

```
e input and output types given upon creation.
-}

--p6
{-
data InstagramUser = Influencer | Normie

lit_collab :: InstagramUser -> InstagramUser -> Bool
lit_collab Influencer Influencer = True
lit_collab _ _ = False

data InstagramUser = Influencer [String] | Normie

is_sponsor :: InstagramUser -> String -> Bool
is_sponsor Normie _ = False
is_sponsor (Influencer sponsors) sponsor= sponsor `elem` sponsors
-}

data InstagramUser = Influencer [String] [InstagramUser]| Normie

count_influencers :: InstagramUser -> Int
count_influencers Normie = 0
count_influencers (Influencer _ followers) =
  length (filter is_influencer followers)
  where
    is_influencer Normie = False
    is_influencer (Influencer _ _) = True

--we can infer that custom value constructors are functions


--p7
data LinkedList = EmptyList | ListNode Integer LinkedList
  deriving Show

ll_contains :: LinkedList -> Integer -> Bool
ll_contains EmptyList _ = False
ll_contains (ListNode x ll) y = if x == y then True else ll_contains ll y

{-
We want to take in the linked list that we want to add the value to
We want to take in the value we would like to add to the linked list
We want to take in the index that we need to visit
And we need to output the New linked list created after we have added the new node to it
-}

ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
ll_insert EmptyList ival x = ListNode ival EmptyList
ll_insert (ListNode curval ll) ival x =
  if x <= 0 then ListNode ival (ListNode curval ll)
            else ListNode curval (ll_insert ll ival (x-1))


--p8
{-
```

```
int LongestRun(vector<boo> vect){
  int longest = 0
  int current = 0
  for(bool i : vect){
    if(i == true){
      current++;
      if(current >= longest)
        longest = current;
    }else{
      current = 0;
    }
  }
  return longest;
}
-}

longest_run :: [Bool] -> Integer
longest_run lb = snd (foldl func (0, 0) lb)
  where
    func (curr, longest) x =
      if x == True
        then if (curr + 1) > longest then (curr + 1, curr + 1) else (curr + 1, longest)
        else (0, longest)

{-
unsigned maxTreeVal(Tree* root){
  if(root == nullptr){
    return 0;
  }

  unsigned max = 0;
  stack<Tree*> stack;
  stack.push(root);
  while(!stack.empty()){
    Tree* currTree = stack.top();
    stack.pop();
    if(currTree->value > max){
      max = currTree->value;
    }
    for(Tree* t : currTree->children){
      if(t != nullptr){
        stack.push(t);
      }
    }
  }

  return max;
}
-}

data Tree = Empty | Node Integer [Tree]
max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node n []) = n
max_tree_value (Node n t) = foldl func n t
```

```
  where
    func accum Empty = accum
    func accum (Node nn tt) = max accum (max_tree_value (Node nn tt))


--p9
fibonacci :: Int -> [Int]
fibonacci n
  | n <= 0 = []
  | otherwise = [fib x | x <- [1..n]]
  where
    fib n
      | n <= 2 = 1
      | otherwise = (fib (n-1)) + (fib (n-2))


--p10
data Event = Travel Integer | Fight Integer | Heal Integer

super_giuseppe :: [Event] -> Integer
super_giuseppe le = altfoldl func 100 le
  where
    altfoldl f accum [] = accum
    altfoldl f accum (x:xs) = if new_accum > 0 then (altfoldl f new_accum xs) else (-1)
      where new_accum = (f accum x)
    func accum (Travel x) = if accum <= 40 then (accum) else (accum + (x `div` 4))
    func accum (Fight x) = if accum <= 40 then (accum - (x `div` 2)) else (accum - x)
    func accum (Heal x) = if accum + x >= 100 then 100 else accum + x
```