Homework 2 - Spring 2023

- Vincent Lin
- [UID REDACTED]
- Section 1D

Question 1

Part (a)

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

Part (b)

```haskell
only_odds :: [[Integer]] -> [[Integer]]
only_odds list =
    filter (all (\num -> num `mod` 2 == 1)) list
```

Part (c)

```haskell
largest_in_list :: [String] -> String
largest_in_list [] = ""
largest_in_list (s:[]) = s
largest_in_list (s:xs) = largest s (largest_in_list xs)
```

Question 2

Part (a)

```haskell
count_if :: (a -> Bool) -> [a] -> Int
count_if predicate [] = 0
count_if predicate (x:xs)
    | predicate x = 1 + count_if predicate xs
    | otherwise = count_if predicate xs
```

Part (b)

```haskell
count_if_with_filter :: (a -> Bool) -> [a] -> Int
count_if_with_filter predicate list =
    length (filter predicate list)
```

Part (c)

```haskell
count_if_with_fold :: (a -> Bool) -> [a] -> Int
count_if_with_fold predicate list =
    foldl counter 0 list
    where
        counter accum item =
            if predicate item then
                1 + accum
            else
                accum
```

This one was a little more challenging. The key point is to notice that the **accumulator** of `foldl` can itself act as a **counter** "variable". Thus, we can simply add 1 to the accumulator every time an `item` meets the `predicate`. `counter` is defined as a private helper function since a one-line lambda would be too cluttered.

Question 3

Part (a)

(DEFINITIONS)

Currying is a technique where a function that takes N arguments is transformed such that its equivalent call is done through a sequence of N functions that take one argument each. For example, if `f` is a function that takes 3 arguments `x`, `y`, `z`, then its curried version `g` is one such that:

```
f(x, y, z) == g(x)(y)(z)
```

Partial function application is a technique where a function is *specialized* such that one or more arguments are fixed, resulting in another function that has fewer arguments. For example, if `f` is a function that takes 3 arguments `x`, `y`, `z`, a partially applied function `g` could be one such that `x` is "always 3" and `y` is "always 5":

```
g(z) == f(3, 5, z)
```

(DIFFERENCES)

In currying, the values of the arguments are not fixed. The caller can still pass in the values they want for each argument, just that it is now done through multiple functions that each take one argument. Partial function application restricts the caller to a version of the function where one or more of the arguments are no longer available to be specified.

Another difference is that currying always results in a function that takes exactly one argument and returns either the final computation or another function with this same constraint. Partial function application always results in another function, one with fewer arguments (not necessarily one) than the function that was partially applied.

Part (b)

`a -> b -> c` is equivalent to (ii), `a -> (b -> c)`.

A quick way to reason about it is to recognize that type definitions are **right-associative** in Haskell:

> The quick and simple rule is right associativity (if you're just adding a parentheses going to the end of the expression it preserves the meaning). - TA Ruining

The deeper answer is that this is a property arising from Haskell automatically **currying** functions behind the scenes. A function `f :: a -> b -> c` is curried to a function that takes one argument of type `a` and return *another* function of type `b -> c`, so the "true" type of `f` is `f :: a -> (b -> c)`, with added parentheses to make it explicit that the entire type `b -> c` describes `f`'s return type.

`a -> b -> c` isn't equivalent to (i) however. (i) describes a function that takes a single argument of type `a -> b` (another function), not `a`.

Part (c)

```
-- Notice that because currying is done under the hood in Haskell, the type
-- definition doesn't need to change when we do it ourselves. In fact, currying
-- gives us insight into the design choice behind writing the type definition as
-- a sequence delimited by ->, with no special treatment of the final return
-- type.
foo :: Integer -> Integer -> Integer -> (Integer -> a) -> [a]
foo x =
    (\y ->
        (\z ->
            (\t ->
                map t [x, x+z..y]
            )
        )
    )
```

Question 4

Part (a)

The lambda labeled `(1)` does not capture any free variables. It does not reference `b` from the enclosing scope, nor does it reference `a` since it redefines it as its own parameter with the same name.

Part (b)

The lambda labeled `(2)` captures the free variable `b`. `b` is in the enclosing scope (that of `f`), and `d` makes a reference to it in its lambda definition.

Part (c)

The lambda labeled `(3)` captures the free variables `c` and `d`, which are technically defined within the scope of the enclosing function `f` via the `let` construct.

We also make the observation that although its definition `c d e` ultimately references `b` within the `d` function, this capture is contained in `d`'s **own closure** and not the lambda labeled `(3)`.

Part (d)

In the function call `f 4 5 6 7`:

- The outermost function `f` takes 2 arguments, so `4` is bound to `a` and `5` is bound to `5`.
- `f` then returns a function which itself takes 2 arguments, so `6` is bound to `e` and 7 is bound to `f`.

Only the value of `b` (`5` in the example) is referenced in the implementation of `f`.

The lambda that is returned by `f` only references `b` - its definition `c d e` is equivalent to `d e` because `c` just returns its argument. Furthermore, `d` always returns the value of `b` regardless of what argument it's given, so `d e` is equivalent to just `b`. In fact, `f` *always* returns `b`, as if it had been defined as:

```
f a b = b
```

Question 5

In the attached reading, functions are first-class citizens if:

1. All items can be the actual parameters of functions
2. All items can be returned as results of functions
3. All items can be the subject of assignment statements
4. All items can be tested for equality.

A closure can be passed as an argument (#1), and this is most obvious with higher order functions like `map`:

```
outer x =
  -- inner is a closure because it captures x
  let inner = (\y -> x + y)
  in map inner [1..x]
```

In the above example, the closure is also assigned to a name called `inner` (#3). We can also return a closure as a result of a function (#2):

```
-- This is also an example of partial function application.
-- The return value of get_func is a closure because it captures x.
get_func x = filter (\item -> item == x)
```

However, you cannot test for equality between two closures. You can't do something like `func1 == func2` with the function objects themselves. This means that while closures exhibit the common traits of first-class functions, **by the definition given**, Haskell closures cannot be considered first-class citizens.

Question 6

Part (a)

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

Part (b)

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

Part (c)

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

Part (d)

```
is_sponsor :: InstagramUser -> String -> Bool
is_sponsor Normie _ = False
is_sponsor (Influencer []) _ = False
is_sponsor (Influencer (x:xs)) sponsor =
    sponsor == x || is_sponsor (Influencer xs) sponsor
```

Part (e)

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

Part (f)

```
count_influencers :: InstagramUser -> Integer
count_influencers Normie = 0
count_influencers (Influencer _ followers) = fromIntegral (length followers)
```

NOTE: I used `fromIntegral` because `length` returns an `Int`, not an `Integer`. Remember that Haskell NEVER converts types implicitly, so returning the return value of `length` by itself would cause a type error.`fromIntegral` allows us to cast the `Int` result to an `Integer` value compatible with `count_influencer`'s type definition.

Part (g)

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

The `[String]` refers to the type of the sponsors list and the `[InstagramUser]` refers to the type of the followers list. Based on the output of `:t`, I can see that custom value constructors are just like any other function. We provide the sponsors and followers as arguments and are returned an instance of the ADT `InstagramUser`.

Question 7

Part (a)

```
ll_contains :: LinkedList -> Integer -> Bool
ll_contains (EmptyList) _ = False
ll_contains (ListNode value list) num =
    num == value || ll_contains list num
```

Part (b)

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

The first parameter is the linked list to insert into. The second parameter is the index position within the list to insert at. The third parameter is the value to insert.

Because all data structures are immutable in functional programming languages such as Haskell, we return a distinct linked list that is the *result* of inserting the value into the provided linked list.

Part (c)

```
-- Base case: an empty list will always become a list with the one given value.
ll_insert (EmptyList) _ num = ListNode num EmptyList
ll_insert (ListNode value tail) index num
    -- Base case: insert at head.
    | index <= 0 = ListNode num (ListNode value tail)
    -- Recursive case: rebuild the head and the result of recursing with
    -- (index-1), effectively "moving down" the list, using the tail as the next
    -- list to consider. Eventually the position we want will appear as the head
```

```
    -- and be caught at the base case.
    | otherwise = ListNode value (ll_insert tail (index-1) num)
```

Question 8

Part (a)

```
#include <vector>
size_t longestRun(std::vector<bool> const &vec)
{
    size_t count = 0;
    size_t max = 0;
    for (size_t i = 0; i < vec.size(); ++i)
    {
        if (vec[i])
        {
            if (++count > max)
                max = count;
        }
        else
            count = 0;
    }
    return max;
}
```

Part (b)

```
longest_run :: [Bool] -> Integer
longest_run list =
    helper list 0 0
    where
        helper :: [Bool] -> Integer -> Integer -> Integer

        -- Base case: there's no more list left, so return the best we have.
        helper [] current_count current_max =
            max current_count current_max

        -- Current TRUE: take max(++count, current_max) and recurse with rest.
        helper (True:xs) current_count current_max =
            helper xs new_count new_max
            where
                new_count = current_count + 1
                new_max = max current_max new_count

        -- Current FALSE: reset current counter to 0 and recurse with rest.
        helper (False:xs) _ current_max =
            helper xs 0 current_max
```

Here we use the technique where we introduce a helper function that will do the recursing instead of the outer function. This gives us a more flexible parameter list, allowing us to pass more information between recursive calls. This way, the Haskell version greatly resembles the algorithm and intuition from the C++ version.

Part (c)

```cpp
#include <queue>

/* Assuming provided code is also defined... */

unsigned maxTreeValue(Tree const *root)
{
    if (root == nullptr)
        return 0;

    unsigned max = 0;
    queue<Tree const *> nodes;
    nodes.push(root);

    while (!nodes.empty())
    {
        Tree const *current = nodes.front();
        nodes.pop();
        if (current->value > max)
            max = current->value;
        for (Tree const *child : current->children)
            nodes.push(child);
    }

    return max;
}
```

Like we learned from CS 32, a recursive algorithm involving trees can likely be rewritten iteratively with the help of a data structure like a stack (DFS) or a queue (BFS). In this case, I use a `std::queue` to BFS through every node in a `Tree`, keeping track of the current maximum at each point.

Part (d)

```haskell
max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node num []) = num
max_tree_value (Node num children) =
    max num children_max
    where
        children_max = maximum [max_tree_value child | child <- children]
```

To recurse on the `Tree`'s children, we can map each child to the result of `max_tree_value` on that child in a simple list comprehension. Then, we take the `maximum` out of all the results and use that as the value `children_max` to compare against the parent's value `num`.

Question 9

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

This implementation kind of cheats by defining a helper function for finding the `x`th Fibonacci number and then using a list comprehension with it to find the first `n` Fibonacci numbers. I assume it isn't the intended solution since it doesn't use the reverse list-building approach described in the hint.

Question 10

I found it hard to implement three separate helper functions as advised by the hint. I found that just using pattern matching on the events argument of my helper function still keeps the code very readable and modular, so I went with this approach instead of defining three separate, specialized functions:

```haskell
handle_events :: [Event] -> Integer -> Bool -> Integer
handle_events [] health _ =
    if health <= 0 then -1 else health

-- Small optimization such that we stop unnecessarily recursing once we reach -1
-- health, set by the fight handler.
handle_events _ (-1) _ = -1

handle_events ((Travel distance):xs) health defensive
    | defensive = handle_events xs health now_defensive
    | otherwise = handle_events xs new_health now_defensive
    where
        new_health = min 100 (health + distance `div` 4)
        now_defensive = new_health <= 40

handle_events ((Fight loss):xs) health defensive =
    if new_health <= 0
        then -1
        else handle_events xs new_health now_defensive
    where
        new_health =
            if defensive
                then health - loss `div` 2
                else health - loss
```

```
        now_defensive = new_health <= 40

handle_events ((Heal gain):xs) health defensive =
    handle_events xs new_health defensive
    where
        new_health = min 100 (health + gain)

super_giuseppe :: [Event] -> Integer
super_giuseppe events =
    handle_events events 100 False
```

The key point to note in this question is that there are two pieces of game "state" we need to keep track of during execution:

- Giuseppe's **hit points**, `health`, which starts at and cannot exceed 100.
- Whether Giuseppe is currently in **defensive mode**, `defensive`, which starts as `False` since 100 > 40.

As seen in previous exercises, whenever we're dealing with "state", we should look to define a helper function(s) to effectively expand the parameters available to our recursive calls to include this state.

I define a helper function `handle_events` which takes in the original `events` list but also the current health `health` and the current defensive state `defensive`. I use pattern matching to determine which kind of event we're currently processing and recurse with an updated health and defensive state as appropriate for the event.