FUNCTIONAL PROGRAMMING – ASSIGNMENT 1

Hamza Bhatti (21223241)

Contents

Introduction

For this assignment, the functional programming language used was Scheme (based on Lisp). A variety of user defined functions from the book “The Little Schemer” will be identified here.

The purpose of this assignment, was to go through the user defined functions, describe the functions and how they work. Tracing executions will be provided to give a demonstration of what occurs in each function to give a result.

Primitive Procedure

Along with the user defined functions that will be identified later, there are a variety of primitive procedures that are set within Scheme. These are described below.

And

The ‘and’ procedure works just like the AND gate. Two statements will have to be true for the outcome to be true.

Not

The ‘not’ procedure will invert a true or false value.

Or

For this procedure, at least one of statement can be true for the outcome to be true.

Eq?

The ‘eq?’ procedure will check to see if two elements or statements are equal. If this is so, true will be returned.

Null?

With this procedure, it will check to see if an element is null. This can be useful when checking to see if a list empty.

Car

The ‘car’ procedure will return the first element of a list. The first element can also an atom or a nested list.

Cdr

The primitive procedure ‘cdr’ works by returning every element within a list, except for the first element.

Cons

The ‘cons’ primitive is used to append an element to the front of a given list.

Recursion

Recursion is a big part of functional programming. This concept is like a function call, but we ourselves call the function. This is very useful when we want to traverse a list.

One case that is important, is the Recursive case. This is when we call our own function by identifying its name. The purpose of this is so that the function call can help us reach a goal. For example, to check if all elements in a list are atoms.

Chapter 1

Atom?

The ‘atom?’ function is used to check if a given element is an atom. An atom can be defined as a string of characters. An atom can be identified when there is an ‘ at the front of the element.

The function can be seen below, with some comments to gain an idea of what each line does.

```
(define atom?
  ; x is element
  (lambda (x)
    ; To be true, must not be pair
    (and (not (pair? x))
         ; Must not be null
         (not (null? x)))))
```

How It Works

An element is provided as an argument to the function. Checks will take place to see if the argument is a ‘pair?’ or ‘null?’. These are then inverted with the ‘not’ primitive. The inverted values will then be evaluated with ‘and’. If both values are true, then the argument was an atom.

Tracing Execution

Case1

```
; (atom? 'Adam)

;1 (and (not (pair? 'Adam))(not (null? 'Adam)))
;2 (and (not #f)(not #f))
;3 (and #t #t))
;4 #t
```

1 The argument is checked to see if it is a pair and null.
2 ‘Adam is not null or a pair. Next values will be inverted with not.
3 Now the values are both true and evaluated with and.
4 As both values are true, the result is true. ‘Adam is an atom.

Case2

```
; (atom? '())

;1 (and (not (pair? '()))(not (null? '())))
;2 (and (not #f)(not #t))
;3 (and #t #f)
;4 #f
```

1 The argument is an empty list and checked to see if it is a pair or null.
2 The empty list is not a pair but is null.
3 The values are inverted to give true and false respectively. They are then evaluated with and.
4 The result is false as both values where not true. An empty list is not an atom.

Chapter 2

Lat?

The 'lat?' functions is a way of checking if each element in a list is an atom. If a nested list is found, it should give a result of false.

The function is provided below with comments.

```
(define lat?
  ; l is list
  (lambda (l)
    (cond
      ; If null, is true
      ((null? l) #t)
      ; If car is atom, then recur
      [(atom? (car l)) (lat? (cdr l))]
      ; Else element is not atom
      (else #f))))
```

How It Works

A list will be provided as an argument. A conditional statement is then used. The first statement checks to see if the list is null, if so it will return true. If it is not, it will then check to see if the 'car' of the list is an atom. If this is so, recursion will take place and the argument will be the 'cdr' of the given list. If the element checked is not an atom, false will be returned. This would be in the case of a nested list.

Tracing Execution

Case 1

```
(lat? '(Andrew Ben Charlie))

;1 (#t (lat? '(Ben Charlie)))
;2 (#t (#t (lat? '(Charlie))))
;3 (#t (#t (#t (lat? '()))))
;4 #t
```

1 The provided list is '(Andrew Ben Charlie). 'Andrew is an atom, so true is the result. The function will then call itself but with the 'cdr' of the list.
2 'Ben is also an atom. The function will call itself with the 'cdr' of the list again.
3 'Charlie is also an atom. The function is called again. The 'cdr of the list an empty list.
4 Because the list is null, true is returned. This means the whole list contained only atoms.

Case 2

```
; (lat? '(Andrew (Ben) Charlie))

;1 (#t (lat? '((Ben) Charlie)))
;2 #f
```

1 The list provided is '(Andrew (Ben) Charlie). 'Andrew is an atom, and so the function will continue with the cdr of the list.
2 False is returned because '(Ben) is not an atom as it is a list itself. The provided list did not contain only atoms.

Member?

The ‘member?’ function identifies whether or not a given element is a member of a given list.

```
(define member?
  ; a is element and lat is list
  (lambda (a lat)
    (cond
      ; if null, a is not present
      ((null? lat) #f)
      (else
       ; either car of lat is equal to a.. if false
       (or (eq? (car lat) a)
           ; start recursion
                (member? a (cdr lat)))))))
```

How It Works

An element is provided along with a list to check if it is part of the list. If the list is null, false will be the final value. This is so because the element wasn’t found. Otherwise an ‘or’ primitive is used. The ‘eq?’ primitive compares the ‘car’ of the list and the element provided. If they are equal, true will be the final answer, identifying the given argument was found. If they are not equal, the function will call itself but the list will be the ‘cdr’ of the list and try and find the given argument.

Tracing Execution

Case 1

```
; (member? 'Joe '(Hamza Amrit Joe Tashan))

;1 (member? 'Joe '(Amrit Joe Tashan))
;2 (member? 'Joe '(Joe Tashan))
;3 #t
```

1 The member to find is ‘Joe and the list is ‘(Hamza Amrit Joe Tashan). As the ‘car’ of the list, ‘Hamza, is not equal to ‘Joe, the function will recurs with the ‘cdr’ of the list.
2 ‘Joe is also not equal to ‘Amrit, the function will call itself again with the ‘cdr’ of the list.
3 The final result is true as ‘Joe was equal to the ‘car’ of the list which was also ‘Joe.

Case 2

```
; (member? 'Joe '(Hamza Amrit Tashan))

;1 (member? 'Joe '(Amrit Tashan))
;2 (member? 'Joe '(Tashan))
;3 (member? 'Joe '())
;4 #f
```

1 The target is ‘Joe again, and the provided list is ‘(Hamza Amrit Tashan). ‘Joe is not equal to the car of the list, ‘Hamza. It will then call the function with the ‘cdr of the list.
2 ‘Joe is also not equal to ‘Amrit.
3 ‘Joe is not equal to ‘Tashan
4 The ‘cdr’ of the list is now an empty list. Because of this, the result is false as the target could not be found in the given list.

Chapter 3

Rember

The function ‘rember’ stands for remove a member. It takes an element and a list as arguments. It makes a new list with the first occurrence of the atom removed from the list.

```
(define rember
  ; takes element and list
  (lambda (a lat)
    (cond
      ; if list is null, return empty list
       ((null? lat) '())
       ; if car of list is equal to a, return cdr of lat
       ((eq? (car lat) a) (cdr lat))
       (else
        ; cons the car of lat, then recurs with a and cdr of lat
        (cons (car lat) (rember a (cdr lat)))))))
```

How It Works

The function takes an element that will be removed from a list, and a list. The list will be checked to see if it is null and if so an empty list will be returned. Next it will check to see if the ‘car’ of the list is equal to the given element. If so, the function will end with the ‘cdr’ of the list being returned, thus removing the target. If the ‘car’ of the list is not equal to the given element, the ‘car’ will be ‘cons’ed and the function will be called with the ‘cdr’ of the list. The function will continue until the target is removed, or it was not found. If not, the list will be rebuilt back to normal.

Tracing Execution

Case 1

```
(rember 'a '(a b c))

;1 '(b c)
```

1 The given target is ‘a and the list is ‘(a b c). As the target and the ‘car’ of the list are equal, the function will return the ‘cdr’ of the list. ‘ a is removed.

Case 2

```
(rember 'b '(a b c))

;1 (cons 'a) (rember 'b '(b c))
;2 (cons 'a '(c))
;3 '(a c)
```

1 The given arguments are ‘b and list ‘(a b c). As the ‘car’ of the list ‘a, is not equal to the target ‘b, the ‘car’ will be ‘cons’ed and the function will be called with the list being the ‘cdr’.
2 The ‘car’ of the list equals ‘b, so the ‘cdr’ of the list is returned.
3 ‘a is then appended to the front of the list, giving ‘(a c).

Case 3

```
(rember 'c '(a b c))

;1 (cons 'a) (rember 'c '(b c))
;2 (cons 'a) (cons 'b) (rember 'c '(c))
;3 (cons 'a) (cons 'b '())
;4 (cons 'a '(b))
;5 '(a b)
```

1 The target is ‘c and the list is ‘(a b c). The ‘car’ of the list is not equal to ‘c. So the ‘a is ‘cons’ed and the function is called again with the ‘cdr’ of the list.
2 ‘c is again not equal to the ‘car’ of the list. ‘b is then ‘cons’ed and the recursion occurs again.
3 ‘c is equal to the car of the list, so an empty list is returned.
4 ‘b is appended to the list.
5 Finally ‘a is appended to the list, giving ‘(a b).

Firsts

The 'firsts' function takes a list, which can be null or non-empty nested lists. It builds another list composed of the first expressions of each nested list.

```
(define firsts
  ; l is list with nest lists
  (lambda (l)
    (cond
      ; if list is null, return empty list
      ((null? l) '())
      (else
       ; else cons the car of the car of l and recurs with cdr of l
       (cons (car (car l)) (firsts (cdr l)))))))
```

How It Works

The function will take a single list as the argument. A check will take place to see if the list is null, if so, the function will return an empty list. An else statement is then triggered if the list is not null. The 'car' of the 'car' of the list will be 'cons'ed. This essentially means the first element of the nested list will be stored. After this, the function will recur with the argument being the 'cdr' of the list. The function will continue until the list is null. Then all the nested 'car's will be 'cons'ed to produce a new list.

Tracing Execution

```
(firsts '((Adam Bob) (Bob Charlie) (Charlie David))

;1 (cons 'Adam) (firsts '((Bob Charlie) (Charlie David))
;2 (cons 'Adam) (cons 'Bob) (firsts '((Charlie David)))
;3 (cons 'Adam) (cons 'Bob) (cons 'Charlie) (firsts '())
;4 (cons 'Adam) (cons 'Bob) (cons 'Charlie '())
;5 (cons 'Adam) (cons 'Bob '(Charlie))
;6 (cons 'Adam '(Bob Charlie))
;7 '(Adam Bob Charlie)
```

1 The argument is '((Adam Bob) (Bob Charlie) (Charlie David)). First the car of the first nested list, which is 'Adam, is 'cons'ed. Recursion then takes place with the 'cdr' of the list as the argument.
2 Next, the 'car' of the next nested list, which is 'Bob, is 'cons'ed. Recursion with the 'cdr' of the list occurs again.
3 Recursion occurs again, but now with an empty list.
4 An empty list has been returned.
5 – 7 All the atoms are appended with 'cons' to produce a new list without any nested lists.

InsertR

The ‘insertR’ function takes 3 arguments. It takes 2 elements, ‘new’ and ‘old’, and a list. The function builds a list with the element, ‘new’, inserted to the right of the first occurrence of ‘old’.

```
(define insertR
  ; takesa new element, and old element to insert
  ; next to and a list
  (lambda (new old lat)
    (cond
      ; if list is null give empty list
      ((null? lat) '())
      (else
       (cond
         ; if car of list equals old
         ((eq? (car lat) old)
          ; first cons new then old to cdr of lat
          (cons old (cons new (cdr lat))))
         (else
          ; cons car lat, then recurse with cdr of lat as lat
          (cons (car lat)
                (insertR new old (cdr lat)))))))))
```

How It Works

First, a check will be done to see if the list is null, if so, and empty list will be returned. Otherwise, the ‘car’ of the list is checked to see if it is equal to ‘old’. If they are equal, ‘new’ then the ‘old’ will be ‘cons’ed to the ‘cdr’ of the list. This results with the ‘new’ element, situated on the right of the ‘old’ element. The function will terminate at this point. If this is not the case, the ‘car’ of the list is ‘cons’ed, then recursion will take place with the ‘cdr’ of the list. A new list will be constructed with the ‘new’ appended in the list.

Tracing Execution

```
(insertR 'cream 'and '(I like cookies and ice-cream))

;1 (cons 'I) (insertR 'cream 'and '(like cookies and ice-cream))
;2 (cons 'I) (cons 'like) (insertR 'cream 'and '(cookies and ice-cream))
;3 (cons 'I) (cons 'like) (cons 'cookies) (insertR 'cream 'and '(and ice-cream))
;4 (cons 'I) (cons 'like) (cons 'cookies) (cons 'and) (cons 'cream '(ice-cream))
;5 (cons 'I) (cons 'like) (cons 'cookies) (cons 'and '(cream ice-cream))
;6 (cons 'I) (cons 'like) (cons 'cookies '(and cream ice-cream))
;7 (cons 'I) (cons 'like '(cookies and cream ice-cream))
;8 (cons 'I '(like cookies and cream ice-cream))
;9 '(I like cookies and cream ice-cream)
```

1 The arguments provided are ‘cream as ‘new’, ‘and as ‘old’ and the list ‘(I like cookies and ice-cream). The ‘car of the list was not equal to ‘and, so the ‘car’ of the list is ‘cons’ed, then recursion takes place with the ‘cdr’ of the list.
2 The ‘car’ of the list is still not equal to ‘old’, so it will be ‘cons’ed and recursion takes place again with the ‘cdr’ of the list.
3 The ‘car’ is again ‘cons’ed and recursion occurs in the same way.
4 Now that the ‘car’ of the list is equal to ‘old’, first ‘cream’ which is ‘new’ will be ‘cons’ed first, the ‘old’ will then be ‘cons’ed to the cdr of the list.
5 – 7 All the atoms are then ‘cons’ed to create a list where ‘cream’ on the right of ‘and’.

InsertL

This function is very similar to ‘insertR’ and takes the same amount of arguments. The ‘insertL’ inserts ‘new’ to the left of the first occurrence of ‘old’.

```
; similar to insertR BUT....
(define insertL
  (lambda (new old lat)
    (cond
      ((null? lat) '())
      (else
       (cond
         ((eq? (car lat) old)
          ; ...cons old first, then new to the cdr of lat
          (cons new (cons old (cdr lat))))
         (else (cons (car lat)
                     (insertL new old (cdr lat)))))))))
```

How It Works

This functions nearly the same way as ‘insertR’, but it inserts ‘new’ on the left of ‘old’. This is because when the ‘car’ of the list is equal to ‘old’, ‘old’ will be ‘cons’ed first to the ‘cdr’ of the list, then ‘new’ will be ‘cons’ed. This results with a new list where the ‘new’ element is placed on the left of ‘old’.

Subst

This function takes 3 arguments, 2 elements ‘new’ and ‘old’ and a list. The aim of the function is to replace ‘old’ with ‘new’ and construct a new list.

```
(define subst
  ; takes an element, old element to replace and a list
  (lambda (new old lat)
    (cond
      ; if list is null, return empty list
      ((null? lat) '())
      (else
       (cond
         ; if car lat equals old
         ((eq? (car lat) old)
          ; then cons the new and return cdr lat
          (cons new (cdr lat)))
         (else
          ; else, cons car of lat, then recurse with new old and cdr of lat
          (cons (car lat) (subst new old (cdr lat)))))))))
```

How It Works

First, the given list will be checked to see if it is null, and if so an empty list will be returned. If this is not so, the ‘car’ of the list and ‘old’ will be evaluated with ‘eq?’. If they are equal, then the ‘new’ will be ‘cons’ed to the ‘cdr’ of the list. This removes the ‘old’ element, and replaces it with ‘new’. If these are not equal, the ‘car’ of the list will be ‘cons’ed and then recursion will take place, with the ‘cdr’ of the list.

Tracing Execution

```
(subst 'love 'like '(I like programming))

;1 (cons 'I) (subst 'love 'like '(like programming))
;2 (cons 'I) (cons 'love '(programming))
;3 (cons 'I '(love programming))
;4 '(I love programming)
```

1 The arguments are ‘love as ‘new’, ‘like is ‘old’ and the list is ‘(I like programming). The ‘car’ of the list is not equal to ‘old’. The ‘car’ is then ‘cons’ed and the function is called again with the ‘cdr’ of the list.
2 Now the ‘car’ of the list equal to ‘old’, ‘love which is ‘new’ is ‘cons’ed to the ‘cdr’ of the list.
3 – 4 All atoms are then ‘cons’ed to create a new list without the ‘old’ element.

Subst2

This function works similar to that first ‘Subst’. This takes 4 arguments, new, o1, o2 and a list. Similar to the first subst, it will replace the first occurrence of either o1 or o2. A new list is produced with the new element replacing either occurrence.

```
(define subst2
  ; takes an element, occurane 1 and 2 from list
  ; and a list
  (lambda (new o1 o2 lat)
    (cond
      ; if list is null, return empty list
      ((null? lat) '())
      (else
       (cond
         ; if either car lat is equal to o1 or o2
         ((or (eq? (car lat) o1) (eq? (car lat) o2))
              ; then cons the new to the cdr of lat
              (cons new (cdr lat))))
          (else
           ; else, cons the car of lat, then recrus with cdr of lat
           (cons (car lat) (subst2 new o1 o2 (cdr lat))))))))
```

How It Works

First, the list will be checked to see if it is null, if so, return an empty list, and otherwise continue. An ‘or’ statement is then used to evaluate if the ‘car’ of the list is either equal to ‘o1’ or ‘o2’. If any of these are true, ‘new’ will be appended to the ‘cdr’ of the list, thus removing o1 or o2. If this is not so, the ‘car’ will be ‘cons’ed, then recursion will take place with the ‘cdr’ of the list.

Tracing Execution

```
(subst2 'love 'hate 'videogames '(I really hate videogames))

;1 (cons 'I) (subst2 'love 'hate 'videogames '(really hate videogames))
;2 (cons 'I) (cons 'really) (subst2 'love 'hate 'videogames '(hate videogames))
;3 (cons 'I) (cons 'really) (cons 'love '(videogames))
;4 (cons 'I) (cons 'really '(love videogames))
;5 (cons 'I '(really love videogames))
;6 '(I really love vidoegames)
```

1 ‘love is ‘new’, ‘hate is ‘o1’ and ‘videogames is ‘o2’ and the list is ‘(I really hate videogames). Because the ‘car’ of the list is not equal to either o1 or o2, it will be ‘cons’ed then recursion occurs.
2 The ‘car’ of the list is still not equal to either ‘o1’ or ‘o2’, and so the ‘car’ is ‘cons’ed. Recursion occurs in the same way again.
3 Now that the ‘car’ is equal to o1 in this case, ‘love is now ‘cons’ed to the ‘cdr’ of the list. This will remove ‘o1’.
4 – 6 All the elements are appended to the list, removing o1 and placing ‘love’ in its place.

Multirember

This function takes 2 arguments, a target 'a' and a list. The functions works by removing all the occurrences of 'a' in 'lat' instead of just the first occurrence.

```
(define multirember
  ; takes an element, and a list
  (lambda (a lat)
    (cond
      ; if list is null, return empty list
      ((null? lat) '())
      (else
       (cond
         ; if car lat is equal to a, then start recursion with cdr of lat
         ; NOTE, will return here till list is null
         ((eq? (car lat) a) (multirember a (cdr lat)))
         (else
          ; otherwise, cons the cdr of lat, then recrus with cdr of lat
          (cons (car lat) (multirember a (cdr lat)))))))))
```

How It Works

This works nearly the same way as the previous rember function. The key is, instead of just returning the 'cdr' of lat, it will go on to recursion with the 'cdr' so that all the occurrences of 'a' are removed from the list. The list will be reconstructed until the list is null.

Tracing Execution

```
(multirember 'icecream '(chocolate icecream and strawberry icecream and
vanilla icecream))

;1 (cons 'chocolate) (multirember 'icecream '(icecream and strawberry
icecream and vanilla icecream))
;2 (cons 'chocolate) (multirember 'icecream '(and strawberry icecream and
vanilla icecream))
;3 (cons 'chocolate) (cons 'and) (multirember 'icecream '(strawberry
icecream and vanilla icecream))
;4 (cons 'chocolate) (cons 'and) (cons 'strawberry) (multirember 'icecream
'(icecream and vanilla icecream))
;5 (cons 'chocolate) (cons 'and) (cons 'strawberry) (multirember 'icecream
'(and vanilla icecream))
;6 (cons 'chocolate) (cons 'and) (cons 'strawberry) (cons 'and)
(multirember 'icecream '(vanilla icecream))
;7 (cons 'chocolate) (cons 'and) (cons 'strawberry) (cons 'and) (cons
'vanilla) (multirember 'icecream '(icecream))
;8 (cons 'chocolate) (cons 'and) (cons 'strawberry) (cons 'and) (cons
'vanilla) (multirember 'icecream '())
;9 (cons 'chocolate) (cons 'and) (cons 'strawberry) (cons 'and) (cons
'vanilla '())
;10 (cons 'chocolate) (cons 'and) (cons 'strawberry) (cons 'and '(vanilla))
;11 (cons 'chocolate) (cons 'and) (cons 'strawberry '(and vanilla))
;12 (cons 'chocolate) (cons 'and '(strawberry and vanilla))
;13 (cons 'chocolate '(and strawberry and vanilla))
;14 '(chocolate and strawberry and vanilla)
```

1 The member to be removed is ‘icecream. As the ‘car’ of the list is not equal to ‘icecream, it will be ‘cons’ed. Recursion takes place with the ‘cdr’ of the list.
2 – 7 Whenever the ‘car’ equals ‘icecream, recursion takes place the ‘cdr’ of the list, this ensures that ‘icecream is removed from the list. Whenever ‘icecream is not found, the ‘car’ is then ‘cons’ed to ensure that the list can be rebuilt.
8 - 14 Eventually an empty list is returned when the ‘cdr’ of the list was null. All the elements stored will be appended to the empty list with all elements equalling ‘icecream removed.

MultiinsertR

Like insertR, a new element is inserted to the right of all the occurrences of ‘old’ instead of just the first.

```
(define multiinsertR
  ; takes a new element, old reference, and list
  (lambda (new old lat)
    (cond
      ; if list is null, return empty list
      ((null? lat) '())
      (else
       (cond
         ; if car of list is equal to old
         ((eq? (car lat) old)
          ; then cons new then old and recurs with cdr of lat
          (cons old) (cons new (multiinsertR new old (cdr lat))))
         (else
          ; else cons car of lat, then recurs with cdr of lat
          (cons (car lat) (multiinsertR new old (cdr lat)))))))))
```

How It Works

When the ‘car’ of the list is equal to ‘old’, first ‘new’ then ‘old’ will be ‘cons’ed so that the ‘new’ element is placed on the right of ‘old. Instead of ending the function by returning the ‘cdr’ of the list, recursion takes place with the ‘cdr’ so that all the occurrences of ‘old’ can have ‘new’ placed to the right of it. This function will end when an empty list is returned so that all stored elements can be appended to it, along with ‘new’ on the right of ‘old’.

Tracing Execution

```
(multiinsertR 'cake 'chocolate '(chocolate and more chocolate))

;1 (cons 'chocolate) (cons 'cake) (multiinsertR 'cake 'chocolate '(and more
chocolate))
;2 (cons 'chocolate) (cons 'cake) (cons 'and) (multiinsertR 'cake
'chocolate '(more chocolate))
;3 (cons 'chocolate) (cons 'cake) (cons 'and) (cons 'more) (multiinsertR
'cake 'chocolate '(chocolate))
;4 (cons 'chocolate) (cons 'cake) (cons 'and) (cons 'more) (cons
'chocolate)(cons 'cake) (multiinsertR 'cake 'chocolate '())
;5 (cons 'chocolate) (cons 'cake) (cons 'and) (cons 'more) (cons
'chocolate)(cons 'cake '())
;6 (cons 'chocolate) (cons 'cake) (cons 'and) (cons 'more) (cons 'chocolate
'(cake))
;7 (cons 'chocolate) (cons 'cake) (cons 'and) (cons 'more '(chocolate
cake))
;8 (cons 'chocolate) (cons 'cake) (cons 'and '(more chocolate cake))
;9 (cons 'chocolate) (cons 'cake '(and more chocolate cake))
;10 (cons 'chocolate '(cake and more chocolate cake))
;11 '(chocolate cake and more chocolate cake)
```

1 The arguments are ‘cake as the new element, ‘chocolate as old and a list ‘(chocolate and more chocolate). Immediately the ‘car’ of the list is equal to ‘old’. This results in the ‘new’ or ‘cake to be ‘cons’ed first, then old. The function still continues when recursion takes place.
4 Eventually ‘chocolate is equal to the ‘car’ of the list, so ‘cake’ will be appended first, then ‘old’ will be appended to the ‘cdr’ of the list.
5 – 11 An empty list is returned and the function ends. All the stored atoms are appended to the list. The new atom ‘cake will be on the right of ‘chocolate.

MutiinsertL

This function takes a 'new' element to be inserted, 'old' which acts as reference and a list. The 'new' element will be inserted to the left of all the occurrences of 'old'.

```
; similar to multiinsertR BUT...
(define multiinsertL
  (lambda (new old lat)
    (cond
      ((null? lat) '())
      (else
       (cond
         ((eq? (car lat) old)
          ; will cons old first, then new, and then recurs
          (cons new) (cons old (multiinsertL new old (cdr lat))))
         (else
          (cons (car lat) (multiinsertL new old (cdr lat)))))))))
```

How It Works

This function is nearly similar to the multiisertR. The main difference is when the 'car' of the list is equal to the 'old' element. The 'old' element will be appended to the list with 'cons' first, then the 'new' element will be appended afterward. Then the recursion will occur. This ensures that 'new' will always be appended on the left of all the occurrences of 'old'.

Multisubst

This function aims to substitute 'old' with 'new' in all its occurrences in a list.

```
(define multisubst
  ; takes a new element, an old element to sub, and list
  (lambda (new old lat)
    (cond
      ; if list is null, return an empty list
      ((null? lat) '())
      (else
       (cond
         ; if the car of list is equal to old
         ((eq? (car lat) old)
          ; then cons the new and recurs with the cdr of lat
          (cons new (multisubst new old (cdr lat))))
         (else
          ; else, cons the car of lat, then recrus with the cdr of lat
          (cons (car lat) (multisubst new old (cdr lat)))))))))
```

How It Works

Like all previous functions, when the list is null, an empty list will be returned. When 'car' of the list equals 'old', 'new' will be 'cons'ed and recursion will occur with the 'cdr' of the list, thus removing the 'old' element. Once all elements have been checked, the empty list will be returned and all stored elements will be appended with all 'old' being replaced with 'new'.

Tracing Execution

```
(multisubst 'love 'hate '(I hate comics and hate games))

;1 (cons 'I) (multisubst 'love 'hate '(hate comics and hate games))
;2 (cons 'I) (cons 'love) (multisubst 'love 'hate '(comics and hate games))
;3 (cons 'I) (cons 'love) (cons 'comics) (multisubst 'love 'hate '(and hate
games))
;4 (cons 'I) (cons 'love) (cons 'comics) (cons 'and) (multisubst 'love
'hate '(hate games))
;5 (cons 'I) (cons 'love) (cons 'comics) (cons 'and) (cons 'love)
(multisubst 'love 'hate '(games))
;6 (cons 'I) (cons 'love) (cons 'comics) (cons 'and) (cons 'love) (cons
'games) (multisubst 'love 'hate '())
;7 (cons 'I) (cons 'love) (cons 'comics) (cons 'and) (cons 'love) (cons
'games '())
;8 (cons 'I) (cons 'love) (cons 'comics) (cons 'and) (cons 'love '(games))
;9 (cons 'I) (cons 'love) (cons 'comics) (cons 'and '(love games))
;10 (cons 'I) (cons 'love) (cons 'comics (and love games))
;11 (cons 'I) (cons 'love (comics and love games))
;12 (cons 'I (love comics and love games))
;13 '(I love comics and love games))
```

1 'love is the element to be substituted in place of the atom 'hate in all its occurrences. Initially the 'car' of the list is not equal to 'old', so it will be 'cons'ed and saved for later.
2 Now 'car' is equal to 'hate, this causes 'love to be 'cons'ed and a recursion takes place with the 'cdr' of the list, thus substituting 'hate' at this instance.
5 At this point, the 'car' of the list and 'old' are equal, so again, 'love will be 'cons'ed and recursion occurs again with the 'cdr' of the list, and 'hate is again substituted.
7 – 13 An empty list is returned as the list is null, so all the stored elements are appended to the list and all occurrences of 'old' have been replaced with 'love.