

CS 3721: Programming Languages Lab

Lab #04: Currying in Scheme

Due date: February 19, 3:30pm. At the beginning of the next recitation.

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:

```
(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:

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

```
>(sort lte '(2 9 6 7))
(list 2 6 7 9)
>(sort gte '(2 9 6 7))
(list 9 7 6 2)
```

- (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.

- (b) 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.)

- (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.

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))))
(define inc (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)
```

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