Homework 2

Tejas Kamtam

305749402

CS 131 - Fall 2023

Problem 1

Part a

```haskell
-- Problem 1a
scale_nums :: [Int] -> Int -> [Int]
scale_nums l a = map (\x -> x*a)  l
```

Part b

```haskell
-- Problem 1b - using partial application
only_odds :: [[Int]] -> [[Int]]
only_odds = filter (\x -> all (\y -> y `mod` 2 == 1) x)
```

Part c

```haskell
-- Problem 1c - using partial application
largest :: String -> String -> String
largest first second = if length first >= length second then
first else second

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

Problem 2

Part a

```haskell
-- Problem 2a
count_if :: (a → Bool) → [a] → Int
count_if f [] = 0
count_if f (x:xs) = (if f x then 1 else 0) + count_if f xs
```

Part b

```haskell
-- Problem 2b
count_if_with_filter :: (a → Bool) → [a] → Int
count_if_with_filter f l = length (filter f l)
```

Part c

```haskell
-- Problem 2c
count_if_with_fold :: (a → Bool) → [a] → Int
count_if_with_fold f l = foldl (\x y → if f y then x+1 else x)
0 l
```

Problem 3

Part a

Partial applications are a way to generalize functions by omitting parameters and allowing the compiler to figure out that when a user uses the function, they will pass in a parameter that will be appended to the end of the function call. Currying breaks up a complex, multi-parameter function into multiple partial applications of a single parameter to convert the output of a fucntion to output another function that accepts a parameter, effectively producing the same result.

Part b

`a→b→c` is equivalent to ii. `a→(b→c)` via currying because we can have a function with the type defintion of ii. that takes another parameter of type `b` and output `c`. However, it is not equivalent to i. because i. takes a function as input while the original type def only takes `a` and `b`.

Problem 4

Part a

No variables are captured in lambda (1)

Part b

`b` is captured in lambda (2)

Part c

`c` and `d` are captured in lambda (2)

Part d

`f` returns a function of 2 arguments `e` and `f` so that when `f 4 5 6 7` is called, 4 and 5 are bound to `a` and `b` respectively and used in lambda (3). This lambda returns a function `c` of a function `d` on the lambda variable `e`. Because `d` takes an input and outputs the free-variable `b`, it returns 5 regardless of the fact that it takes in `e` (i.e., `a=4`). So 5 is now passed into function `c` which is just the identity function, so it too returns 5 as the final output of `f 4 5 6 7`.SO, the only true free-variables referenced is `b` (although `a` and `b` are both taken as input to lambda (3)).

Problem 5

Closures are first-class citizens, but unlike C function pointers, they cannot execute the code they are caputring. Calling the function pointer with params in C actually executes the code and returns e.g., 8 in the given example. On the other hand, Haskell closures create new functions kind of like function pointers, BUT they can "freeze" the parameters to some values so that all future calls of these new closures will use the captured param values while you can use C function pointers with any params but not make unique "frozen" parametered funcs.

Problem 7

Part a

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

Part b

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

The function should take the linked list, position, and value as input. The position must be a whole number so it must be an Integer (or Int). The type definition of a ListNode specifies the data type must be an Integer. Because all data/variables is immutable and all functions must returnsomething, it makes the most sense to return this new LinkedList with the value inserted

Part c

```
ll_insert :: (Eq LinkedList) => LinkedList -> Integer ->
Integer -> LinkedList
ll_insert (ListNode x xs) pos y
  | pos <= 0 = ListNode y (ListNode x xs)
  | xs == EmptyList = ListNode x (ListNode y EmptyList)
  | otherwise = ListNode x (ll_insert xs (pos-1) y)
```

Added `Eq` type constraint for LinkedList to allow checking for equality.

Problem 8

Part a

```cpp
int longestRun(vector<bool> vec) {
  int max = 0, count = 0;
  for (int i = 0; i < vec.size(); i++) {
    vec[i] ? count++ : count = 0;
    max = max < count ? count : max;
  }
  return max;
}
```

Part b

```haskell
longest_run :: [Bool] -> Int
longest_run l = maximum (scanl1 (\x y -> if y==1 then x+1 else
0) (map (\x -> if x then 1 else 0) l))
```

Part c

```cpp
int maxTreeValue(Tree *tree) {
  if (tree == nullptr) return 0;
  int max = tree->value;
  queue<Tree*> q;
  q.push(tree);
  while (!q.empty()) {
    Tree *current = q.front();
    q.pop();
    max = max > current->value ? max : current->value;
    if (current->children.size() > 0){
      for (int i = 0; i < current->children.size(); i++) {
        q.push(current->children[i]);
      }
    }
  }
  return max;
}
```

Part d

```haskell
max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node x []) = x
max_tree_value (Node x xs) = maximum (x:(map max_tree_value
xs))
```

Problem 9

```haskell
fibonacciN :: Int -> [Int]
fibonacciN (-1) = []
fibonacciN n = take n (1 : 1 : zipWith (+) (fibonacciN n) (tail
(fibonacciN n)))
```