# PPL CS F 301 & IS F 301
# Function Programming Language (Scheme)

Lab 2

**Introduction to Scheme**

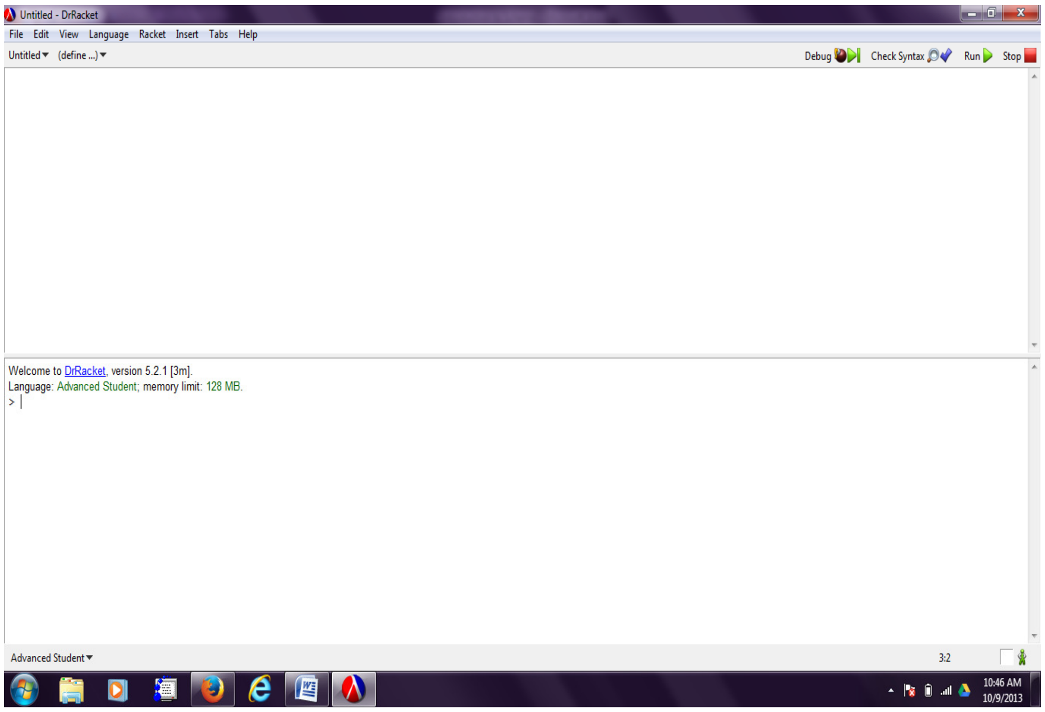
In this lab you will learn to write simple programs using Scheme programming language. After doing examples and exercises, you should be able to start using Scheme.

You will be using DrRacket interactive environment to write Scheme program.

The simplest interaction with Scheme follows a "read-evaluate-print" cycle. A program (often called a *read-evaluate-print loop*, or REPL) reads each expression you type at the keyboard, evaluates it, and prints its value.

## Step1: Getting Started

- Select DrRacket from the program list by pressing start button on your desktop.
- Do not try to explore DrRacket too much today. Just write some program and use **Run** button to execute it.
- You will see a pair of windows.
- Select appropriate language from Language menu --> choose Language--> Advanced Student.



```
Welcome to DrRacket, version 5.2.1 [3m].
Language: Advanced Student; memory limit: 128 MB.
> 
```

## Step2: Writing Expressions

The bottom window says "Welcome to DrRacket." This is the **interactions** window. In the interactions window, type (+ 2 3) and press Enter.

Try all the expression below and observe the output carefully.

- (+ 4 5 6 7 8 9)
- (/ 36 6 3)
- (+ (* 5 4)(- 6 2))
- (+ 5 4(- 6 2))
- (+ (* 2 2 2 2 2))(* 5 5)

- (+ (* 2 2 2 2 2)(* 5 5))
- (* 4 1/2)
- (* 3 1/2)
- (* 3 0.5)
- (/ (* 6/7 7/2) (- 4.5 1.5))
- (+ 0.5 0.75)
- (+ 1/2 3/4)

## Step 3: Defining Functions

- You can write a function without name using keyword lambda.
  **example 1:** ((lambda (x) (* x 3)) 2) or ((λ (x) (* x 3)) 2)
  Note: Use **Ctrl + \ ** to insert **λ**
  **example 2:** ((lambda (x y) (* x y)) 2 3)

- To give name to a function use keyword define (write the definition in **Definition Window** on the top screen and press Run button )
  **Example 3:** (define addone (lambda(x) (+ x 1)))

  call add function as (addone 4) in the Interactive Window

  **Example 4:** (define mul (lambda(x y)(* x y)))

  call mul function as (mul 3 4) in the Interactive Window

  **Example 5:** mul function without lambda

  (define (mulv x y)     (* x y))

  **Example 6:** Function Currying for higher order function

  (define (curried-mul x) (lambda (y) (* x y)))
  (define muly  (curried-mul 3))

  call mul function as (muly 4)

**Example 7:**

- **Write a definition to generate a square of a number.**
- **Use square definition to sum the square of two number $x^2 + y^2$**
  (define (square x) (* x x))
  (define (sumsquares x y)  (+ (square x) (square y)))

## Step 4: Saving your Program

Entering code directly into the interactions window is good for testing out quick ideas but to save your programs, the definitions written in **Definition Window** at the top of the screen go to file menu and choose **Save Definitions**.

**Exercises 1**

**Write expression in Scheme for the following**

- 1.2 × (2 - 1/3) + -8.7
- (2/3 + 4/9) / (5/11 - 4/3)
- 1 + 1 / (2 + 1 /  (1 + 1/2))
- 1 × -2 × 3 × -4 × 5 × -6 × 7
- ($a^2$+$b^2$-4ab)

## Lists in Scheme

- Lists are the basic structured data type in Scheme.
- Parameters are **quoted** to prevents Scheme from evaluating the arguments.
- The simplest way to build a list is with the ***list*** procedure.
  **(list 1 2 3 4 5 6) will generate list (1 2 3 4 5 6)**

- Examples of some of the built in list handling functions in Scheme.

```
'( )                                ;empty list
'(1 2 3 4)                          ;(list 1 2 3 4)
(quote (1 2 3 4))
(list '( ) '(1 2 3 4))              ; (list empty (list 1 2 3 4))
(list (quote())(quote(1 2 3 4)))
(define lst (list 1 2 3 4 5 6))     ;naming a list
'(1 2 ,(+ 3 4))
(quote (1 2 (+ 3 4)))
(list 123 '(4 5 6))               ; (list 123 (list 4 5 6))
'((1 2) (3 4))                        ; check what it is
```

**cons:** Takes two arguments and returns a pair (list). (Note: 2nd argument should be list)

```
(cons 123 '(4 5 6))                  ;(list 123 4 5 6)
(cons '( ) '(1 2 3 4))               ; (list empty 1 2 3 4)
 (cons 1 '(2))                        ;(list 1 2)
 (cons '1 '(2))                      ;(list 1 2)
 (cons '1 '(2 3 4))                  ;(list 1 2 3 4)
 (cons '(1 2 3) '(4 5 6))            ; (list (list 1 2 3) 4 5 6)
```

**(To understand the storage of list in scheme refer the diagram in last page)**

**car :** Returns the first member of a list or dotted pair.

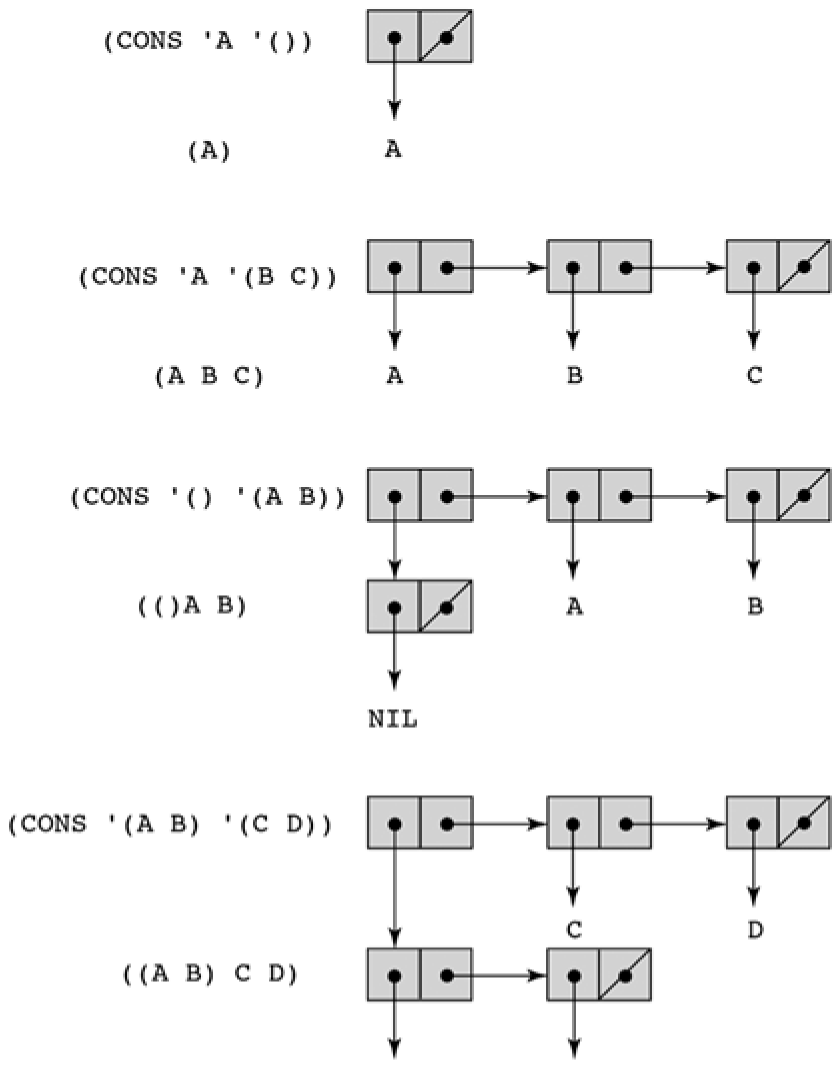
```
(car '(123 245 564 898))                     ;returns 123
(car '(first second third))                  ;returns   first
(car '(programming (is no) more difficult))  ;returns programming
```

**cdr:** Returns the list without its first item.

```
(cdr '(7 6 5))                    ;  (6 5)
(cdr '(it rains every day))       ;  (rains every day)
(cdr (cdr '(a b c d e f)))        ;   (c d e f)
(cddr '(a b c d e f))             ;   short form of previous one
```

```
(car (cdr '(a b c d e f)))                              ;   b
(cadr '(a b c d e f))                                   ;   short form of previous one
(caddr '(a b c d e f))
(cadddr '(a b c d e f))
(caddddr '(a b c d e f))                                 ; nesting is allowed till 4 level in scheme
```

**Result of several CONS**



**"map"** is often considered to be one of the more useful function for manipulating a list. map Takes a list, and a procedure, and applies the procedure to every element of the list:

| |
|---|
| (map (lambda (x) (* x x)) (list 1 2 3 4 5 6)) |
| (map (lambda (x) (>= x 10)) '(11 10 9 5 15 6 0)) |
| (map cadr '((A B) (C D) (E F))) |

**"append"** takes two or more lists and constructs a new list with all of their elements.

| | |
|---|---|
| (append '(1 2) '(3 4)) | Write the output Here |

**Notice append is different from list and cons**

| | |
|---|---|
| (list '(1 2) '(3 4)) | Write the output Here |
| (cons'(1 2) '(3 4)) | Write the output Here |

**"reverse"** takes one list, and returns a new list with the same elements in the opposite order.

| |
|---|
| (reverse '(1 2 3 4)) |
| (reverse '((1 2) (3 4))) |
| (map reverse '((A B) (C D) (E F))) |

**Exercise:**
Below given 3 version of reverse function Implemented using **"append", "cons"** and **"list".**
Check which one will work correctly. Justify

```
(define (myreverseA lst)
 (if (null? lst)
    '()
     (append (myreverseA (cdr lst)) (car lst))))
```

```
(define (myreverseC lst)
  (if (null? lst)
     '()
     (cons (myreverseC (cdr lst)) (car lst))))
```

```
(define (myreverseL lst)
  (if (null? lst)
     '()
     (list (myreverseL (cdr lst)) (car lst))))
```

## Some programs to learn Recursion and Conditions

**(1) Sumall**

```
(define (sumall list)
   (cond
     ((null? list) 0)
     (else (+ (car list) (sumall (cdr list))))))
```

**(2) List of n identical values**

```
(define (makelist n value)
   (cond
     ((= n 0) '())
     (else
       (cons value (makelist (- n 1) value)))))
```

**(3) Remove-duplicates**

```
(define (remove-duplicates list)
    (cond ((null? list) '())
          ((ismember (car list) (cdr list))
           (remove-duplicates (cdr list)))
          (else
           (cons (car list) (remove-duplicates (cdr list))))))
(define (ismember item list)
   (cond ((null? list) #f)
         ((equal? (car list) item) #t)
         (else (ismember item (cdr list))) ) )
```

**Take Home Assignment** **5M**

1. Complete the Exercise1 in page 3 ( all expression should be name less functions. should be written one after the other. Once you click the run button it should give me the output). Write all functions in a single file with file name **exercise1.rkt**

2. Map function perform well with this type of list **(map (lambda(x) (+ 1 x)) '( 1 2 3 4 )**. But fails to work when the list is nested For example **(map (lambda(x) (+ 1 x)) '( 1 (2 3) (4 (5) 6)))**. Modify the map function and name it as "mymap" to work with nested list.

**zip it in one folder and upload it in Nalanda.**
**Note:** All program will be checked with plagiarism tool (specially for checking programs). Any type of copying will be awarded zero (without any enquiry).