UNIVERSITY OF
WEST LONDON

Contents

Chapter 1

Functional Programming

1.1 Introduction

Functional programming is a style of programming language which is designed to matched mathematical concept of functions(*just like the way functions are writing in maths which required evaluation of variables*). In contrast, functional programming have no-variable, no iterating concepts and assignment statements e.g like while, for & '=' statement which are founds in other imperative programming languages like Java ,C/C++ and JavaScript. Identifiers are use in functional programming to bond a value to it,by contrast functional programming suppose not to have no variable side effect.

1.1.1 Arithmetic Operators

The arithmetic operators in racket programming language are listed below in tabular form comparing with the normal maths form.

Operator Name	**Maths**	**Expression**	**Racket**	**Expression**
Addition	+	3 + 5	+	(+ 3 5)
subtraction	-	3 - 5	-	(- 3 5)
Division	/	3 / 5	/	(/ 3 5)
Multiplication	*	3 * 5	*	(* 3 5)
Assignment	=	a = 5	let	(let ([a 5]) expr..)

Note: I did not use set! function ,instead I use the **let** function because it does not cause a side effect on identifier value, but set! function does.

These are the basic arithmetic operators in racket functional language.

.

1.1.2 Logical Operators

Racket programming also have the logical operator to enable the programmer make logical decisions which will be tabular below:

And Operator

The and returns true only if the both condition or operand are true else false.

A	**B**	**A and B**
True	True	True
True	False	False
False	True	False
False	False	False

Syntax

(and (condition) (condition))
It returns boolean value #t or #f

Or Operator

The or returns true if any one the operand is true else false.

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

Syntax

(or (condition) (condition))

It returns boolean value #t or #f

not Operator

The not negate the operand to the opposite .

A	not A
True	False
False	True

Syntax

(not (s-condition))

It returns boolean value #t or #f

1.1.3 Control Structure

The control structure in rackets help you to control the order how program is executed. There are two may basic control structure mostly use known to me.

if statement

The if statement or functions have three parameter.
Syntax

(if (condition) [true-expression] [false-expression])

- **(condition)**: condition is the condition expression you are testing e.g (= test 20) it returns #t or #f
- **[true-expression]** : this expression is evaluated when the condition is true
- **[false-expression]** : this expression is evaluated when the condition is false

Examples of the **if statement** will be demonstrate

```
#lang racket

(if
 (= 4 5);condition: check if 4 is equal to 5
 "4 is equal to 5"; condition is true
 "4 is never equal to 5"; false
 );close the if bracket
```

cond statement

The cond statement is use to execute multi-condition statement, in other programming language it act as the if-else-if ..... statement.
Syntax
(cond
[(condition) (expression)]
[(condition) (expression)]

.....
[else (expression)])

Exercises

1.1.4 Atom

The basic primitive data types in functional programming are call atoms. which are either a real values, integer , string and symbols.
Note: a Null values is not an atom neither is list value an atom.

Examples

Examples of an atom is

- hello is this an atom? Yes it is an atom because it is a string of characters
- 123 is this an atom? Yes of course it is an atom because it strings of characters
- 123 is this atom? Yes it is an atom because is an integer number
- 0.3 is this an atom? Yes it is an atom because is a real number.
- Is a Null value an atom? No way!, is not an atom because it is not a primitive type.
  Null values: is a value that does not exist so is not a primitive data type.

The atom?

This atom? is user-defined function, since Racket programming language did provided us with one.
This function will enable us to check if an S-expression value is an atom or not.

```
#lang racket
;Time/Data 12:15  11th Feb 2014
;Program : atom? function testing

;This export the atom? function to be reused
;by other racket files that require it.

(provide atom?)

;This is the atom function declaration
(define atom?
  (lambda (x)
    (and
     (not (pair? x))
     (not (null? x)))))
```

Examples

Testing the atom? function :

```
#lang racket
;Testing the atom function with values
    ~~~~~~~~~~~~~~~~~~~~~~

;test with  pair values
(printf "1) is this an atom ~a \n" (
   atom? '(23 2)))

;test with string value
(printf "2) is this an atom  ~a \n" (
   atom? (quote johnson)))

;test with numbers value
(printf "3) is this an atom  ~a \n" (
   atom? 23))

;testing with null value
(printf "4) is this an atom  ~a \n" (
   atom? null))
```

The test result is show below:

```
Welcome to DrRacket, version 5.3.4 [3m].
Language: racket; memory limit: 128 MB.
1) is this an atom #f
2) is this an atom  #t
3) is this an atom  #t
4) is this an atom  #f
> |
```

1.1.5 Symbolic-Expression?

An S-expression is an atom or group of atoms,pairs or a list.

Examples

Questions	Answers
• Is true that is an S-expression 'Johnson	• Yes! it is an S-expression because all atom are S-expressions
• How many S-expressions are there in the list object '(1 2 3 4)	• 4 S-expressions since the list contains 4 atoms which are 1,2,3,4
• Guest the number of S-expressions in this list object '((1 2) 3 4)	• 3 S-expressions since the list contains list object which is (1 2) and two atoms which is 3 and 4. A list inside another list is regard as an S-expression since an S-expression is also a list

---

1.1.6 List?

A list is a group of S-expressions or atoms in a quote bracket.

Example:

- '(a b c) : This is a list of 3 items a b c while a, b and c are atoms.
- '((a b) Johnson bogs fowl) : This is a list of 4 items;
  we previously we talk about the number of S-expressions in a list. In this list we have 1 list object and three atom element in the main list which make the list have 4 elements only. now I think you will asking a question in your mind that no? it is a list of 5 items.
  Explanations :
  (a b) is an S-expressions or a list what ever you what to call it. But the main key here is that it is count has one item on the main list same as the rest values johnson,bogs and fowl.

1.1.7 Define A function

in racket , function are defined with the keyword define.
Syntax
(define (function-name arg0 arg1 ...)
.....expression .....
)

Or

(define function-name
(lambda (arg0 arg1 ...)
expression .....
))

Examples A function that will return the area of a cicle when the radius is pass to it

```
#lang racket
;Create variable to hold the PI value
(define PI 3.142)

;create the function

(define area-of-circle?
  (lambda (r)
    (* (* r r) PI)))
```

1.1.8 Recursion

Recursive function are functions which call their self directly or indirectly (from other functions). This concept will use in further section to be able to understand recursion in racket.
Note: In recursive function there must be a control base or certain condition for the program to terminated.

Examples
"The fact? function" that will return the factorial of a number.

```
1  #lang racket
2   (require sgl)
3
4  ;The factorial function !
5  (define fact?
6    (lambda (n)
7      (cond
8        [(<= n 1) 1]
9        [else
10        (* n (fact? (- n 1)))])))
```

Function Explanation
When the function is first called,

- The line 7 of the code will check if n (is 3) is less or equal to 1
  which is not
- Then line 9 will then be executed and multiply n (= 3) to the return value of fact? of n-1 which is 2.

1. (3 * (fact? 2))
   the function call his self again and pass 2 And again the line inner function will execute...

   (a) The line 7 codes check again if n (= 2) is less or equal to 1
   which is not
   (b) Then line 9 will again multiply n (= 2) to the return value of fact of n-1 which is 1 .
      i. (3 * (2 * (fact? 1))) And again the inner function of the first inner function execute again...
      The function call his self and pass 1 to the argument
      A. The line 7 codes check again if n (= 1) is less or equal to 1
      which is yes then it returns 1 and the function terminated
      The final result will look like this and evaluate **(3 * (2 *(1))** = 6

```
Welcome to DrRacket, version 5.3.4 [3m].
Language: racket; memory limit: 128 MB.
> (fact? 3)
6
>
```

This is how recursive functions works as we go along you will understand better , since we are going to be using recursive function a lot in this log book.

1.2 Exercise

Create functions for the following:

- dollar-to-euro
- celsius-to-fahrenheit
- vat
- body-mass-index

1.2.1 Exercise Solution

- codes for dollar-to-euro

```
1  #lang racket
2
3
4    (define dollar-to-euro?;function-name
5     (lambda (d r)    ;parameters
6           (* d  r)))
```

- codes for celsius-to-fahrenheit

```
#lang racket


(define censius-to-fehriheit?
   (lambda (c)
     ;;the function body
     (round (* c 33.80))))
```

- codes for vat

```
#lang racket


(define vat?
  (lambda (amount rate)
    ;calculate the vat and return the result
   (/ (* amount rate) 100) ));end function
```

- body-mass-index

```
#lang racket


(define bmi?
  (lambda (h w)
     (* (/ (/  w h) h) 1.00)))
```

Chapter 2

Racket in-build Functions

The racket have many in-build function but the primitive ones will be discuss in this chapter and how they can be use according to their syntax or laws.

2.1 Laws

These are the common rules and syntax of how to use the primitive functions in Racket programming language.

2.1.1 cons

The cons is a primitive functions which take two parameters and join then to form a pair or a list .
The return values is always a list object

THE LAW
The law of the primitive cons state that , if the second parameter of cons is list it returns a normal list else if the second parameter is an atom it returns an association list the results is always a list

Examples
simple program codes to demonstrate the use of **cons**.

Simple cons programs

```
#lang racket
;Authur Obaro Isreal Johnson
;Time/Data 12:15  11th Feb 2014
;Program : atom? function testing


;joining an atom together with the  cons
    primitive function

(printf "~a - this is an association list
    object\n"(cons 'a 'b));

;joining an atom to a list
(printf "~a - this is a normal  list
    object\n"(cons 'a '( b c)));


;joining two pair (list) to  another pair

(printf "~a -  This is a normal list of 3
    S-expressions" (cons '(1 2) '(3 4)))
```

Output Results

```
Welcome to DrRacket, version 5.3.4 [3m].
Language: racket; memory limit: 128 MB.
(a . b) - this is an association list object
(a b c) - this is a normal  list object
((1 2) 3 4) -  This is a normal list of 3 S-expressions
>
```

With this example I think we now understand what cons does and how to use it.

2.1.2 car

This primitive function which takes a list object as its parameter and return the first element of the list object.

THE LAW
The law of the primitive car state that, is defined only for non-empty list and returns the first element(S-expression) of the list. the return value is an S-expression or an atom

An error will flash when you pass an empty list to the **car** function.

An Examples Simple programs in Racket to demonstrate the use of the **car** primitive function.

Syntax (car lat)
where **lat** is the list object parameter
It returns the first S-expression of the list.

For example if lat is the list '(1 2 3 4 5) (car lat)
will return 1

2.1.3 cdr

The primitive function **cdr** is defined for list object only and it returns a sub-list.

THE LAW
The law of the primitive cdr: is defined only for non-empty list object and its return value is always sub-list of the main list without the first element or e-expression.

```
1  #lang racket
2
3
4  (cdr '(1 2 3 4 5))
5
6  (cdr '((john car) goat love))
7
8  (cdr '(1 2))
```

The output result

```
Welcome to DrRacket, version 5.3.4 [3m].
Language: racket; memory limit: 128 MB.
'(2 3 4 5)
'(goat love)
'(2)
>
```

Note:
This function will be use through out in this log book.

2.1.4 null

The null? is use only with a list object (**Recall that:** *empty list object are not null value*), which takes only one parameter to check if the list is null or not.

THE LAW
The primitive function null? takes one list parameter and check if is null or empty.

It returns #t if the list is null or empty else #f.

Examples programs to demostrate the use of null?

```
1  #lang racket
2
3
4
5  (define is-null
6    (lambda (l)
7      (if (null? l)
8          #t; if null return true
9          #f;if not-null return false;
10         )));end function
```

The output result

```
Welcome to DrRacket, version 5.3.4 [3m].
Language: racket; memory limit: 128 MB.
> (is-null '())
#t
> (is-null '(1 2 3))
#f
> (is-null 3)
#f
>
```

Note:
Instead of the user-defined function the null? can be use instead , but for clarity we will be using the is-null function instead which reads nicely.

2.1.5 eq

THE LAW
The primitive function eq? takes two atom parameters and compare them if they are equal in binary or not.

its return #f ; if the parameters are not equal and #t if they are equal

Examples programs to demonstrate the use of the *eq?* primitive function:

```
#lang racket

;Let compare if two non-numeric value are
    same

(eq? "johnson" "obaro")
(eq? "A" 64)
(eq? "A" "a")
(eq? "johnson" "Johnson")
(eq? "johnson" "johnson")
```

The output result

```
Welcome to DrRacket, version 5.3.4 [3m].
Language: racket; memory limit: 128 MB.
#f
#f
#f
#f
#t
>
```

Note:
This function is very useful on non-numeric compared values.

By now we must have be use to the following primitive functions and how to use them...

- cons
- cdr
- car
- null?
- and the eq?

2.2 Seminal Exercises 2 and Solutions

1. We want to write a program to calculate the amount of scholarship awarded based on the GPA of the student according to the following conditions: A GPA less than 2.5 earns nothing A GPA which is greater than or or equal to 2.5 but less than 3.0 is awarded $300 A GPA which is greater than or equal to 3.0 but less than or equal to 4.0 is awarded $350

```
#lang racket

(define gpa-award
  (lambda (gpa)
    ;create a variable to hold the remark
    (define remark "");
    (cond
      ;check if gpa is greater than 4.0
      [(> gpa 4.0)(set! remark "invalid gpa")]
      ;check if the gpa is between 4.1 to 3.0
      [(and (<= gpa 4.0) (>= gpa 3.0) )
       (set! remark "Âč350")]
      ;check if the gpa is between 3.0 and 2.5
      [(and (< gpa 3.0) (>= gpa 2.5))
       (set! remark "Âč300") ]
      ;else set remark  to Âč0.0
      [else (set! remark "Âč0.0")]
      );;end cond bracket

    remark))
```

2. Write a program that provides information about a person's weight using your BMI program and the following conditions: Underweight is less than 18.5 Normal weight is from 18.5 to 25 Overweight is from 25 to 30 Obese level 1 is from 30 to 35 Obese level 2 is from 35 to 40 Obese level 3 is greater than 40
3. Write a program that provides information about t tube ticket price according to the following conditions:
   (For simplicity we use ticket prices for going from Ealing Broadway to Hammersmith (Zone 3 to 2)) $ 4.70 Cash payment Anytime $ 1.60 Oyster Monday to Friday 0630 - 0930 $ 1.60 Oyster Monday to Friday 1600 - 1900 $ 1.50 Osyter All other times
   Hint1: You can use Monday = 1 to Sunday = 7
   Hint2: You can use 0 for 12AM and 2359 for 11:59PM

4. Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.
5. Define a function which checks whether an atom is the last S expression of a list or not.

The print-list? unction

From the primitive functions we are going to write a program that will take a list as its parameter and prints is element.

```
#lang racket


;let a create a function call print-list
    with a parameter
;l -represent the list object t print

(define print-list?
  (lambda (l)
    (cond
      ;check if the list is null or not
      [(null? l) #f]
      ;else
      [else
       ;get the first element with car
          and print the elements
       (printf "~a \n" (car l))
       ;recursed the function with the
          value list from cdr function
       (print-list? (cdr l))
       ])

    ))
```

The output result

```
Welcome to DrRacket, version 5.3.4 [3m].
Language: racket; memory limit: 128 MB.
> (print-list? '())
#f
> (print-list? '(1 2 3 4))
1
2
3
4
#f
>
```

Note:
This function is derived from the primitive type, and more complex functions can also be derived.

```
#lang racket
;Author : obaro


;required the atom file
(require "atom.rkt");


;A function to check if the S-Expresions
    in a list is an atom.
(define lat?
  (lambda (l)
     (cond
          ;check if the list is null
       ((null? l) #t)
           ;check if the first element in
               the list is an atom
       ((atom? (car l))
        ;if the element is an atom,recall
            the function with
        ;cdr return value of the main
           list.
        (lat? (cdr l)))
       [else #f])));else not an atom
          return false
```

The output result

```
Welcome to DrRacket, version 5.3.4 [3m].
Language: racket; memory limit: 128 MB.
> (lat? '())
#t
> (lat? '(1 2 3))
#t
> (lat? '((1 2) 4 5))
#f
>
```

Note:
This function is derived from the primitive type, and more complex functions can also be derived.

Chapter 3

User Defined Function (from primitive types)

In this section , I am going build more complex function from the primitive ones that was study before. Then we will later look at some commandments and good practice style of building a function in functional programming.

3.1 The Commandments

These are the good practice,guides or conventions that should be follow to be able to write a good functions in functional programming.

3.1.1 The First Commandment

Is based on the null? primitive functions; states: Always ask null in an expression when working with functions.

Examples:
Create a function called lat? that will check if the S-expressions in a list object are atom.

The lat? function source codes

```
1  #lang racket
2  ;Author : obaro
3
4
5  ;required the atom file
6  (require "atom.rkt");
7
8
9  ;A function to check if the S-Expresions
       in a list is an atom.
10 (define lat?
11   (lambda (l)
12     (cond
13           ;check if the list is null
14       ((null? l) #t)
15           ;check if the first element in
                 the list is an atom
16       ((atom? (car l))
17        ;if the element is an atom,recall
              the function with
18        ;cdr return value of the main
             list.
19        (lat? (cdr l)))
20       [else #f])));else not an atom
          return false
```

Explanations of the function
The lines of the codes are explained below:

1. # **lang racket :** This specified the current language in use so that racket can know how to compiled it.

6. **(require "atom.rkt)** : This line of code includes the atom? function which was previously defined in the **atom.rkt** file.
   This **atom?** function will be use in this **lat?** function that is why I included it.

14. The line of code ask if the list object pass to the lat? as a parameter is a null or not . Which is the rule of the first commandment.
    Always remember to do this while working with function.

16. The line of code ask if the first element is an atom or not , by calling the external user-defined atom? function in the **atom.rkt** file which we previously required.

19. The function re-call itself (recursive function) and pass the cdr of the main list as the new parameter.

The following is the running test for the **lat?** function with atom

```
Welcome to DrRacket, version 5.3.4 [3m].
Language: racket; memory limit: 128 MB.
> (lat? '(meet dog food car))
#t
>
```

Let look at another function that ask for null again as the first expression;
The name of the function will be called the is-member with two parameter

```
#lang racket
;include modues here


;Author obaro
;This function will check is the first
;argument is a member of the list object

(define is-member?
  (lambda (a lst)
    ;check if the list is null
    (cond
      ;check if null
      [(null? lst) #f]
      ;check if it is equal
      [(eq? a (car lst)) #t]
      [else
       ;recursed the function again
        (is-member? a (cdr lst))
       ])));end
```

; **Code Explanation**

9. This line defined the function name **is-member**
10. The line defined the function parameters using the lambda function
14. Check if the function is null , which is the first expression in the function block. obeying the first commandment.
16. check if the list object first element matched the value of a, if yes it return #t and terminated.
19. else it call the function again, and pass the cdr of the list to the function and the normal value of a.

    This will continue util the cdr will return an empty list or is value matches the a-value

Note: always remember to ask for null to avoid errors or function crashed.

Chapter 4

Deeply Understand Of Recursion

In this chapter will are going to be talk more about how recursive functions worked with examples; and why we need to use recursive functions and other commandments with examples.

4.0.2 The Second Commandment{ *of cons?*}

This commandment is based on the cons which state: always try to use cons function to build a list object.
Let create a function called the **rember** that will removed an s-expression.

Examples:

```
1  #lang racket
2  (provide rember?);make it importable
3
4  ;The function will find the a-value in
5  ;the l=list object and remove it when
       found.
6  (define rember?
7    (lambda (a l)
8      (cond
9        ;check if the list is empty or not
10       ((null? l) (quote ()))
11       ;check if the a-value is equal to
             the car of the list
12       ((eq? a (car l))  (cdr l))
13       (else
14        ; Maintain the list or build the
              list with cons function
15        (cons (car l) (rember? a (cdr l)))
              ))))
```

Code Explanation
Since we are interested in how the function builds a list with the cons and how it works , we are going to take about the recursive part later, but for now let understand what the functions does.

1. **#lang racket** : specific the language in used.
2. **(provide rember?)**: this make the function accessible in other module when imported.
6. **(define rember?** : defined the name of the function.
10. check if the list is **null** : this is the first commandment.
12. This codes check if the search value matches the list element,then return the cdr of the list object to discard the first element.
16. else if there is no match , then the function call itself(recursive function) and cons the first element that will be discard with the cdr function , with its return type.
These will be explained in details below with some values:

running example of the working function rember?

```
Welcome to DrRacket, version 5.3.4 [3m].
Language: racket; memory limit: 128 MB.
> (rember? 'sister '(boy girl sister mother))
'(boy girl mother)
>
```

The returning value of this function is a list , which have removed the 'sister value from the list.
The question is this ? How did this function recursively worked...

The function works this way:
let a=sister and l='(boy girl sister mother)

when the function is first called.the following lines:

10. The function check if the list is null **((null? l) (quote ()))**
    let substitute the value of l which is **'(boy girl sister mother)** into the code it will then be
    ((null? '(boy girl sister mother)) (quote ()))
    The return value here will be false, since the l is not null;

12. This will be the next line of code to execute: **((eq? a (car l)) (cdr l))** which check if the a which is **sister** is equal to the value of the car l which is **boy** of-course not , they are not equal.

15. else the following line of code will be execute:
    (cons (car l) (rember? a (cdr l)))
    I am going to carefully analysis this piece of codes to get what is happening here...

    (cons ;this joined an atom to a list
    (car l) ;this return the first element **'boy**
    (rember? a (cdr l)));This re-called the function rember? and pass a='sister and (cdr l) which is now (girl sister mother)

    Substitute the piece of code with the real value gives us this;
    (cons 'boy (rember? 'sister '(girl sister mother)))
    The function recall itself a gain, that means it copy itself.
    Now let a=sister and l='(girl sister mother)

    15.1. The function will repeat the steps one again:
    The function check if the list is null **((null? l) (quote ()))**
    let substitute the value of l which is **'(girl sister mother)** into the code it will then be
    ((null? '(girl sister mother)) (quote ()))
    The return value here will be false, since the l is not null;

    15.2. This will be the next line of code again to execute: **((eq? a (car l)) (cdr l))** which check if the a which is **sister** is equal to the value of the car l which is **girl** of-course not , they are not equal.

    15.3. else the following line of code will be execute again:
    (cons (car l) (rember? a (cdr l)))

    Substitute the piece of code with the real value into the code;
    (cons 'girl (rember? 'sister '(sister mother)))
    The function recall itself a gain, that means it copy itself.
    Now let a=sister and l='(sister mother)

    15.3.1. The function call it self again to repeat step one:
    The function check if the list is null **((null? l) (quote ()))**
    let substitute the value of l which is **'(sister mother)** into the code it will then be
    ((null? '(girl sister mother)) (quote ()))
    The return value here will be false, since the l is not null;

    15.3.2. This will be the next line of code again to execute: **((eq? a (car l)) (cdr l))** which check if the a which is **sister** is equal to the value of the car l which is **sister**, ofcourse yes it is, i mean they are equal. the function will return the list value '(mother) and terminated.

Now let gather all the date and see... the final result.
3.
(cons ' boy
3.3
(cons ' girl
3.3.2
'(mother)

(cons 'boy (cons 'girl '(mother))
= (cons 'boy '(girl mother))
='(boy girl mother)
We just used the cons to build the list again, and that is how recursive function works and help use to build a list object.

4.0.3 The Third Commandment

The commandment is base on the cons which state that: when building a list described the typical element and then cons it with the natural recursion.
This commandment is actual telling you that , you can use cons to append elements to a list with natural recursive.
Example of the insertR function.

Codes Listing of InsertR function

```
1  #lang racket
2  ; make the function visible when required
3  (provide insertR?)
4  ;The function insert the new-value at
5  ;the right of the found old-value
6  (define insertR?
7    (lambda (new old lst)
8     (cond
9       ;check if the lst object is null
10      [(null? lst) '()]
11      [else
12       (cond
13         ;check if the old value is found
14         [(eq? (car lst) old)
15          ;if found add it to the front of
                the old value
16          (cons old (cons  new (cdr lst)))
               ]
17         [else
18          ;else recursive the function and
                keep the list value
19          (cons (car lst) (insertR? new
               old (cdr lst))) ])])))
```

Code Explanation The function is use to insert an S-expression or atom , says **new** value to the right of the list object.

1. #lang **racket**: defined the language template in use.
3. **(provide insertR?)** : make the insertR? function visible when it is required in another racket file.
6. This line defined the function name **insertR?**.
7. Defined the function's parameters old, new and lst.
10. check if the list is null or not
14. check if the **car** of the **lst** is equal to the old (parameter) value passed.
16. if line (14) is true , then it execute this line of code. **cons old (cons new (cdr lst)))** . This is the third commandment but the only thing here is that it did not recurred. so let check the next line.
20. Take not of this line of code:
    This is the third commandment.
    (cons (car lst) (insertR? new old (cdr lst)))
    What this line of code does is that , it keep track of the first typical element and then cons it to the recursive return value of the function.

    Below is how the line of code works recursively...

Function syntax
(insertR? new old lst);

let assumed that old = **'girl** and new = **boy** and lst = **'(mother father girl sister)** is the list object we want to insert the new value at the right side of the old which is **'girl**, so that the end result will look like this: **'(mother father girl *boy* sister)**

Execute the function (insertR? 'boy 'girl '(mother father girl sister));
The first call of the function the following lines :

10. check if the list is **null** = No is not null, then
11. check if (car of the lst) = **'mother** is equal to the old= **'girl** which is not , then
20. else line(20) is executed , as follow:
    (cons (car '(mother father girl sister)) (insertR? 'boy 'girl (cdr '(mother father girl sister))))
    To simplify this line it will then becomes:
    (cons 'mother (insertR? 'boy 'girl '(father girl sister)))
    The function just call it self again , let trace the execute of the function.
    (insertR? 'boy 'girl '(father girl sister))

(a) The line(10) of the function will check again if the lst is **null**= No is not null, then

(b) the second line check if (car of the lst) = **'father** is equal to the old= **'girl** which is not , then

(c) The line (20) will be executed again , now
recalled that **lst** is now = **'(father girl sister)** which makes the codes like this:
(cons (car '(father girl sister)) (insertR? 'boy 'girl (cdr '(father girl sister))))
If we simplify the codes it will becomes:

(cons 'father (insertR? 'boy 'girl '(girl sister)))
The function as recall itself again , with just a pair list. Now let evaluate it again:

---

i. The line(10) of the function will check again if the lst is **null**= No is not null, then
ii. the second line check if (car of the lst) = **'girl** is equal to the old= **'girl** which is! , then ,
iii. The line (16) of the code will execute which is **(cons 'girl (cons 'boy (cdr '(girl sister))))**]
If the line of code is simply it will becomes:
(cons 'girl (cons 'boy '(sister)))

(cons 'girl **'(boy sister)**)
'(girl boy sister) : this will be the final return value of the inner called function.

The second function will evaluate this...
(cons 'father *'(girl boy sister)*)
Result will be this: '(father girl boy sister) this will be return to the first function.

The outermost function will then have this;

(cons 'mother '(father girl boy sister))
Finally the result will be:
'(mother father girl boy sister)

The running snapshot of the function

```
Welcome to DrRacket, version 5.3.4 [3m].
Language: racket; memory limit: 128 MB.
> (insertR? 'boy 'girl '(mother father girl sister))
'(mother father girl boy sister)
>
```

This same function can be swap to insert at the left hand-side...

insertL? function codes

```
#lang racket
(provide insertL?)
(define insertL?
  (lambda (new old lst)
   (cond
     [(null? lst) '()]
     [else
      (cond
        [(eq? (car lst) old)
         (cons new
               (cons  old (cdr lst)))]
        [else
         (cons (car lst) (insertL? new old (cdr lst)))
         ]) ])))
```

4.0.4 The Fourth Commandment

The fourth commandment is base of recursive function principles which states: Always change at least one argument while recurring. It must be changed to be closer to termination. The changing argument must be tested in the termination condition: when using cdr, test termination with null?

The Examples

In this section , we are going to understand , why we need to use recursive function and the reason why it is used. Before you think of using a recursive function , you will be considering :

- The base (*The condition that will be met to make the function terminated*).
  The counter? function

```
1  #lang racket
2
3  (provide counter?)
4  ;This function will counter from "
       from" to "to" number
5  (define counter?
6    (lambda (from to)
7      (cond
8        ;this is the base condition
             that will terminate the
             function
9       [(or (> from to) (= from to)) to
            ]
10      [else
11        (cons from (format "~a\n" from
              ))
12        ;the from argument is increase
               by one so that the base
             condition
13        ;may hold.
14        (counter? (+ from 1) to)])))
```

- The parameter argument (if there is any) that will be change to make the out different to melt the condition.

Graphical Tracing Of Example Code

4.1 Seminal Exercises 3

1. Trace execution for (fac 5)
   Solution

2. Trace execution for (len '(a b c d e))
   Solution

3. Trace execution for (remove 'x '(x y z x))
   Solution

4. Write a function to sum n values such that (sum 3) generates (+ 3 2 1)
   Solution

5. Write a function to sum the range of values between n and m such that (sumrange 1 10) gives 55
   Solution

6. Write a function to count the number of bananas in '(apple banana pear orange banana)
   Solution

Chapter 5

Case Study[The game of Choice]

This is a simple game developed using the function programming concept with opengl library of of the racket programming .

Chapter 6

Recommendations