CSc 600-01 (Section 1)

Homework 4 - Functional Programming

prepared by Ilya Kopyl

CSc 600 Homework 4 - Scheme and Functional Programming

Ilya Kopyl

April 27, 2018

Homework is prepared in LaTeX with TeXShop editor (under GNU GPL).

1. The concept of first class objects is fundamental for Scheme programming. In particular, in Scheme language any function is a first class object. The main properties of a function as a first class object 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.

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.

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.

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

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.

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.

g) The first class object may be readable and printable. Show examples of:

- reading function(s) from keyboard,
- reading function(s) from a file,
- displaying a function.

The answer is listed on the pages 2 through 11.

1. (a) Show an example of the anonymous function and its use.

At the very basic case, an anonymous function is nothing more but a lambda expression - a body of a function. Since it is not associated with any identifier (i.e. when it is unnamed), we may use it only when we explicitly write it inside of other expressions:

```
> ((lambda(x) (* x x)) 2)
4
```

Also, an anonymous function can be used in cases when we otherwise would have to define an inner function (inside another function):

```
> (define (add-num-to-each-element-in-list num lst)
    (map (lambda (x) (+ x num)) lst))

> (add-num-to-each-element-in-list 10 '(1 2 3))
(11 12 13)
```

1. (b) Show examples of defining and using named functions.

Since we already know what a lambda function is, we can now take it and associate it with a name (identifier) - and then we could use just this name to evaluate any expression with it:

```
> (define square (lambda(x) (* x x)))

> square
#<procedure:square>

> (square 2)
4
> (square 4)
16
```

- thus the function definition is a process of binding a lambda expression to some identifier. For simplicity and convenience we could make the same function definition with the use of syntactic sugar and omit 'lambda' keyword. Semantically, such function definition would still remain the same:

```
> (define (square x) (* x x))

> (square 2)
4
> (square 4)
16
```

We could also take our defined function square and store it in a variable:

```
> (define a sqr)

> a
#<procedure:sqr>

> (a 2)
4
> (a 4)
16
```

The reason why we define functions is to follow the Single Responsibility Principle, when each function is doing only one thing. But at certain point we would need to do a function composition, i.e. to define a function that is composed of series of expressions that use previously defined functions:

```
> (define (sum-of-squares lst)
    (apply + (map square lst)))

> (sum-of-squares '(1 2 3 4))
30
```

1. (c) Show an example of a data structure (e.g. a list) that contains functions.

If we think of a function as an entity that is responsible for only one operation/modification, then chaining different functions to each other to perform sequential modification of the same data can be viewed as a sort of Henry Ford's conveyor belt. In the previous code we considered a case when the number, kinds of functions and their order of evaluation are known and determined, but a data is not. But let's consider a situation when we initially don't know a number, nor kinds of functions, nor their order of evaluation at all. We would need to figure out how to modify a given data dynamically, at runtime. The approach is to pass a list of functions as an extra argument to our master function. With the help of a tail recursion we could take one function at a time from that list, and evaluate it with our data, provided that the arity of these functions match with the number of data arguments. But I digressed. There are two data structures to store functions in: a list and a quoted list. Let's consider both approaches:

```
; functions we are going to work on
> (define (square x)
    (* x x))

> (define (double x)
    (* x 2))
```

```
; example of a quoted list: functions turned into symbols
> (quote (square double))
(square double)
> '(square double)        ; syntactic sugar equivalent
(square double)

> (symbol? (car '(square)))
#t

; example of a list of functions:
> (list square double)
(#<procedure:square> #<procedure:double>)

> (procedure? (car (list square)))
#t


; we can assign both lists to variables:
> (define lst-of-fun1 '(square double))

> (define lst-of-fun2 (list square double))


; differences in use:
> ((car (list square)) 5)
25

> ((eval (car '(square))) 5)
25
```

1. (d) Show an example of comparing functions and lists for equality.

To answer this question we must first understand what equality means when it comes to comparing functions or lists with each other. For any object to be equal to another object in Scheme it either needs to point to the same location in memory as the other one, or to have the same type and value, or to have the same, equal numerical value.

Per Revised Report on the Algorithmic Language Scheme (r7rs):

- The eqv? procedure defines a useful equivalence relation on objects. Briefly, it returns #t if obj1 and obj2 are pairs, vectors, bytevectors, records, or strings that denote the same location in the store; or if obj1 and obj2 are procedures whose location tags are equal.

- The equal? procedure, when applied to pairs, vectors, strings and bytevectors, recursively compares them, returning #t when the unfoldings of its arguments into (possibly infinite) trees are equal (in the sense of equal?) as ordered trees, and #f otherwise. It returns the same as eqv? when applied to booleans, symbols, numbers, characters, ports, procedures, and the empty list. If two objects are eqv?, they must be equal? as well. In all other cases, equal? may return either #t or #f.
- The eq? predicate is similar to eqv? except that in some cases it is capable of discerning distinctions finer than those detectable by eqv?. On symbols, booleans, the empty list, pairs, and records, and also on non-empty strings, vectors, and bytevectors, eq? and eqv? are guaranteed to have the same behavior. On procedures, eq? must return true if the arguments? location tags are equal. On numbers and characters, eq?'s behavior is implementation-dependent, but it will always return either true or false. On empty strings, empty vectors, and empty bytevectors, eq? may also behave differently from eqv?.

By mathematical definition of a function, the only valid way to compare two functions for their equivalence is to verify that for every valid input both functions produce the same output, and that both functions also have the same domain and range. This means that a function equality can only be defined correctly in terms of operational equivalency; that is, the implementation doesn't matter, only the behavior does. This, of course, is an undecidable problem in any nontrivial language. It is impossible to determine if any two functions are operationally equivalent because, if we could, we could solve the halting problem.

Let's consider a series of experiments to prove this point:

```
> (define (square x)
    (* x x))
> (eq? square square)
#t
> (equal? square square)
#t
> (eqv? square square)
#t


> (define a square)
> (eq? a square)
#t
> (equal? a square)
#t
> (eqv? a square)
#t
```

```
> (define (sqr x)
    (* x x))
> (eq? square sqr)
#f
> (equal? square sqr)
#f
> (eqv? square sqr)
#f
```

- as you can see, if two identifiers both point to the same object (a function) in memory, the result of comparison of these two identifiers with any of the predicates eq?, equal?, eqv? would result in true. However, there are no means to check the equivalence of functions located in two different locations in memory, even if they consist of the same sequence of expressions.

Examples of comparing lists for equality:

```
; base case: an empty list
> (equal? '() '())
#t
> (eq? '() '())
#t
> (eqv? '() '())
#t

; a list of 1 symbol
> (equal? '(a) '(a))
#t
> (eq? '(a) '(a))
#f
> (eqv? '(a) '(a))
#f

; same as:
> (equal? (list 'a) (list 'a))
#t
> (eq? (list 'a) (list 'a))
#f
> (eqv? (list 'a) (list 'a))
#f


> (equal? '(a (b) c) '(a (b) c))
#t
> (eq? '(a (b) c) '(a (b) c))
#f
> (eqv? '(a (b) c) '(a (b) c))
#f
```

```
> (define list1 '(1 2 3))
> (define list2 '(1 2 3))

> (equal? list1 list2)
#t
> (eq? list1 list2)
#f
> (eqv? list1 list2)
#f

> (define list11 list1)

> (equal? list11 list1)
#t
> (eq? list11 list1)
#t
> (eqv? list11 list1)
#t
```

1. (e) Show an example of passing function as an argument to another function.

Let's continue the discussion started in the problem (1.c) and define a function conveyor that takes as its arguments a list of numbers (as data that we want to modify), and either a single function or a list of functions (as a sequence of operations to perform on that data):

```
; initial setup:
(define (square x)
    (* x x))

(define (double x)
    (* x 2))

(define (atom? x)
  (and (not (pair? x))
       (not (null? x))))

(define (conveyor lst-functions lst-data-operand1)
    (cond
      [(empty? lst-functions) lst-data-operand1]
      [(atom? lst-functions) (map lst-functions lst-data-operand1)]
      [else (conveyor (cdr lst-functions)
                      (if (procedure? (car lst-functions))
                          (map (car lst-functions) lst-data-operand1)
                          (map (eval (car lst-functions)) lst-data-operand1)))]))
```

```
; passing a single function as an argument:
> (conveyor square '(1 2 3 4))
(1 4 9 16)

> (conveyor double '(1 2 3 4))
(2 4 6 8)

> (define lst-fun1 '(square double))
> lst-fun1
(square double)

> (define lst-fun2 (list square double))
> lst-fun2
(#<procedure:square> #<procedure:double>)

; passing a quoted list of functions (where functions are symbols) as an argument:
> (conveyor lst-fun1 '(1 2 3 4))
(2 8 18 32)

; passing a list of functions as an argument:
> (conveyor lst-fun2 '(1 2 3 4))
(2 8 18 32)

> (conveyor '(square double sqrt) '(1 2 3 4))
(1.4142135623730951 2.8284271247461903 4.242640687119285 5.656854249492381)
```

1. (f) Show an example of returning a function as a result of another function.

In order to return a function we must put a lambda expression as the last expression to evaluate inside another function. Let's consider the following code:

```
; auxiliary function
(define (atom? x)
  (and (not (pair? x))
       (not (null? x))))

; two hard things in Computer Science: cache invalidation and naming things
(define (multiple-fun x)
  (cond
    [(number? x) (lambda(d) (* d d))]
    [(procedure? x) (lambda(d) (* d d))]
    [(atom? x) (lambda(d) (display d))]
    [else (lambda(d) (car d))]))
```

```
; examples of use:
> (multiple-fun 123)
#<procedure:...hw4-problem1.rkt:51:17>

> ((multiple-fun 123) 32)
1024

> ((multiple-fun 'a) "Hello, world!")
Hello, world!

> ((multiple-fun '(a b c)) '(Apple Banana))
Apple

> ((multiple-fun (lambda(n) (+ n n))) 256)
65536

> ((multiple-fun +) 256)
65536

; leveraging the function conveyor defined in the previous problem (1e)
> (conveyor (list (multiple-fun +)) '(1 2 3 4))
(1 4 9 16)

; which is the same as:
> (map (multiple-fun +) '(1 2 3 4))
(1 4 9 16)
```

1. (g) Show examples of reading function(s) from the keyboard, reading function(s) from a file, and displaying a function.

Reading a function from keyboard:

```
> (eval (read))
(lambda(x) (+ 1 2))
#<procedure:...t/private/kw.rkt:446:14

> ((eval (read)) 256)
(lambda(x) (+ x x))
512

> (define (read-fun-from-keyboard)
    (eval (read)))

> ((read-fun-from-keyboard) 256)
(lambda(x) (* x 4))
1024
```

Reading functions from a file, displaying them as quoted lists, and evaluating:

```
> (define in (open-input-file "/Users/ilya.kopyl/Documents/functions.txt"))
> (read in)
'(lambda (x) (* x x))
> (read in)
'(lambda (x) (+ x x))
> (read in)
'(define (square x) (* x x))
> (read in)
'(define (double x) (* x 2))
> (read in)
#<eof>
> (close-input-port in)


> (define in (open-input-file "/Users/ilya.kopyl/Documents/functions.txt"))
> ((eval (read in)) 32)
1024
> ((eval (read in)) 8)
16

> square
. . square: undefined;
 cannot reference an identifier before its definition

> (eval (read in))

> square
#<procedure:square>

> (square 16)
256


> double
. . double: undefined;
 cannot reference an identifier before its definition

> (eval (read in))

> double
#<procedure:double>

> (double 2)
4

> (close-input-port in)
```

Displaying a source code of a function:

```
ilya.kopyl@ilyakopyl-ltm TIScheme(1) $ mit-scheme
MIT/GNU Scheme running under OS X

Copyright (C) 2014 Massachusetts Institute of Technology

1 ]=> (define (square x) (* x x))

;Value: square

1 ]=> (pp square)
(named-lambda (square x)
  (* x x))
;Unspecified return value

1 ]=> (define (flatten2 sequence)
        (if (not (list? sequence)) (list sequence)
                (apply append (map flatten2 sequence))))

;Value: flatten2

1 ]=> (pp flatten2)
(named-lambda (flatten2 sequence)
  (if (not (list? sequence))
      (list sequence)
      (apply append (map flatten2 sequence))))
;Unspecified return value
```

2. Write a Scheme function *sigma* that computes the standard deviation of any number of arguments.

The mean value of n numbers is $\overline{x} = (x_1 + x_2 + \ldots + x_n)/n$.

The mean value of n squares is $\overline{x^2} = (x_1^2 + x_2^2 + \ldots + x_n^2)/n$.

The standard deviation is defined as $\sigma = \sqrt{\overline{x^2} - (\overline{x})^2}$.

Write a Scheme function *sigma* that computes the standard deviation of any number of arguments, as in the following example:

```
> (sigma 1 2 3 2 1)
0.748331477354788
> (sigma 1 3 1 3 1 3)
1.
> (sigma 1 3)
1.
> (sigma 1)
0.
```

Source code of the program:

```
#lang racket

(define (square x)
  (* x x))

(define (square-lst x)
  (map square x))

(define (sum-lst x)
  (apply + x))

(define (mean-lst lst)
  (/ (sum-lst lst)
     (length lst)))

; at least 1 argument is required
(define (sigma x . y)
  (let ((data (cons x y)))        ; defined the local variable for convenience
    (sqrt (- (mean-lst (square-lst data))
             (square (mean-lst data))))))
```

The result of the program execution:

```
> (sigma 1 2 3 2 1)
0.7483314773547883
> (sigma 1 3 1 3 1 3)
1
> (sigma 1 3)
1
> (sigma 1)
0
> (sigma 100 94 92 88 87 87)
4.6785562825394
```

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

```
> (line 5)
*****
```

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

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

The answer is listed on the page 14.

Source code of the program:

```
#lang racket

; auxiliary predicate
(define (non-negative? x)
  (if (or (equal? x 0)
          (positive? x)) #t #f))

;auxiliary predicate
(define (valid-input? x)
  (if (and (number? x)
           (and (integer? x)
                (non-negative? x))) #t #f))

(define (line x)
  (if (not (valid-input? x)) (display "Argument must be a non-negative integer.\n")
      (if (equal? x 0) (newline)              ; base case - print newline character
          (begin (display "*")                ; print *
                 (line (- x 1))))))           ; call line function for x-1


(define (histogram lst)
  (if (not (list? lst)) (display "Argument must be a list.\n")
      (if (empty? lst) (display "")           ; base case - print an empty string
          (begin (line (car lst))             ; call line function for car of the lst
                 (histogram (cdr lst))))))    ; call histogram for cdr of the lst
```

The result of the program execution:

```
> (line -1)
Argument must be a non-negative integer.
> (line pi)
Argument must be a non-negative integer.
> (line 0)

> (line 1)
*
> (line 5)
*****
> (histogram 12)
Argument must be a list.
> (histogram '(1 2 3 3 2 1))
*
**
***
***
**
*
```

4. Write a Scheme program for computing a maximum of function $f(x)$ within the interval $[x_1, x_2]$. Use the trisection method, and find the coordinate of maximum $xmax$ with accuracy of 6 significant decimal digits.

Source code of the program:

```
#lang racket

; auxiliary predicate
(define (difference-sufficiently-small? x1 x2)
  (let ((threshold 1e-10))    ; threshold can be easily changed in the future
    (if (< (abs (- x2 x1)) threshold) #t #f)))

; auxiliary function
(define (round-to-n-significant-decimal-digits x n)
  (/ (round (* x (expt 10 n)))
     (expt 10 n)))

; auxiliary function
(define (mean x . y)
  (/ (apply + (cons x y))
     (+ 1 (length y))))

; auxiliary function
(define (third-of-delta-between x1 x2)
  (/ (- x2 x1) 3))

; main function
(define (fmax f x1 x2)
  (if (not (procedure? f)) (display "First argument must be a procedure.")
      (cond
        [(difference-sufficiently-small? x1 x2)
         (let ((rounded-xmax (round-to-n-significant-decimal-digits (mean x1 x2) 6)))
           (display "xmax = ")
           (display rounded-xmax)
           (newline)
           (display "ymax = ")
           (display (round-to-n-significant-decimal-digits (f rounded-xmax) 6)))]
        [else (let ((a1 (+ x1 (third-of-delta-between x1 x2)))
                    (a2 (- x2 (third-of-delta-between x1 x2))))
                (if (< (f a1) (f a2))
                    (fmax f a1 x2)
                    (fmax f x1 a2)))])))
```

Results of the program execution:

```
> (difference-sufficiently-small? 1.00000000001 1.00000000008)
#t
> (difference-sufficiently-small? 1.00000000011 1.0000000008)
#f

> (fmax (lambda(x) (* x (- 1 x))) 0 10)
xmax = 1/2
ymax = 1/4
> (fmax (lambda(x) (* x (- x 1))) 0 10)
xmax = 10
ymax = 90
> (fmax (lambda(x) (* x (- x 1))) 0 9.95)
xmax = 9.95
ymax = 89.0525
> (fmax (lambda(x) (* x (- x 1))) 0 9.9875)
xmax = 9.9875
ymax = 89.762656
> (fmax (lambda(x) (* x (- x 1))) 0 9.987654)
xmax = 9.987654
ymax = 89.765578
> (fmax (lambda(x) (* x (- x 1))) 0 9.9876543)
xmax = 9.987654
ymax = 89.765578
> (fmax (lambda(x) (* x (- x 1))) 0 9.987654321)
xmax = 9.987654
ymax = 89.765578
```

5. Develop a program that computes the scalar product of two vectors. The program must not accept vectors having different size (in such a case print an error message). For example:

```
> (scalar-product '#(1 2 3) '#(2 1 1))
7
> (scalar-product '#(1 2 3) '#(1 2 3 4 5))
ERROR: Different sizes of vectors!
```

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

(b) **Write the program using recursion.**

The answer is listed on the pages TBD through TBD.

Source code of the program:

```
#lang racket

; the inner product of two vectors:
; A * B = (a1 * b1 + a2 * b2 + ... + an * bn)

; auxiliary predicate
(define (both-vectors? v1 v2)
  (and (vector? v1) (vector? v2)))

; auxiliary predicate
(define (vector-lengths-equal? v1 v2)
  (equal? (vector-length v1) (vector-length v2)))

; auxiliary predicate
(define (valid-vectors-input? v1 v2)
  (cond
    [(not (both-vectors? v1 v2))
     (begin (display "Error: Both arguments must be vectors.\n") #f)]
    [(not (vector-lengths-equal? v1 v2))
     (begin (display "Error: Both vectors must have the same length.\n") #f)]
    [else #t]))

; main program - entry point
(define (inner-product v1 v2 is-recursive)
  (cond
    [(not (valid-vectors-input? v1 v2)) (display "")]
    [else (if (equal? is-recursive #t)
              (inner-product-recursive (vector->list v1) (vector->list v2))
              (inner-product-iterative v1 v2))]))

; recursive implementation of inner-product
(define (inner-product-recursive lst1 lst2)
  (cond
    [(empty? lst1) 0]
    [(+ (* (car lst1) (car lst2))
        (inner-product-recursive (cdr lst1) (cdr lst2)))]))

; iterative implementation of inner-product
(define (inner-product-iterative v1 v2)
  (let ((sum 0))
    (do ((i 0 (add1 i)))
      ((>= i (vector-length v1)) sum)
      (set! sum (+ sum (* (vector-ref v1 i) (vector-ref v2 i)))))))
```

Results of the program execution:

```
> (inner-product #(1 2 3) 123 #f)
Error: Both arguments must be vectors.

> (inner-product #(1 2 3) 123 #t)
Error: Both arguments must be vectors.

> (inner-product #(1 2 3) #(1 2 3 4) #f)
Error: Both vectors must have the same length.

> (inner-product #(1 2 3) #(1 2 3 4) #t)
Error: Both vectors must have the same length.

> (inner-product #(1 2 3) #(2 1 1) #f)
7

> (inner-product #(1 2 3) #(2 1 1) #t)
7

> (inner-product #(1 2 3) #(3 2 1) #t)
10

> (inner-product #(1 2 3) #(3 2 1) #f)
10
```