

CSC 600-01 (SECTION 1)  
**Homework 5 - Introduction to Ruby**  
*prepared by Ilya Kopyl*

# CSC 600 HOMEWORK 4 - RUBY

## INTRODUCTION

Ilya Kopyl

May 2, 2018

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

**1. Function  $\text{rand}(n+1)$  returns a random integer between 0 and  $n$ . Write a function that creates an array of 100 random numbers between 0 and 10.**

Source code of the program:

```
def generate_new_rand_int_arr(n=100, upper_bound=10)
  n.times.map { rand(upper_bound + 1) }
end

p generate_new_rand_int_arr
```

Result of the program execution:

```
[9, 10, 10, 5, 5, 4, 0, 9, 8, 5, 7, 8, 7, 4, 2, 2, 4, 2, 8,
4, 6, 6, 5, 4, 7, 5, 0, 10, 1, 1, 4, 5, 4, 10, 2, 0, 1, 9, 4,
6, 1, 9, 3, 4, 0, 5, 7, 0, 4, 2, 7, 1, 4, 3, 3, 5, 0, 3, 1,
5, 1, 0, 7, 3, 4, 2, 3, 7, 1, 6, 5, 8, 5, 2, 7, 6, 9, 0, 7,
3, 10, 9, 4, 8, 0, 10, 5, 7, 7, 5, 6, 4, 1, 7, 5, 10, 1, 9,
7, 6]
```

**2. Make a function *show(v)* that displays the array *v*.**

Source code of the program:

```
puts "Hello, world!"
```

The result of the program execution:

```
Hello, world!
```

**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  $x_{max}$  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? 1.000000000001 1.000000000008)
#t
> (difference-sufficiently-small? 1.000000000011 1.000000000008)
#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 17 through 18.

Source code of the program:

```
#lang racket

; the inner product of two vectors:
;  $A * B = (a_1 * b_1 + a_2 * b_2 + \dots + a_n * b_n)$ 

; 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
```



**6. The files "matrix1.dat" and "matrix2.dat" are created using a text editor and contain two rectangular matrices. For example,**

matrix1.dat:

```
2  3
1  2  3
4  5  6
```

matrix2.dat:

```
3  3
1  2  3
1  2  3
1  2  3
```

In both cases the first row contains the size of the matrix (the number of rows and the number of columns). The remaining rows contain the values of elements.

- (a) Develop programs **row** and **col** that read a matrix from a file and display a specified row or column. For example:

```
> (row "matrix1.dat" 2)
4 5 6
> (col "matrix1.dat" 2)
2 5
```

Matrices should be stored in memory as vectors whose components are vectors.

- (b) Develop a program for matrix multiplication **mmul** that multiplies two matrices stored in specified input files, and creates and displays an output file containing the product. For example:

```
> (mmul "matrix1.dat" "matrix2.dat" "matrix3.dat")
6 12 18
15 30 45
```

In this example the contents of the new file "matrix3.dat" should be

```
2  3
6  12  18
15 30  45
```

The answer is listed on the pages 20 through 22.

Source code of the program:

```
#lang racket

; display components of the vector
(define (display-vector v)
  (do ((i 0 (add1 i)))
      ((>= i (vector-length v))
       (display " ")
       (display (vector-ref v i))
       (display " "))))

; read matrix from file
(define (read-matrix filename)
  (let* ((inport (open-input-file filename))
         (nrow (read inport))
         (ncol (read inport))
         (mat (make-vector nrow)))
    (do ((i 0 (add1 i)))
        ((>= i nrow) (close-input-port inport) mat)
      (let ((row (make-vector ncol)))
        (do ((j 0 (add1 j)))
            ((>= j ncol)
             (vector-set! mat i row)
             (vector-set! row j (read inport)))))))

; return i-th row of the matrix in filename
(define (ro filename i)
  (define mat (read-matrix filename))
  (vector-ref mat i))

; display i-th row of the matrix in filename
(define (row filename i)
  (display-vector (ro filename i)))

; return j-th col of the matrix in the filename
(define (co filename j)
  (define mat (read-matrix filename))
  (define nrow (vector-length mat))
  (define column (make-vector nrow))
  (do ((i 0 (add1 i)))
      ((>= i nrow) column)
    (vector-set! column i (vector-ref (vector-ref mat i) j)))

; display j-th col of the matrix in filename
(define (col filename j)
  (display-vector (co filename j)))
```

```

; 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))))))

; matrix multiplication, display and create
(define (mmul f1 f2 f3)
  (define m1 (read-matrix f1))
  (define m2 (read-matrix f2))
  (define nrow (vector-length m1))
  (define ncol (vector-length m2))
  (define outputport (open-output-file f3))
  (display nrow outputport)
  (display " " outputport)
  (display ncol outputport)
  (newline outputport)
  (do ((i 0 (add1 i)))
      ((>= i nrow) (close-output-port outputport) (display ""))
    (begin (let ((row (make-vector ncol)))
      (do ((j 0 (add1 j)))
          ((>= j ncol) (display-vector row) (newline) (newline outputport))
            (vector-set! row j (inner-product-iterative (row f1 i) (col f2 j)))
              (display (vector-ref row j) outputport)
                (display " " outputport)))))))

```

Results of the program execution:

```

> (read-matrix "mymatrix1.dat")
'#(#(1 9 5) #(2 5 6) #(4 3 1))

> (read-matrix "mymatrix2.dat")
'#(#(1 4 5) #(1 8 9))

> (row "mymatrix1.dat" 0)
1 4 5
> (row "mymatrix1.dat" 0)
1 4 5
> (row "mymatrix1.dat" 1)
1 8 9

> (col "mymatrix1.dat" 0)
1 1
> (col "mymatrix1.dat" 1)
4 8

> (col "mymatrix1.dat" 2)
5 9

```

```

> (mmul "mymatrix1.dat" "mymatrix2.dat" "mymatrix3.dat")
29 44 34
53 76 62

> (read-matrix "mymatrix3.dat")
'#(#(29 44 34) #(53 76 62))

> (read-matrix "matrix1.dat")
'#(#(1 2 3) #(4 5 6))

> (read-matrix "matrix2.dat")
'#(#(1 2 3) #(1 2 3) #(1 2 3))

> (mmul "matrix1.dat" "matrix2.dat" "matrix3.dat")
6 12 18
15 30 45

> (read-matrix "matrix3.dat")
'#(#(6 12 18) #(15 30 45))

```