hw

# Homework 2 - CS131

## 1.

### a)

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

-- >>> scale_nums [1, 4, 9, 10] 3
-- [3,12,27,30]
```

### b)

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

-- >>> only_odds [[1, 2, 3], [3, 5], [], [8, 10], [11]]
-- [[3,5],[],[11]]
```

### c)

```haskell
largest :: String -> String -> String
largest a b =
    if length a >= length b then a else b

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

-- >>> largest_in_list ["how", "now", "brown", "cow"]
-- "brown"
-- >>> largest_in_list ["cat", "mat", "bat"]
-- "cat"
-- >>> largest_in_list []
-- ""
```

## 2.

### a)

```haskell
count_if :: (a -> Bool) -> [a] -> Int
count_if f a
    | null a = 0
    | f (head a) = 1 + count_if f (tail a)
    | otherwise = count_if f (tail a)

-- >>> count_if (\x -> mod x 2 == 0) [2, 4, 6, 8, 9]
-- 4
-- >>> count_if (\x -> length x > 2) ["a", "ab", "abc"]
-- 1
```

### b)

```
count_if_with_filter :: (a -> Bool) -> [a] -> Int
count_if_with_filter f a = length (filter f a)

-- >>> count_if_with_filter (\x -> mod x 2 == 0) [2, 4, 6, 8, 9]
-- 4
-- >>> count_if_with_filter (\x -> length x > 2) ["a", "ab", "abc"]
-- 1
```

### c)

```
count_if_with_fold :: (a -> Bool) -> [a] -> Int
count_if_with_fold f a = foldl (\x y -> if f y then x + 1 else x) 0 a

-- >>> count_if_with_fold (\x -> mod x 2 == 0) [2, 4, 6, 8, 9]
-- 4
-- >>> count_if_with_fold (\x -> length x > 2) ["a", "ab", "abc"]
-- 1
```

## 3.

### a)

Currying is when a function, instead of taking multiple parameters, takes one parameter and returns another function that takes one parameter that returns another function… etc. So currying changes function structure from `f(x, y, z)` to `f(x)(y)(z)`. Partial application is the process of calling a curried function with less parameters than it can possibly take so that the return value is itself still a function.

### b)

`a -> b -> c` is equal only to ii. This is because Haskell is right-associative and evaluating first `a -> b` as a function changes the result but evaluating `b -> c` as a function doesn't change the result due to the order of operations Haskell performs.

### c)

```haskell
foo :: Integer -> Integer -> Integer -> (Integer -> b) -> [b]
foo =
    \x ->
        ( \y ->
            ( \z ->
                (\t -> map t [x, x + z .. y])
            )
        )
```

## 4.

### a)

No variables captured

### b)

`b` is captured

### c)

`c` and `d` are captured

### d)

They are bound as follows:

| **4** | **5** | **6** | **7** |
|---|---|---|---|
| a | b | e | f |

Only `b` ( `5` ) really matters in the function implementation: `a` is never referenced at all, `e` is referenced (as a parameter to `d` ) but `d` always returns `b` regardless of its parameter, and `f` is never referenced.

## 5.

Haskell closures are also first-class citizens as they are just treated like any other expression. Closures, unlike function pointers, can capture variables from the scope of the caller whereas function pointers must be given them explicitly as arguments since C cannot have nested functions. C function pointers don't really have any advantages over Haskell closures other than the fact that they can mutate the variables given to them (depending on the function implementation) but this is mostly a programming language difference rather than a specific difference between closures and function pointers.

## 6.

## a)

```
data InstagramUser = Influencer | Normie
```

## b)

```
lit_collab :: InstagramUser -> InstagramUser -> Bool
lit_collab Influencer Influencer = True
lit_collab a b = False
```

## c)

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

## d)

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

## e)

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

### f)

```
count_influencers :: InstagramUser -> Integer
count_influencers Normie = 0
count_influencers (Influencer a b) = sum [1 | _ <- b]
```

### g)

```
-- >>> :t Influencer
-- Influencer :: [String] -> [InstagramUser] -> InstagramUser
```

Here we see that `Influencer` is actually a function that returns an InstagramUser (base class) storing certain data in the form of its parameters.

## 7.

### a)

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

### b)

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

This function takes a `LinkedList` (list to insert node into) an `Integer` (the index) and another `Integer` (the value). The function returns a new `LinkedList` since the original cannot be modified due to everything in Haskell being immutable.

### c)

```haskell
ll_insert EmptyList index value = ListNode value EmptyList
ll_insert (ListNode head next) index value
    | index <= 0 = ListNode value (ListNode head next)
    | otherwise = ListNode head (ll_insert next (index - 1) value)
```

## 8.

---

### a)

```cpp
#include <algorithm>
#include <vector>

int longestRun(std::vector<bool> list) {
    int current = 0;
    int max = 0;

    for (int i = 0, l = list.size(); i < l; i++) {
        if (list[i]) {
            current++;
```

```
                max = std::max(max, current);
            } else {
                current = 0;
            }
        }

        return max;
    }
```

## b)

```
longest_run :: [Bool] -> Int
longest_run list =
    checklist list 0 0
  where
    checklist list current maximum
        | null list = maximum
        | head list = checklist (tail list) (current + 1) (max (current + 1) maximum)
        | otherwise = checklist (tail list) 0 maximum
```

## c)

```
unsigned maxTreeValue(Tree *root) {
    if (!root)
        return 0;

    unsigned maximum = 0;
    queue<Tree *> q;
    q.push(root);
    while (!q.empty()) {
        Tree *current = q.front();
        q.pop();

        if (!current)
            continue;

        maximum = max(maximum, current->value);
        for (int i = 0, l = current->children.size(); i < l; i++) {
            q.push(current->children[i]);
        }
    }

    return maximum;
}
```

## d)

```
max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node val trees) = max val (maxlist [max_tree_value x | x <- trees])
  where
    maxlist list
        | null list = 0
        | length list == 1 = head list
        | otherwise = max (head list) (maxlist (tail list))
```

## 9.

```haskell
fibonacci :: Int -> [Int]
fibonacci n
    | n <= 0 = []
    | n == 1 = [1]
    | n == 2 = [1, 1]
    | otherwise = prev ++ [last prev + (last . init) prev]
  where
    prev = fibonacci (n - 1)
```

## 10.

```haskell
data Event = Travel Integer | Fight Integer | Heal Integer

super_giuseppe :: [Event] -> Integer
super_giuseppe events = propogateEvents events 100
  where
    propogateEvents [] hp
        | hp <= 0 = -1
        | otherwise = hp
    propogateEvents ((Travel n) : xs) hp = propogateEvents xs (travelHandler n hp)
    propogateEvents ((Fight n) : xs) hp = propogateEvents xs (fightHandler n hp)
    propogateEvents ((Heal n) : xs) hp = propogateEvents xs (healHandler n hp)

travelHandler :: Integer -> Integer -> Integer
travelHandler n hp
    | hp <= 0 = -1
    | hp <= 40 = hp
    | otherwise = min 100 (hp + (n `div` 4))

fightHandler :: Integer -> Integer -> Integer
fightHandler n hp
    | hp <= 0 = -1
    | hp <= 40 = hp - (n `div` 2)
    | otherwise = hp - n

healHandler :: Integer -> Integer -> Integer
healHandler n hp
    | hp <= 0 = -1
    | otherwise = min (hp + n) 100
```