Soyeon Kim, 106292803

Homework 3

CS 131, Fall 2024
Carey Nachenberg

HASK10

Part A

```haskell
scale_nums :: [Integer] -> Integer -> [Integer]
scale_nums ls factor = map (* factor) ls
```

Part B

```haskell
only_odds :: [[Integer]] -> [[Integer]]
only_odds ls = filter (\l -> all (\x -> x `mod` 2 /= 0) l) ls
only_odds ls = filter (\l -> all (odd) l) ls
```

Part C

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

HASK11

Part A

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

Part B

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

Part C

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

HASK12

Part A

None

Part B

b

Part C

c, d

Part D

- **`f a b`**: `f 4 5 6 7` ⟼ `(f 4 5) 6 7` ⟼ a = 4, b = 5
- **`c = \a -> a`**: `a` is not the same as the outer `a`; it just returns the arg.
- **`d = \c -> b`**: no matter what the arg is, it returns the variable `b`, which is 5.
- **`\e f -> c d e`**: e = 6, f = 7

`\e f -> (c d) e` ⟼ `\e f -> d e` ⟼ `\e f -> b` ⟼ `\e f -> 5`
So `b = 5` is actually referenced in the implementation of `f`.

HASK13

Closures in Haskell are first-class citizens because they can be passed as arguments to functions and assigned to variables. Function pointers are mutable; they can point to another function with the same arguments and return type. However, they do not capture variables or states. On the other hand, closures are immutable, capture the variables and states, and allow lazy evaluation.

HASK14

Part A

Partial application is the process of taking a function that requires multiple arguments and fixing one or more of those arguments, returning a new function that takes the remaining arguments. It allows you to call a function with fewer arguments than it originally expects. The resulting function can then be called later with the rest of the arguments.
Currying is the transformation of a function that takes multiple arguments into a sequence of functions, each taking a single argument. In a curried function, each function in the sequence returns another function that takes the next argument, until all arguments are provided and the final result is produced.

Part B

It is equivalent to (ii), but not (i). `a -> b -> c` takes an argument `a` and returns another function `b -> c`. However, `(a -> b) -> c` takes a single function `a -> b` and returns a value `c`.

Part C

```
foo = \x -> \y -> \z -> \t -> map t [x,x+z..y]
```

Part D

```
bar :: (Integer -> a) -> [a]
```

Soyeon Kim, 106292803

HASK15

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_sponser :: InstagramUser -> String -> Bool
is_sponser Normie _ = False
is_sponser (Influencer sponsers) sponsor = elem sponser sponsers
```

Part E

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

Part F

```
count_influencers :: InstagramUser -> Int
count_influencers Normie = 0
count_influencers (Influencer _ followers) = length [f | f@(Influencer _ _) <-
followers]
```

*From ChatGPT: In Haskell, the `@` notation (also called **as-patterns**) allows you to name a pattern while simultaneously breaking it down. It gives you a way to refer to the entire structure you're matching, while also pattern matching on its components.

Part G

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

The custom value constructors are just another function that takes custom types.

HASK16

Part A

```
ll_contains (ListNode val next) target = val == target || ll_contains next target
```

Part B

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

- LinkedList: a linked list that we are trying to insert.

- Int: an index that we want to insert at.
- Integer: a value that we want to insert.
- LinkedList: return a newly inserted linked list to maintain immutability.

Part C

```
ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
ll_insert ll index val = insert_helper ll index val 0
 where
   insert_helper EmptyList index val _ = ListNode val EmptyList
   insert_helper ll@(ListNode v n) index val curr
     | index <= curr = ListNode val ll
     | otherwise = ListNode v (insert_helper n index val (curr + 1))
```

Part D

P nodes must be created and N-P nodes can be reused from the original list.