1. HASK10: Map, Filter, Reduce
    a.

```
scale_nums lst n =
    map (\x -> x * n) lst
```

    b.

```
only_odds :: [[Integer]] -> [[Integer]]
only_odds = filter (all odd)
```

    c.

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

2. HASK11: First-class Functions, Higher-order Functions, Recursion
    a.

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

    b.

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

    c.

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

3. HASK12: Variable Capture
    a. Does not capture any other variable, takes a and returns it
    b. Takes an argument c and ignores it, instead capturing variable b from the outer scope
    c. Captures c and d from let instruction
    d. Variables a and b capture 4 and 5 respectively since they are the first two instructions. Variables e and f capture 6 and 7 respectively since they are passed into the lambda

4. HASK13: Closures
    a. The main difference between Haskell closures and first-class citizens in C is that while closures are also first-class citizens, they encapture and use the variables at

the enclosed scope so that they're always running a function with the same parameters as the given point in time that they were made.

5. HASK14: Currying, Partial Function Application
    a. While both of them can make half-made functions, Currying revolves around breaking down 2+ more parameter functions into functions that take only one parameter while in partial function application you apply subsets of parameters to the current function.
    b. Only ii, since in i the (a -> b) refers to passing one variable in for that parenthesis and for ii the a -> (b -> c) refers to passing in a and then another function that would finish the rest of the currying process.
    c.

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

    d.

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

6. HASK15: Algebraic Data Types
    a.

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

    b.

```
lit_collab :: InstagramUser -> InstagramUser -> Bool
lit_collab user1 user2 = (user1 == Influencer) && (user2 == Influencer)
```

    c.

```
data InstagramUser = Influencer [String] | Normie deriving (Eq)
```

    d.

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

    e.

```
data InstagramUser = Influencer [String] [InstagramUser] | Normie deriving (Eq)
```

    f.

```
count_influencers :: InstagramUser -> Integer
count_influencers (Influencer _ followers) = fromIntegral $ length (filter isInfluencer followers)
    where
        isInfluencer :: InstagramUser -> Bool
        isInfluencer (Influencer _ _) = True
        isInfluencer Normie = False
count_influencers Normie = 0
```

g. Influencer is a type of data structure that can take in multiple values and will return you a type of the data it is a part of

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

7. HASK16: Algebraic Data Types, Immutable Data Structures

a.

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

ll_contains :: LinkedList -> Integer -> Bool
ll_contains EmptyList _ = False
ll_contains (ListNode value rest) target
    | value == target = True
    | otherwise = ll_contains rest target
```

b.

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

c.

```haskell
ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
ll_insert lst value index
    | index <= 0 = ListNode value lst
    | otherwise = insertAt lst value index

insertAt :: LinkedList -> Integer -> Integer -> LinkedList
insertAt EmptyList value _ = ListNode value EmptyList
insertAt (ListNode v rest) value 1 = ListNode value (ListNode v rest)
insertAt (ListNode v rest) value index = ListNode v (insertAt rest value (index - 1))
```

d. All nodes can be reused since it’s a linked list. You just make one new node and then you insert it wherever you need to insert it into the list.