Keeyon Ebrahimi
N14193968
Assignment 2

Each Problem is on it's own page

1. (a)
```
(define (listfromTo a b)
        (cond ((> a b) '())
              (else (cons a (listfromTo (+ a 1) b)))
              ))
```

(b)
```
(define (removeMult a L)
        (cond ((null? L) '())
              (else
                    (if (= (modulo (car L) a) 0)
                        (removeMult a (cdr L))
                        (cons (car L) (removeMult a (cdr L)))
                        ))))
```

(c)
```
(define (sieve n)
        (sieveHelper (listfromTo 2 n)))

(define (sieveHelper L)
        (cond ((null? L) '())
              (else (cons (car L) (sieveHelper (removeMult (car L) (cdr L)))))
              ))
```

2. (a) My solution will require 3 pieces of code to be added into the interpreter.

   (i) Code in my-eval that will recognize calls to **longest**

   (ii) Function **handle-longest** that is called from my eval and takes all the $(exp_i ... result_i)$ pairs in a list

   (iii) **handle-longest-helper** That will find the longest *exp* return list and call handle block on the result.

   (i) This code will be in the $my-eval$ and be used to to recognize **longest** and appropriately call **handle-longest**

```
((eq? (car exp) 'longest)
      (handle-longest (cddr exp) env))
```

   (ii) This **handle-longest** will call the helper method, **handle-longest-helper** that does the bulk of the work. We are giving it an extra empty list. We do the bulk of the work in the helper method because we want an extra parameter that stores the current longest length (*expression...result*), which is what we pass in as an empty list originally. The first condition is the base case checking if we have anymore remaining Expression Result combos to check, and if we do not, we execute whatever is in LongestER. If the remaining Expression Result list still has elements, we check length and recursively call, updating LongestER with the appropriate value.

```
(define (handle-longest ERList env)
      (handle-longest-helper ERList ()' env)
```

   (iii) This **handle-longest-helper** is where the bulk of the work is being done. The first contains the remaining (*Exp...result*) list that need to be evaluate, and the second argument will always contain the current longest (*exp...result*) value.

```
(define (handle-longest-helper RemainingER LongestER env)
      (cond ((null? RemainingER) (my-eval (cadr LongestEr) env))
            (else
                  (cond ((>  (length (my-eval (caar RemainingER) env))
                             (length (my-eval (car LongestER) env)))
                                   (handle-longest-helper (cdr RemainingER)
                                                          (car RemainingER) env))
                        (else (handle-longest-helper (cdr RemainingER) LongestER env)))
                  )))
```

(b) We are able to use let and cond within our handle-let and our handle-cond because our interpreter is actually using the if, cond, let, etc. functionality of the Scheme interpreter below it. It is not using it's own if, cond, etc.

For example, lets say we have Interpreter A that is interpreting itself, which we will call

Interpreter B. When we see a *cond* statement within B, we will call A's $handle - cond$. This $A : Handle - Cond$ will be using the cond of the original scheme interpreter below it, allowing us to use *cond* in $handle - cond$.

I love this question, for when I was implementing the Scheme Interpreter, I was a bit confused about this issue, and it was a nice little light bulb when I learned why this was ok.

3. (a) In ML, the reason all lists have to be of the same type is because ML is a statically typed language. This means that all type checking is done at compile time. This means that it is not possible for there to be a run time type error.

   If a list has different types within it, and we map a function through every element of an array, we can easily hit a run time type error. To eliminate this case, ML does not allow lists of different types.

   (b) $fun\ foo\ (one)\ (two)\ bar\ =$
   $\quad let\ val\ x\ = one\ bar$
   $\quad in\ two\ x$
   $\quad end$

   (c) The type of the following function is

```
'a -> ('b * 'c -> bool) -> 'b * 'c -> int list -> int list
```

   (d) In order to prove this we have to prove the type of each input.

Arg z: The line where we define Bar, we can see that bar will return either z or a. This means that z and a must be the same type. The next line we pass $[1, 2, 3]$, into bar as a. We know that $[1, 2, 3]$ is explicitly type $int\ list$. We are passing in this $int\ list$ into bar, so now we know that a is an $int\ list$. Because a's type and z's type is the same, and a's type is $int\ list$, we can now deduce that z's type is $int\ list$

Arg f: We never use argument f, we f can store anything, meaning that it is polymorphic. When organizing arguments left to right, this is the first polymorphic item we see, it is labeled as $'a$

Arg (x, y): All we do with these two variables is we pass them into another function that is another of foo's arguments. Because passing them into another function gives us no more information on the type that x and y are, we know that they are polymorphic. They also have nothing to do with argument f at all, so they do not get the same lettering as f. a' is already taken so (x, y) get $'b{*}'c$

Arg $(op >)$: In the definition of bar, we see that whatever is returned from the $(op >)$ is used in an if statement. This means that it's return type must be a boolean. The $(op >$ is also being used between argument x and y, which we previously deduced were type $'b$ and $'c$ respectively. This means that $(op >)$ will be of type $('b * 'c\text{-}> bool)$

4. (a) This would not be an **ADT** for a stack because with this implementation, you cannot declare an object of queue. This is because currently queue is in a package.

   (b)
```
package queuePackage is
    package queue is private:
        function extract return integer;
        function insert(x: integer);
    private:
        ...
        ...
        ...
    end queue;
end queuePackage;
```

   It is now possible to to declare a type queue.

5. (a) In order to be an Object Oriented Language the three elements you need are Encapsulation, Inheritance, Subtyping with dynamic dispatch

   (i) **Encapsulation:**
   We have to be able to encapsulate data and code into **objects** In java we have Methods and instance variables. In C++ we have Data members and member functions. Abstract Data types are also a good indicator of encapsulation. Abstract Data types have visible parts and non visible parts. Abstract Data types also have procedures that you can call.

   (ii) **Inheritance:**
   This means that we do not have to re-write code. Code for one type can be re-used with another type

   (iii) **Subtyping with dynamic dispatch**
   This refers to the ability to treat one class as if it were another type. Type B is a subtype of A if all the values in A are also in B.

(b) (i) Subset interpretation of subtyping is saying that if $B <: A$, then every value of type B is also of type A. More specifically, if $B <: A$, then we can treat type B as if it was a type A.

   (ii) Class derivation in Java satisfies the subset interpretation of subtyping because if we have a Class A and Class B, and $B <: A$, then we can use class B whenever we are expecting a Class A. This shows that subtying of classes is Covariant. Here is an example.

   Lets say we can an *Animal* class, and we have a *Dog* class. As we know, a dog is an animal, and should have all the same functionality of all animals. We can then make the Dog class Extend the Animal class, so now $Dog <: Animal$. This means that if we have a function that takes in an object of the class *Animal*, we can now give that function an object of the class *Dog* instead.

   (iii) Functions in Scala satisfy the subset interpretation of subtyping this way.
   If $B <: A$ and $D <: C$, then $A \to D <: B \to C$. This means that We are Contravarient on the Input Types and Covarient on the output types.

   If we have Classes, $Bat <: Animal$ and $Dodge <: Car$, then we have
   $Animal \to Dodge <: Bat \to Car$

   As for the Contravarient input type, here is an explanation. Lets say we have a function foo that takes in our $Bat \to Car$ function, creates a new Bat and runs our Bat through the $Bat \to Car$ function. Now we get the output Car easy peasy.
   Now lets imaging passing in the $Animal \to Dodge$ function into foo instead. If we

do that, now we pass the created *Bat* into the $Animal \rightarrow Dodge$ function, which we know is ok. When it comes to inputs of functions being contravarient, it helps to think of it this way. If we have $Dog <: Animal$, we know that *Dog* contains all the information of *Animal* and anything that is a supertype of *Animal*. That means that if a function can safely operate on all values within a dog, we know that this function can easily operate on *Animal* and other *supertypes* because dog already contains that information.

As for the return type of Covariance, this is the simple example. We have already established that in part (*ii*). We know that the return value from the function will just be a single value, and single variables act the same as Java classes. Subtypes of a single value can be used in place of a supertype, meaning return values act with Covariance.