Homework 2

Problem 1

```
-- a

scale_nums:: [Integer] -> Integer -> [Integer]
scale_nums nums scaler = map(\num -> num * scaler) nums

-- b
only_odds:: [[Integer]] -> [[Integer]]
only_odds nums = filter (\list -> all odd list) nums

-- c

largest:: String->String->String
largest a b
    | length a >= length b = a
    | otherwise = b

largest_in_list:: [String] -> String
largest_in_list words = foldl largest "" words
```

Problem 2

a)

```
count_if:: (a->Bool) -> [a] -> Integer
count_if _ [] = 0
count_if predfunc (x:xs)
    | predfunc x = 1 + count_if predfunc xs
    | otherwise = count_if predfunc xs
```

c)

```
count_if_with_fold:: (a->bool) -> [a] -> Integer
count_if_with_fold predfunc xs = foldl (\acc x -> if predfunc xs then acc + 1 else acc) 0 xs
```

Problem 3

a)

Currying passes multiple arguments into a a sequence of single-argument functions, whereas partial application fixes multiple arguments into functions that return functions of some lesser number of arguments. They are different as currying uses a chain of functions that only take a single argument, while partial takes more than or equal to 1.

b)

a → b → c is equal to i, but not ii. That is because, a→b→c is a curried function that takes argument a and returns b→c which is the equivalent of i, but not ii because it represents a function of type a→b that returns c.

Problem 4

```
f a b =
  let c = \a -> a   -- (1)
      d = \c -> b   -- (2)
  in \e f -> c d e  -- (3)
```

a) none, only takes argument a and returns itself

b) b is captured, takes argument c and returns b

c) c and d are captured, takes arguments e and f but uses c and d which aren't in the score of the input

d)

```
f 4 5 ->

a = 4
b = 5
c = (\e f -> c d e) 6 = \f -> c d 6 -> c = 6
d = (\f -> c d 6) 7 -> d = 7
```

a = 4, b = 5, e = 6, f = 7

The implementation of only uses the following referenced variables:

- b — used in lambda expression in (2)
- c, d, e — used in lambda expression (3)

a and f aren't used because a and f aren't captured in their respective lines as explained earlier.

Problem 5

Yes, haskell closures are first-class citizens — they can be passed as arguments, used as return values, and assigned to variables as displayed by some of our problems above. The differences compared to pointers in C, are that haskell can use nested functions and all it's variables are immutable, whereas in C nested functions cannot exist (as explained in the problem) and its variables are mutable, making the behavior of function pointers somewhat unpredictable if they reference mutable data.

Problem 7

a)

```
data LinkedList = EmptyList | ListNode Integer LinkedList
    deriving Show

ll_contains:: LinkedList -> Integer -> Bool
ll_contains EmptyList num = False
ll_contains (ListNode val next) num = val == num || ll_contains next num
```

b)

```
ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
```

Given a LinkedList, we want to insert an integer value at some position within the LinkedList. That means the input must include a LinkedList, and we must return a new LinkedList with the new element inserted.

To add the new element, we need an Integer value corresponding to the index at which we want to insert the value, and we must have the value to insert into the list.

Henceforth, we have the inputed LinkedList → Integer → Integer to correspond to the list, value, and position at which we must add the new ListNode, and return a new LinkedList.

c)

```
ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
ll_insert EmptyList _ val = ListNode val EmptyList
ll_insert list ind val
  | ind <= 0 = ListNode val list
ll_insert (ListNode n next) ind value
  | ind == 1 = ListNode n (ListNode val next)
  | otherwise  = ListNode n (ll_insert next (ind - 1) val)
```

Problem 8

b)

```
longest_run :: [Bool] -> Int
longest_run = fst . foldr step (0, 0)
  where
    step b (maxr, curr) =
      if b
        then let new = curr + 1 in (max new maxr, new)
        else (maxr, 0)
```

d)

```
data Tree = Empty | Node Integer [Tree]

max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node val children) = val `max` (foldr (max . max_tree_value) 0 children)
```