

Keeyon Ebrahimi
N14193968
Assignment 2

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 $(exp_i \dots result_i)$ 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 =
  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 its 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 -> bool)*