

CS 3721: Programming Languages Lab

Lab #03 Solutions: Higher order functions in Scheme

1. The operator, *let**. This operator is used for variable bindings, and has the following syntax:

```
(let* <bindings> <body>)
```

<bindings> should have the form

```
((<variable1> <init1>) ...),
```

where each <init> is an expression, and <body> should be a sequence of one or more expressions. The bindings are performed sequentially from left to right. Therefore, the second binding is done in an environment in which the first binding is visible, and so on. An example:

```
(let* ((x 7)           ; first binding
      (y 3)           ; second binding
      (z (+ x y)))   ; third binding: first and second bindings are visible
      (* z x y))     ; <body>: you can use your <bindings> in the <body>
```

=> 210

Given the lengths of the legs of several right triangles in a list, define a recursive function *calcHypotenuse* which calculates the hypotenuses of the triangles using the *let** operator. *Hint:* You may want to embed the following template into the else part of the check for whether the input list is null.

```
(let*
  ((triangle <expression for evaluating triangle, you may use your input list>)
   (leg1 <expression for evaluating leg1, you may use triangle>) ; second binding
   (leg2 <expression for evaluating leg2, you may use triangle>) ; third binding
   (hypotenuse <expression for evaluating hypotenuse> )) ; fourth binding
  (... hypotenuse ...) ; <body>
)
```

Test your function with the expression below.

```
> (calcHypotenuse '((5 12) (3 4) (7 24) (8 15)))
(list 13 5 25 17)
```

Solution:

```
(define calcHypotenuse (lambda (lst)
  (cond
    ((null? lst) '())
    (else
     (let*
      ((triangle (car lst))
       (leg1 (car triangle))
       (leg2 (car (cdr triangle)))
       (hypotenuse (sqrt (+ (* leg1 leg1) (* leg2 leg2)))))) ; <binding> ends here
      (cons hypotenuse (calcHypotenuse (cdr lst))) ; let body
      )))))
```

2. The function *insert* takes a number x and a sorted list of numbers in ascending order as arguments. It returns a sorted list in ascending order including the number x . Following is the scheme code for this function.

```
(define insert (lambda (x myList)
  (cond
    ((null? myList) (cons x '())) ;base case
    (else
     (if (<= x (car myList))
         (cons x myList)
         (cons (car myList) (insert x (cdr myList))))))))
```

You can call it by the following:

```
> (insert 6 '(1 5 7))
(list 1 5 6 7)
```

Now, using *insert*, write a function *sort* that takes a list of numbers and evaluates to a list containing those same integers sorted into ascending order (*Hint*: Use *let* to give a name to the result of recursively sorting the rest of the list (*cdr field* of the list)). Test your function with the expression below.

```
> (sort '(3 9 6 1))
(list 1 3 6 9)
```

Solution:

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

3. Higher-order functions. *Higher-order function* is a function that either takes a function as an argument or returns a function as a result (or both). The following *maplist* function is a higher-order function that takes a function and list and applies the function to every element in the list. The result is a list that contains all the results of function application.

```
(define maplist (lambda (f myList)
  (cond
    ((null? myList) '())
    (else
     (cons (f (car myList)) (maplist f (cdr myList)))))))
```

You can use this function by:

```
> (maplist (lambda (x) (* x x)) '(1 2 3 4 5))
(list 1 4 9 16 25)
```

Now, using *maplist*, write a function *seconds* that takes a list containing lists having more than one elements and returns the second elements of the lists in a flat list. Test your function with the expression below.

```
>(seconds '((e l i) (m a n) (n i n g)))
(list 'l 'a 'i)
```

Solution:

```
(define seconds (lambda (myList)
  (maplist car (maplist cdr myList))))
```

or

```
(define seconds (lambda (myList)
  (maplist (lambda (x) (car (cdr x))) myList)))
```

4. Modify the first-order functions *insert* and *sort* in question 2 into higher-order functions. Now, *insert** and *sort** take a comparison function as an argument in addition to their original arguments. In this way, we can sort a list in an order (ascending or descending) according to the comparison function. Test your functions with the following expressions.

```
(define lte (lambda (x y) (<= x y))) ; comparison function - ascending order
(define gte (lambda (x y) (>= x y))) ; comparison function - descending order
```

```
>(sort* lte '(3 9 6 7))
(list 3 6 7 9)
>(sort* gte '(3 9 6 7))
(list 9 7 6 3)
```

Solution:

The original insert function:

```
(define insert (lambda (x myList)
  (cond
    ((null? myList) (cons x '())) ;base case
    (else
     (if (<= x (car myList))
         (cons x myList)
         (cons (car myList) (insert x (cdr myList))))))))
```

The modified insert function:

```
(define insert* (lambda (f x myList)
  (cond
    ((null? myList) (cons x '())) ;base case
    (else
     (if (f x (car myList))
         (cons x myList)
         (cons (car myList) (insert* f x (cdr myList))))))))
```

The original sort function:

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

And, the modified sort function:

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

Extra Credit:

5. More Recursion. Write a scheme code to define a function *multirember* that takes an atom *a* and a list of atoms *lat* and returns a new list in which all occurrences of the member *a* have been removed. Test your function with the following expression.

```
> (multirember 'green '(orange yellow green red blue green))
(list 'orange 'yellow 'red 'blue)
```

Solution:

```
(define multirember (lambda (a lat)
  (cond
    ((null? lat) '())
    ((eqv? (car lat) a) (multirember a (cdr lat)))
    (else (cons (car lat) (multirember a (cdr lat)))))))
```

6. Write a scheme code to define a function *multirember** modifying your *multirember* function in question 5. This time, the list is composed of atoms and lists. It returns a new list in which all occurrences of the member *a* have been removed from the main list and any embedded lists. Test your function with the following expression.

```
> (multirember* 'green '((yellow) red (green bay) green))
(list (list 'yellow) 'red (list 'bay))
```

Solution:

```
(define multirember* (lambda (a lst)
  (cond
    ((null? lst) '())
    ((pair? (car lst))
     (cons
      (multirember* a (car lst))
      (multirember* a (cdr lst))))
    ((eqv? (car lst) a) (multirember* a (cdr lst)))
    (else (cons (car lst) (multirember* a (cdr lst)))))))
```