Scheme

1. Answer:

; 1a

An anonymous function example:

((lambda (x) (+ x x))6)

Output:

12

; 1b

A named function example:

(define (square n) (\* n n))

#| apply |#

(square 6)

Output:

36

; 1c

A data structure with function example:

(define a (lambda (x) (+ x x)))

(define b (lambda (x) (\* x x)))

#| apply |#

(list (a 7)(b 8))

Output:

'(14 64)

; 1d

Equality test example:

#| list |#

(define list-evena (list 2 4 6))

(define list-evenb (list 2 4 6))

(define list-odd (list 1 3 5))

#| function |#

(define multi2 (lambda (x) (\* x 2)))

(define plus2 (lambda (x) (+ x 2)))

#| apply |#

(equal? list-evena list-evenb)

Output:

#t

(equal? list-evena list-odd)

Output:

#f

(equal? (multi2 2) (plus2 2))

Output:

#t

(eq? (multi2 2) (plus2 2))

Output:

#t

(eq? list-evena list-evenb)

Output:

#f

(eqv? (multi2 2)4)

Output:

#t

; 1e

Function as an argument example:

(define sum (lambda (x)

(if (null? x)

0

(+ (car x) (sum (cdr x))))))

(define length (lambda (x)

(if (null? x)

0

(+ 1 (length (cdr x))))))

(define mean (lambda (x)

(/ (sum x) (length x))))

#| apply |#

(mean (list 1 2 3))

Output:

2

; 1f

Function as a return value example:

(define (squar n) (\* n n))

#| apply |#

((compose squar +) 6 2)

Output:

64

; 1g

Interaction with files:

#| read from keyboard |#

(define formula (read))

Input:

(lambda (x) (\* x x))

#| read from file |#

(define file (open-input-file "formula.txt"))

(read file)

Output:

'(lambda (x) (- x x))

#| display formula |#

(display formula)

Output:

(lambda (x) (\* x x))

(close-input-port file)

1. Answer:

;square

(define square (lambda (n) (\* n n)))

;sum

(define (sum x)

(if (null? x)

0

(+ (car x) (sum (cdr x)))))

(define (sum-of-sqr x)

(if (null? x)

0

(+ (square (car x)) (sum-of-sqr (cdr x)))))

;length

(define (length x)

(if (null? x)

0

(+ 1 (length (cdr x)))))

;average

(define mean-sum

(lambda(x . y)

(/ (+ x (sum y)) (+ 1 (length y)))))

(define mean-sum-sqr (lambda (x . y)

(/ (+ (\* x x)(sum-of-sqr y)) (+ 1 (length y)))))

;std deviation

(define (sigma x . y)

(sqrt (- (apply mean-sum-sqr x y) (square (apply mean-sum x y)))))

#| apply |#

(sigma 1 2 3 4 5)

Output:

1.4142135623730951

(sigma 1 2 3 4 5 6 7)

Output:

2

(sigma 1 2 3)

Output:

0.816496580927726

(sigma 1 )

Output:

0

1. Answer:

;star line

(define (line x)

(cond

((zero? x) (newline))

(else (display "\*") (line (- x 1)))))

#| apply |#

(line 10)

Output:

\*\*\*\*\*\*\*\*\*\*

;histogram

(define (histogram x)

(map line x)

(display ""))

#| apply |#

(histogram '(9 10 5 6 2 1 15 3 8))

Output:

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*

\*\*

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*

\*\*\*\*\*\*\*\*

1. Answer:

#lang racket

;Display Answer

(define Display

(lambda (x i)

(display (/ (round (\* x (expt 10 i))) (expt 10 i)))))

;Trisection method

(define Max

(lambda (fun a b)

(cond

((< (- b a) 1e-10)

(display "Max: f(")

(Display (/ (+ a b) 2) 4)

(display ") = ")

(Display (fun (/(+ a b) 2)) 4))

(else

(let ((x1 (+ a (/ (- b a) 3)))

(x2 (- b (/ (- b a) 3))))

(if (< (fun x1) (fun x2))

(Max fun x1 b)

(Max fun a x2)))))))

#| apply |#

(Max sin (/ pi -6) (/ pi 6))

Output:

Max: f(0.5236) = 0.5

(Max (lambda (x) (+ x (/ 1 x))) 0.1 10)

Output:

Max: f(10.0) = 10.1

(Max (lambda (x) (\* x x)) -80 80)

Output:

Max: f(-80) = 6400

(Max sqrt 0 1)

Output:

Max: f(1) = 1.0

1. Answer:
2. Scalar product of two vectors with Do-Loop.

#lang racket

(define (scalar-product-iterative a b)

(cond

((equal? (vector-length a)(vector-length b))

(do ((i 0 (+ i 1)) (sum 0))

((>= i (vector-length a)) sum)

(set! sum (+ sum (\* (vector-ref a i) (vector-ref b i))))))

(else(display "ERROR: DIFFERENT SIZES OF VECTORS!"))))

#| apply |#

(scalar-product-iterative '#(1 2 3) '#(4 5 6))

Output:

32

(scalar-product-iterative '#(1 3 5) '#(2 4))

Output:

ERROR: DIFFERENT SIZES OF VECTORS!

1. Scalar product with Recursion.

(define sum 0)

(define (scalar-product-recursive a b)

(cond

((equal? (vector-length a)(vector-length b))

(cond

((zero? (vector-length a))

(display sum)

(set! sum 0))

(else

(set! sum (+ sum (\* (vector-ref a 0) (vector-ref b 0))))

(scalar-product-recursive (vector-drop a 1) (vector-drop b 1)))))

(else(display "ERROR: DIFFERENT SIZES OF VECTORS!"))))

#| apply |#

(scalar-product-recursive '#(1 2 3) '#(4 5 6))

Output:

32

(scalar-product-recursive '#(1 3 5) '#(2 4))

Output:

ERROR: DIFFERENT SIZES OF VECTORS!

1. Answer:
2. Read and Display Vector

#| Read & Display |#

;Read from a file

(define (read-file file)

(let\* ((infile (open-input-file file))

(r (read infile))

(c (read infile))

(matrix (make-vector r)))

(do ((i 0 (+ i 1)))

((>= i r) (close-input-port infile) matrix)

(let ((row (make-vector c)))

(do ((j 0 (+ j 1)))

((>= j c) (vector-set! matrix i row))

(vector-set! row j (read infile)))))))

;number of row

(define (num-row file pos)

(define matrix (read-file file))

(vector-ref matrix pos))

;number of col

(define (num-col file pos)

(define matrix (read-file file))

(define rw (vector-length matrix))

(define cl (make-vector rw))

(do ((i 0 (+ i 1)))

((>= i rw) cl)

(vector-set! cl i (vector-ref (vector-ref matrix i) pos))))

;Display vector

(define (display-vector v)

(do ((i 0 (+ i 1)))

((>= i (vector-length v)) (display ""))

(display (vector-ref v i)) (display " ")))

;Display row

(define (row file index)

(display-vector (num-row file index)))

;Display colum

(define (col file index)

(display-vector (num-col file index)))

#| apply |#

(read-file "matrix1.dat")

Ouput:

'#(#(1 2 3) #(4 5 6))

(read-file "matrix2.dat")

Ouput:

'#(#(1 2 3) #(1 2 3) #(1 2 3))

(row "matrix1.dat" 0)

Ouput:

1 2 3

(col "matrix1.dat" 0)

Ouput:

1 4

(row "matrix2.dat" 1)

Ouput:

1 2 3

(col "matrix2.dat" 1)

Ouput:

2 2 2

1. Matrix Multiplication

;scalar product of a b

(define (scalar-product v1 v2)

(cond

((equal? (vector-length v1) (vector-length v2))

(do ((i 0 (+ i 1)) (sum 0))

((>= i (vector-length v1)) sum)

(set! sum (+ sum (\* (vector-ref v1 i) (vector-ref v2 i))))))

(else (display "ERROR: Different sizes of vectors"))))

;multiply matrix

(define (mmul fileone filetwo outputfile)

(define mat1 (read-file fileone))

(define mat2 (read-file filetwo))

(define rw (vector-length mat1))

(define cl (vector-length mat2))

(define outfile (open-output-file outputfile))

(display rw outfile) (display " " outfile)

(display cl outfile) (newline outfile)

(do ((i 0 (+ i 1)))

((>= i rw) (close-output-port outfile) (display ""))

(let ((row1 (make-vector cl)))

(do ((j 0 (+ j 1)))

((>= j cl) (display-vector row1) (newline) (newline outfile) )

(vector-set! row1 j (scalar-product (num-row fileone i) (num-col filetwo j)))

(display (vector-ref row1 j) outfile) (display " " outfile)))))

#| apply |#

(mmul "matrix1.dat" "matrix2.dat" "matrix3.dat")

Ouput:

6 12 18

15 30 45

(read-file "matrix3.dat")

Ouput:

'#(#(6 12 18) #(15 30 45))