TC2006 Programming Languages

Assignment 3: Basic Programming in Racket

Team: Sebastián Saldaña A01570274, Ana Elisa Estrada A01251091 and Estefanía Charles A01283472

In this assignment you will start practicing with the DrRacket language. Only one of the team members will be responsible for uploading the source code of the solutions to Canvas. The functions must be documented using Racket comments. Do not forget to add the team members identification information also as comments in the source file. **Observation**: displaying is different from returning when we are referring to the results of a function call.

1. Program the **smallest-pair** function that receives five non-negative integers and displays the two smallest values separated by a hyphen. The values can be repeated. Important: DO NOT USE LISTS for solving the problem, otherwise it will not be graded. Try with:

   ```
   > (smallest-pair 5 8 1 4 3)      => 1-3
   > (smallest-pair 1 0 1 0 1)      => 0-0
   > (smallest-pair 4 9 8 2 6)      => 2-4
   ```

```
(define (smallest-pair a b c d e)
  (cond ((and (< a c) (< a d) (< a e) (< b c) (< b d) (< b e) (if (<= a b) #t #f)) (printf "~a-~a" a b))
        ((and (< a c) (< a d) (< a e) (< b c) (< b d) (< b e)) (printf "~a-~a" b a))
        ((and (< a b) (< a d) (< a e) (< c b) (< c d) (< c e) (if (<= a c) #t #f)) (printf "~a-~a" a c))
        ((and (< a b) (< a d) (< a e) (< c b) (< c d) (< c e)) (printf "~a-~a" c a))
        ((and (< a b) (< a c) (< a e) (< d b) (< d c) (< d e) (if (<= a d) #t #f)) (printf "~a-~a" a d))
        ((and (< a b) (< a c) (< a e) (< d b) (< d c) (< d e)) (printf "~a-~a" d a))
        ((and (< a b) (< a c) (< a d) (< e b) (< e c) (< e d) (if (<= a e) #t #f)) (printf "~a-~a" a e))
        ((and (< a b) (< a c) (< a d) (< e b) (< e c) (< e d)) (printf "~a-~a" e a))
        ((and (< b a) (< b d) (< b e) (< c a) (< c d) (< c e) (if (<= b c) #t #f)) (printf "~a-~a" b c))
        ((and (< b a) (< b d) (< b e) (< c a) (< c d) (< c e)) (printf "~a-~a" c b))
        ((and (< b a) (< b c) (< b e) (< d a) (< d c) (< d e) (if (<= b d) #t #f)) (printf "~a-~a" b d))
        ((and (< b a) (< b c) (< b e) (< d a) (< d c) (< d e)) (printf "~a-~a" d b))
        ((and (< b a) (< b c) (< b d) (< e a) (< e c) (< e d) (if (<= b e) #t #f)) (printf "~a-~a" b e))
        ((and (< b a) (< b c) (< b d) (< e a) (< e c) (< e d)) (printf "~a-~a" e b))
        ((and (< c a) (< c b) (< c e) (< d a) (< d b) (< d e) (if (<= c d) #t #f)) (printf "~a-~a" c d))
        ((and (< c a) (< c b) (< c e) (< d a) (< d b) (< d e)) (printf "~a-~a" d c))
        ((and (< c a) (< c b) (< c d) (< e a) (< e b) (< e d) (if (<= c e) #t #f)) (printf "~a-~a" c e))
        ((and (< c a) (< c b) (< c d) (< e a) (< e b) (< e d)) (printf "~a-~a" e c))
        ((and (< d a) (< d b) (< d c) (< e a) (< e b) (< e c) (if (<= d e) #t #f)) (printf "~a-~a" d e))
        (else (printf "~a-~a" e d)))
```

2. Program the recursive function **logarithm** that returns the value of the logarithm of **y=1+x** by calculating n terms of the following series:

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots + (-1)^{n-1}\frac{x^n}{n}, \qquad -1 < x \leq 1$$

   Try with:

   x = y - 1

   ```
   > (logarithm 1.3 1)     => 0.3
   > (logarithm 0.5 4)     => -0.6822916
   ```

```
(define (logarithm y n)
   (if (= n 1)
       (sub1 y)
       (
           +
           (logarithm y (sub1 n))
           (* (expt (sub1 0) (sub1 n)) (/ (expt (sub1 y) n) n))
       )
   )
)
```

3. Program the recursive function sequences that displays n alternating sequences of the integers 1 through m first in ascending and then descending order.
   Try with:

```
> (sequences 4 6)
=> 1 2 3 4 5 6
   6 5 4 3 2 1
   1 2 3 4 5 6
   6 5 4 3 2 1
```

```
(define (sequences n m)
  (cond ((= n 1) (asc m))
        ((even? n) (sequences (- n 1) m) (newline) (des m))
        (else (sequences (- n 1) m) (newline) (asc m))))

(define (des n)
   (cond ( (= n 1)
      (display 1))
      (else (display n) (display " ") (des (- n 1)) )))

(define (asc n)
   (cond ( (= n 1)
      (display 1))
      (else (asc (- n 1)) (display " ")(display n))))
```

4. Program the recursive function **repeat** which, given a list of nonnegative integers, returns a list where each value is repeated the number of times it represents.
   Try with:

```
> (repeat '(0 1 2 3))        => (1 2 2 3 3 3)
> (repeat '(4 0 3 0 2))      => (4 4 4 4 3 3 3 2 2)
```

```
(define (repeat lst)
  (if(null? lst)
     '()
     (append (repeat-n (car lst))
             (repeat (cdr lst))
  )))
```

```
(define (repeat-n n)
  (repeatAux n n))

(define (repeatAux num cont)
  (if (zero? cont)
      '()
      (cons num (repeatAux num(- cont 1)))))
```

5. Program the recursive function **counters** that from a list of positions, given as positive integers, create a list of counters where each value in the list represents the number of times that position appears in the list of positions. The largest position will determine the size of the counters list.
   Try with:

```
> (counters '(1 2 3 4 5 6)) => (1 1 1 1 1 1)
> (counters '(2 2 4 2 4 2)) => (0 4 0 2)
> (counters '(6 8 3 6 6 1)) => (1 0 1 0 0 3 0 1)
```

```
(define (counters lst)
  (if (null? lst)
      '()
      (intPos (car lst)(counters (cdr lst))
  )))

(define (intPos pos cnts)
  (cond ((and (= pos 1) (null? cnts)) '(1))
        ((and (> pos 1) (null? cnts)) (cons 0 (intPos (- pos 1) '())))
        ((= pos 1) (cons (+ 1 (car cnts)) (cdr cnts)))
        (else (cons (car cnts) (intPos (- pos 1) (cdr cnts))))
))
```

6. In Racket, a variable number of arguments can be handled by separating the last argument (variable parameter) with a period, for example (define (procedure . args) …), where args will result in a list containing all the given arguments.
Program the **integers** recursive function that returns the total number of integers found in an arbitrary sequence of flat lists (lists without sublists) containing integers and symbols.
Try with:

```
> (integers '(1 2 3 4 5 6))                => 6
> (integers '(a b) '(c d))                 => 0
> (integers '(1 a) '(2 3) '(b c) '(9 d))   => 4
```

```
(define (integers . lst)
  (if (empty? lst)
     0
     (enteros (append* lst))
   )
)

(define (enteros lst)
  (if (empty? lst)
    0
    (if (integer? (car lst))
        (+ 1 (enteros (cdr lst)))
        (+ 0 (enteros (cdr lst)))))
)
```

7. Program the recursive function **prime-factors** which, receives an integer n as input (n > 0), and returns a list containing the prime factors of n in ascending order. Prime factors are the prime numbers that divide a number exactly. If all the prime factors are multiplied, the original number is obtained.
Try with:

```
> (prime-factors 1)          => ()
> (prime-factors 6)          => (2 3)
> (prime-factors 96)         => (2 2 2 2 2 3)
> (prime-factors 97)         => (97)
> (prime-factors 666)        => (2 3 3 37)
```

```
(define (prime-factors n)
   (if (= n 1)
       '()
       (cons (divisor n 2) (prime-factors (quotient n (divisor n 2))))
   )
)


(define (divisor n m)
   (if (= (remainder n m) 0)
       m
```

```
        (divisor n (+ m 1))
    )
)
```

8. Program the recursive function **shape** that, given a flat list and two positive integers N and M, returns a list with n sublists containing m elements each. If the flat list does not contain NxM elements, the missing elements should appear as hyphens (-) and if the flat list has more than NxM elements, the remaining elements should be discarded. Try with:

```
> (shape '(1 2 3 4 5 6) 2 3)      => ((1 2 3)(4 5 6))
> (shape '(1 2 3 4 5) 4 2)        => ((1 2)(3 4)(5 -)(- -))
> (shape '(1 2 3 4 5 6 7) 1 4)    => ((1 2 3 4))
```

```
; Problem 8
(define (shape lst n m)
    (cond
        ((= n 0) '() )
        ((null? lst) (cons (create-sublist lst m) '()))
        (else (cons (create-sublist lst m) (shape (slice-list lst m) (sub1 n) m)))
    )
)

(define (slice-list lst m)
    (cond
        ((null? lst) '())
        ((= m 1) (cdr lst))
        (else (slice-list (cdr lst) (sub1 m)))
    )
)

(define (make-hyphens m)
    (if (= m 0)
        '()
        (cons '- (make-hyphens (sub1 m)))
    )
)

(define (create-sublist lst m)
    (if (= m 0)
        '()
        (if (null? lst)
            (append (make-hyphens m) '())
            (cons (car lst) (create-sublist (cdr lst) (sub1 m)))
        )
```

```
    )
)
```

9. Program the recursive function **enumerate** that, given a possibly nested list, returns a list with the same form, but instead of each original atom returns a pair indicating its depth and its position in the (sub)list where it is found.
   Try with:

   `> (enumerate '(a b c))` **=> ((1 1) (1 2) (1 3))**

   > `(enumerate '(3 (b (c 2 (d)) a) 1))`

   => ((1 1) ((2 1) ((3 1) (3 2) ((4 1))) (2 3)) (1 3))

```
(define (enumerate lst)
  (enum-aux lst 1 1)
)

(define (enum-aux lst nivel pos)
  (cond ((null? lst)'())
        ((not (list? (car lst))) (cons (list nivel pos) (enum-aux (cdr lst) nivel (+ 1
pos))))
        (else (cons (enum-aux (car lst) (+ 1 nivel) 1) (enum-aux (cdr lst) nivel (+ 1
pos))))
        )
  )
```