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**

   $((eq?\ (car\ exp)\ 'longest)$
   $\quad (handle-longest\ (cddr\ exp)\ env))$

   (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.

   $(define\ (handle-longest\ ERList\ env)$
   $\quad (handle-longest-helper\ ERList\ ()'\ env)$

   (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.

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

(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) $fun\ foo\ (one)\ (two)\ bar\ =$
   $let\ val\ x\ = one\ bar$
   $in\ two\ x$
   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.

(c)
```
def maxTree[T <: Ordered](t: Tree[T <: Ordered[T]]) {
    t match {
      case Node(v,l,r) =>
          val maxL = maxTree(l);
          val maxR = maxTree(r);
          if (maxL < maxR)
          {
              max = maxR;
          }
          else
          {
              max = maxL;
          }
          if (max < v)
          {
              max = v;
          }
          max
      case Leaf(v) => v;
      case Empty() => ();
    }
```

(d) Solution on next page in order to not break up code

(d) (i)
```
Class A {...}
Class B extends A { ...}
void foo (ArrayList<A> a)
{
    A c = new A();
    a.add (c);
}

ArrayList<A> x = new ArrayList<A>();
ArrayList<B> y = new ArrayList<B>();
foo(y);  // Now ArrayList<A> an element of type B, which is not appropriate
```

(ii)
```
void foo (ArrayList<? super B> b)
{
    B c = new B();
    b.add (c);
}

ArrayList<A> x = new ArrayList<A>();
ArrayList<B> y = new ArrayList<B>();
foo(x);
```

(e) (i)
```
class E[+T] (val bar: T){
    val baz = bar;
    def get = baz;
}
```

(ii) Code on new page to avoid a break in code

(ii)
```
class A {
    val x = 12;
}

class B extends A {
    val y = 21;
}

class E[+T] (val bar: T){
    val baz = bar;
    def get = baz;
}

def main(args: Array[String]) {
    val b: B = new B();
    val bb: Foo = new Foo(b);
    val a: A = new A();
    val aa: Foo = new Foo(a);
    aa = bb;
    // With generics we are now using
    // type b when an A is expected
    //  This is covariance at work
```

(f) (i)
```
class E[-T] (val bar: List[T]){
    val foo: T = new T();
    val listT = bar + foo;
    bar = bar + foo;
    def get listT;
}
```

(ii) On new page to avoid code break

(ii)
```
class A {
    val x = 12;
}

class B extends A {
    val y = 21;
}

class E[-T] (val bar: List[T]){
    val foo: T = new T();
    val listT = bar + foo;
    bar = bar + foo;
    def get listT;
}

def main(args: Array[String]) {
    val quz: E[B] = new E( List(new A()); // Expecting an E[B], and we give E[A]
                                         // Give supertype when expected subtype
                                        // Contravariance
```

6. (a) The benefit of a mark-and-sweep garbage collector over a reference counting collector is that it is much simpler to implement. Also, a difference between the two is that the mark and sweep only does work when the heap is almost full, whereas the reference counting method is constantly doing work. If you have a program that will never get close to filling up the heap, mark-and-sweep garbage collector will never have to do work during execution, which can save time on smaller applications.

   (b) The advantage of the copying garbage collector over a mark and sweep garbage collector is that a copying garbage collector lines up all of the data when we stick all the marked items into the new heap. Now we have live data, and a heap pointer, and a bunch of free memory all in a line. Because everything is lined up, the total amount of space we take will be less and it will be easier to fit all your data on memory avoiding the need to store to disk.

   (c) The idea behind the Generational Copying Garbage Collection is that the older an object gets, the longer it is expected to live. What we do is we create multiple heaps, usually around 4, and we initially start storing information at heap 0. Once this heap fills up, we start a copying garbage collector. As we mark items that have survived, we also increment a counter on the item to indicate how many garbage collections it has survived. Once an item has survived a certain amount of garbage collections, we move it to the higher lever. We then keep operating on the lower heap, and seem to only delete things that are freshly used and expected to live the shortest amount of time. We repeat the process and keep operating on the lower level, deleting some items and adding some back to the higher level heap. If the higher level heap, heap level 1 ever fills up, we will do the same thing that we did when the lower level heap filled up, except we will do it with the higher level heap and a heap that is even higher than the higher level heap. So when heap 1 fills up, we will start a mark and sweep on that, and if some elements have survived a certain amount of garbage collections in heap level 1, we will then move it to heap level 2.

   (d) Solution on next page to avoid code break

(d) We need to first check if the reference count is equal to 0. If it is equal to 0, we need to call delete on all the children and we then we need to add the pointer to the free list.

```
#include <iostream>
#include <vector>

Class Foo ()
{
    int referenceCount;
    std::vector <Foo> children;
    std::vector <Foo *> childrenPointer;

    // For delete, we are passing x
    // x contains the memory location of itself
    // This means that in order to delete children
    // We must call delete on child and pass in
    // The child's memory location
    private void delete(x)
    {
        referenceCount = referenceCount - 1;
        // If this goes to 0, we know we need
        // 1.  Delete item by adding to freeList
        // 2.  Call Delete for all Foo's children

        if (referenceCount == 0)
        {
            for (int i = 0; i < children.size(); i++)
            {
                children[i].delete(childrenPointer[i]);
            }

            addToFreeList(x);
        }
    }
}
```