Goals of this lab:

- Be able to better use higher-order functions in Scheme; Learn what is currying, and be able to use currying in Scheme

1. High Order Functions. From the last lab, we have the generic version of insertion sort function that calls generic insert subroutine. These functions are:

```scheme
(define insert (lambda (func x myList)
    (cond
      ((null? myList) (cons x '()))
      (else
       (if (func x (car myList))
           (cons x myList)
           (cons (car myList) (insert func x (cdr myList))))))
)

and,

(define sort (lambda (func myList)
    (cond
      ((null? myList) '())
      (else
       (let* ((sortedCdr (sort func (cdr myList))))
           (insert func (car myList) sortedCdr))))))
```

The usage of sort is the following if we have the defined functions lte, and gte:

```scheme
(define lte (lambda (x y) (<= x y)))
(define gte (lambda (x y) (>= x y)))

>(sort lte '((3 0) (5 8) (4 9) (1 3)))
(list 3 0 4 9 1 5 8)

(a) You have a list of pairs consisting of numbers such as

'((3 0) (5 8) (4 9) (1 3))

You will sort the pairs on first numbers. First, write your function, and then plug it into the sort function above.

**Solution:**

```scheme
(define sortOnFirst (lambda (x y) (<= (car x) (car y))))

(sort sortOnFirst '((3 0) (5 8) (4 9) (1 3))) ;plug it into the sort function
>;(list (list 1 3) (list 0 3) (list 3 4) (list 5 8))
```

You have a list of pairs consisting of first name and last name such as
'(('Brad" "Johnson") ('Eli" "Manning") ('Mark" "Brunell") ('Tom" "Brady'))
You will sort the pairs on last name. First, write your function, and then plug it into
the generic sort function. (String is a data type in scheme. Hint: While comparing
strings, use string <?, string >?, string <=?, string >=?, string =? operators.)
Solution:

(define sortOnLast (lambda (x y) (string<=? (car (cdr x)) (car (cdr y)))))

(sort sortOnLast ;plug it into the sort function
'(('Brad" "Johnson") ('Eli" "Manning") ('Mark" "Brunell") ('Tom" "Brady')))
>((("Tom" "Brady") ('Mark" "Brunell") ('Brad" "Johnson") ('Eli" "Manning"))
(c) You have a list of triples consisting of numbers such as
'((3 0 7) (5 8 3) (4 9 9) (1 3 1))
You will sort the pairs on the last numbers. First, write your function, and then
plug it into the sort function above.
Solution:

(define sortOnLastN (lambda (x y) (<= (car (cdr (cdr x))) (car (cdr (cdr y)))))

(sort sortOnLastN '((3 0 7) (5 8 3) (4 9 9) (1 3 1)))
>((1 3 1) (5 8 3) (3 0 7) (4 9 9))

2. Currying in Scheme. Currying is the idea of interpreting an arbitrary function to be of
one parameter, which returns a possibly intermediate function, which can be used further
on in a calculation. Currying can be seen as a way of generating intermediate functions
which accept additional parameters to complete a calculation. An Example:

(define make-adder (lambda (x)
  (lambda (y) (+ x y)))
(make-adder 1))

make-adder is our currying function. It generates an intermediate function which is used
in the function, inc, later.

(inc 3) => 4

By using the same make-adder function, we can create another increment function. This
time, incr-by-two.

(define inc-by-two (make-adder 2))

(inc-by-two 5) => 7

(a) Now, create a curried version of the sort function that takes a comparison operator,
and returns a sort function. Then, Use this currying function to define the functions
sortAscending, and sortDescending. Usage of sortAscending and sortDescending will
be the following:

(sortAscending '((4 7 2 5)) => (list 2 4 5 7)
(sortDescending '((4 7 2 5)) => (list 7 5 4 2)
Solution:

For the curried version of the sort function:

```scheme
(define sortCurry (lambda (func)
    (lambda (myList)
        (cond
            ((null? myList) '())
            (else
                (let* ((sortedCdr (sort func (cdr myList))))
                    (insert func (car myList) sortedCdr))))))
```

or, simply:

```scheme
(define sortCurry (lambda (func)(lambda (myList) (sort func myList))))
```

and, the function definitions:

```scheme
(define sortAscending (sortCurry <=))
(define sortDescending (sortCurry >=))
```

(b) Using your curried sort function, define specialized-sort functions to solve the problems in question 1.

Solution:

For the question 1, part a:

```scheme
(define sortOnFirstCurry (sortCurry sortOnFirst))
(sortOnFirstCurry '((3 0) (5 8) (4 9) (1 3)))
>((1 3) (3 0) (4 9) (5 8))
```

For the question 1, part b:

```scheme
(define sortOnLastCurry (sortCurry sortOnLast))
(sortOnLastCurry '(("Brad" "Johnson") ("Eli" "Manning"
"Mark" "Brunell") ("Tom" "Brady"))
>((("Tom" "Brady") ("Mark" "Brunell") ("Brad" "Johnson") ("Eli" "Manning"))
```

For the question 1, part c:

```scheme
(define sortOnLastNCurry (sortCurry sortOnLastN))
(sortOnLastNCurry '((3 0 7) (5 8 3) (4 9 9) (1 3 1)))
>(((1 3 1) (5 8 3) (3 0 7) (4 9 9))
```