

### Ques 4.3

a)' Define the function **tri-area** that returns the area of a triangle when passed the arguments **base** and **height**.

```
( define tri-area  
  (lambda (base height)  
    ( display "Area is " )  
    (* (* base height) 0.5)  
  )  
)
```

```
(tri-area 5 4)
```

```
( define tri-area  
  (lambda (base height)  
    ( (lambda (half)  
      ( display "Area is " )  
      (* (* base height) half)  
    )  
    0.5 )  
  )  
)
```

```
Area is 10.0
```

### Ques 4.3

- a) Define the function **tri-area** that returns the area of a triangle when passed the arguments **base** and **height**. The constant **one-half** should be established by **let**.

```
(define tri-area
  (lambda (base height)
    (lambda (half)
      (display "Area is ")
      (* (* base height) half)
    )
    0.5 )
)
```

```
(tri-area 5 4)
```

```
(define tri-area
  (lambda (base height)
    (let ((half 0.5))
      (display "Area is ")
      (* (* base height) half)
    )
  )
)
```

```
Area is 10.0
```

### Ques 4.3

- a) Define the function **tri-area** that returns the area of a triangle when passed the arguments **base** and **height**. Use **let** to establish the constant **one-half**.

```
(define tri-area  
  (lambda (base height)  
    (let ((half 0.5))  
      (display "Area is ")  
      (* (* base height) half)  
    )  
  )  
)
```

```
(tri-area 5 4)
```

```
(define (tri-area base height)  
  (let ((half 0.5))  
    (display "Area is ")  
    (* (* base height) half)  
  )  
)
```

```
Area is 10.0
```

b) The following function named **add-two** is meant to read two numbers from the screen and print the numbers and their sum. Debug the program, keeping the constants in the **let** block.

```
(define (add-two)
  (let ((first ((display "Enter first number: ")
                 (read)))
        (second ((display "Enter second number: ")
                 (read))))
    (sum (+ first second))
  )
  (display "Total is " sum)
)
```

b) The following function named **add-two** is meant to read two numbers from the screen and print the numbers and their sum. Debug the program, keeping the constants in the **let** block.

```
(define (add-two)
  (let ((first ((display "Enter first number: ")
                 (read)))
        (second ((display "Enter second number: ")
                 (read))))
    (sum (+ first second))
  )
  (display "Total is " sum)
)
```

b) The following function named **add-two** is meant to read two numbers from the screen and print the numbers and their sum. Debug the program, keeping the constants in the **let** block.

```
(define (add-two)
  (let ((first ((display "Enter first number: ")
                 (read)))
        (second ((display "Enter second number: ")
                 (read)))
        (sum (+ first second)))
    (display "Total is " sum)
  )
)
```

```
(display "Enter first number: ")
```

```
(let ((first (read)))
```

.....

```
(let ((sum (+ first second)))
```

```
(display ...)
```

b) The following function named **add-two** is meant to read two numbers from the screen and print the numbers and their sum. Debug the program, keeping the constants in the **let** block.

```
(define (add-two)
  (display "Enter first number: ")
  (let ((first (read)) )
    (display "Enter second number: ")
    (let ((second (read) ) )
      ( let ((sum (+ first second)) )
        (display "Total is" )
        (display sum)
      )
    )
  )
)
```

The image shows a screenshot of the DrScheme IDE. The window title is "Untitled - DrScheme\*". The menu bar includes "File", "Edit", "View", "Language", "Scheme", "Special", and "Help". The toolbar contains buttons for "Untitled", "Save", "Step", "Check Syntax", "Run", and "Stop". The main text area contains the following Scheme code:

```
(define (add-two)
  (display "Enter first number: ")
  (let ((first (read)))
    (display "Enter second number: ")
    (let
      ((second (read)))
      (let
        ((sum (+ first second)))
        (display "total is ")
        (display sum)
        )
      )
    )
  )
)
(add-two)
```

The bottom panel shows the output of the program:

```
Welcome to DrScheme, version 208.
Language: Standard (R5RS).
Enter first number: 5
Enter second number: 9
total is 14
```

b) The following function named **add-two** is meant to read two numbers from the screen and print the numbers and their sum. Debug the program, keeping the constants in the **let** block.

```
(define (add-two)
  (display "Enter first number: ")
  (let ((first (read)) )
    (display "Enter second number: ")
    (let* ((second (read) )(sum (+ first second)) )
      (display "Total is" ) (display sum)
    )
  )
)
```

File Edit View Language Scheme Special Help

Untitled

Save

Step

Check Syntax

Run

Stop

(define ...)

```
(define (add-two)
  (display "Enter first number: ")
  (let ((first (read)))
    (display "Enter second number: ")
    (let*
      ((second (read)) (sum (+ first second)))
        (display "total is ")
        (display sum)
      )
    )
  )
(add-two)
```

```
Enter first number: 2
Enter second number: 3
total is 5
```

>

4:2

Read/Write

not running

Ques 4.4. What do the following functions do?

```
(i) (  
      do ((vec (make-vector 5))  
          (i 0 (+ i 1)))  
          ((= i 5) vec)  
          (vector-set! vec i i)  
    )
```

**do** *((variable init step) ...) (test expression)  
command ...*

1. **variable** is initialised by **init**.
2. Iteration begins by evaluating **test**.
3. If **test** is false, **command** and **step** will be evaluated. After which continue to Step 2.
4. If **test** is true, exit do loop and return value of **expression** as the value of do.

variable1: vec  
init1: (make-vector 5)  
Step 1: null (nothing)  
variable2: i  
init2: 0  
Step 2: (+ i 1)  
test: (= i 5)  
expression: vec  
command: (vector-set! vec i i)

```
( do ((vec (make-vector 5))  
      (i 0 (+ i 1)))  
      ((= i 5) vec)  
      (vector-set! vec i i)  
  )
```

**do** *((variable init step) ...)* *(test expression) command ...*

1. **variable** is initialised by **init**.
2. Iteration begins by evaluating **test**.
3. If **test** is false, **command** and **step** will be evaluated. After which continue to Step 2.
4. If **test** is true, exit do loop and return value of **expression** as the value of do.

```
vec= (make-vector 5); i = 0
```

```
Result is → (vector .. .. .)
```

```
((= i 5) ...) → false
```

```
(vector-set! vec 0 0)
```

```
Result is → (vector 0 .. .. .)
```

```
Step now updates i;
```

```
Result is → i (= i+1) = 1
```

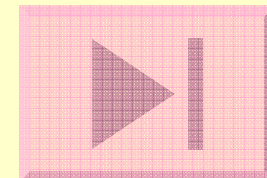
```
( do ((vec (make-vector 5))  
      (i 0 (+ i 1)))  
      ((= i 5) vec)  
      (vector-set! vec i i)  
  )
```

**do** ((*variable init step*) ...) (*test expression ...*) *command ...*

1. *variable* is initialised by **init**.
2. Iteration begins by evaluating *test*.
3. If *test* is false, *command* and **step** will be evaluated. After which continue to Step 2.
4. If *test* is true, exit do loop and return value of **expression** as the value of do.

**Result:** (vector 0 1 2 3 4)

The function basically assigns the *i* value into a vector *vec* by iterating the *i* value from 0 to 4.

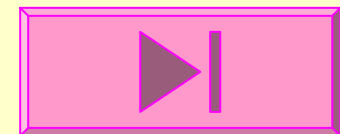


variable1 : x  
init1 : x = '(1 3 5 7 9) // different scope  
step1 : (cdr x)  
variable2 : sum  
Init2: sum = 0  
step2 : (+ sum (car x))  
test : (null? x)  
expression : sum  
command : nothing (null)

```
(ii) (let ((x '(1 3 5 7 9)))  
      (do ((x x (cdr x))  
          (sum 0 (+ sum (car x))))  
          ((null? x) sum)  
          )  
      )
```

**do** ((*variable init step*) ...) (*test expression ...*) *command* ...

1. **variable** is initialised by **init**.
2. Iteration begins by evaluating **test**.
3. If **test** is false, **command** and **step** will be evaluated. After which continue to Step 2.
4. If **test** is true, exit do loop and return value of **expression** as the value of do.



x = '(1 3 5 7 9); sum = 0

`((null? x) → false`

Step now updates x & sum;

`(x = cdr(x))`

Result is → `x = '(3 5 7 9);`

`(sum = sum + car(x))`

Result is → `sum (= sum + 1) = 1;`

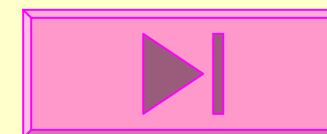
**Result: 25**

Initialize list x with values '(1 3 5 7 9) and for every iteration add the value of the 1st element to the current sum.

```
(ii) (let ((x '(1 3 5 7 9)))
      (do ((x x (cdr x))
          (sum 0 (+ sum (car x))))
          ((null? x) sum)
      )
    )
```

**do** *((variable init step) ...) (test expression ...) command ...*

1. **variable** is initialised by **init**.
2. Iteration begins by evaluating **test**.
3. If **test** is false, **command** and **step** will be evaluated. After which continue to Step 2.
4. If **test** is true, exit do loop and return value of **expression** as the value of do.



Finished tut 4 ...