Shenliang Wang

10/15/2017

CSC 600

SECTION #1

HW#4

Scheme

1.The concept of first class objects is fundamental for Scheme programming. In particular, in Scheme functions are first class objects. The main properties of functions as first class objects are exemplified by answering the following questions:

**(a) The first class object may be expressed as an anonymous literal value (constant). Show an example of the anonymous function and its use.**

> 123

123

**(b) The first class object may be stored in variables (i.e. it may have a symbolic name). Show examples of defining and using named functions.**

> (define x 5)

> x

5

**(c) The first class object may be stored in data structures. Show an example of a data structure(e.g. a list) that contains functions**

> (length (list 2 4 6 8 10))

5

**(d) The first class object may be comparable to other objects for equality. Show an example of comparing functions and lists for equality.**

> (eq? ‘x ‘x)

#t

> (eqv? (\* 2 2) (sqrt 16))

#t

> (equal? 3 (/ 6 2))

#t

**(e) The first class object may be passed as parameter to procedures/functions. Show an example of passing function as an argument to another function.**

> ((lambda (x) (\* 2 x)) 5)

10

**(f) The first class object may be returned as result from procedures/functions. Show an example of returning a function as a result of another function.**

> (define fact2 (lambda (n) (if (= n 0) 1 (\* n (fact2 (- n 1))))))

> (fact2 3)

6

**(g) The first class object may be readable and printable. Show examples of reading function(s) from the keyboard, reading function(s) from a file, and displaying a function.**

> (define p (open-input-file "testdata.txt"))

> (read p)

123

>(display p)

thank\_you

2.

code:

#lang racket

**;Squares the number**

(define (squarenumber x)

(\* x x))

**;Add up all the number son the list**

(define (listsum lst)

(if (null? lst) 0

(+ (car lst) (listsum (cdr lst)))))

**;Square all the number in the list**

(define (squarelist lst)

(if (null? lst) 0

(+ (squarenumber (car lst)) (squarelist (cdr lst)))))

**;Initital function**

(define sigma (lambda x(sqrt (- (/ (squarelist x) (length x))

(squarenumber (/ (listsum x) (length x)))))))

3)

a) Write a recursive Scheme procedure **line** that prints n asterisks in a line as follows:

**code:**

#lang racket

(define (line n)

(cond ((< n 1) (newline))

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

(define (histogram list)

(if (null? list) (newline)

(begin (line (car list))

(histogram (cdr list)))))

**output:**

> (line 5)

\*\*\*\*\*

b) Write a recursive Scheme procedure **histogram** that uses the procedure **line** , and prints a histogram for a list of integers:

**code:**

#lang racket

(define (line n)

(cond ((< n 1) (newline))

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

(define (histogram list)

(if (null? list) (newline)

(begin (line (car list))

(histogram (cdr list)))))

**output:**

> (histogram '(1 2 3 3 2 1))

\*

\*\*

\*\*\*

\*\*\*

\*\*

\*

4)Write a Scheme program for computing a maximum of function f(x) within the interval [x1, x2]. Use the trisection method, and find the coordinate of maximum xmax with accuracy of 6 significant decimal digits.

#lang racket

(define (fmax X Y funct)

(let\* ((trisect (/ (- Y X) 3.))

(xtri (+ X trisect))

(ytri (- Y trisect)))

(cond [(< (abs (- (funct X) (funct Y))) .0000001)

(/ (+ Y X) 2.)]

[(> (funct xtri) (funct ytri))

(fmax X ytri funct)]

[else (fmax xtri Y funct)])))

5. (a) Write the program in iterative style using the DO loop.

**Code:**

#lang racket

(define scalar-product

(lambda (x y)

(cond((= (vector-length x) (vector-length y))

(let ((z 0))

(do ((i 0 (add1 i)))

((>= i (vector-length x)) (display z))

(set! z

(+ z (\* (vector-ref x i) (vector-ref y i))))))

(else (display "Error: different sizes of vectors!")))))

**output:**

> (scalar-product '#(1 2 3) '#(2 1 1))

7

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

Error: different sizes of vectors!

b) Write the program using recursion.

**code:**

#lang racket

(define scalar-product

(lambda (x y)

(cond ((= (vector-length x) (vector-length y))

(recur (vector->list x) (vector->list y)))

(else (display "Error: different sizes of vectors!")))))

(define recur

(lambda (list1 list2)

(cond ((null? (cdr list1)) (\* (car list1) (car list2)))

(else (+ (\* (car list1) (car list2))

(recur (cdr list1) (cdr list2)))))))

**ouput:**

> (scalar-product '#(1 2 3) '#(2 1 1))

7

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

Error: different sizes of vectors!