Taller 1: Definición recursiva de programas e inducción (Racket)

Fundamentos de Lenguajes de Programación / 750095M / Grupo 01 / Prof. Robinson Duque / Monitor Juan Marcos Caicedo / 2019-1

1. Elabore una función llamada *copy* que reciba dos argumentos: un número **n** y una entrada **x**. La función debe retornar una lista con **n** ocurrencias de **x**. *Ejemplos:*

```
> (copy 7 'seven)
(seven seven seven seven seven seven seven)
> (copy 4 (list 1 2 3))
((1 2 3) (1 2 3) (1 2 3) (1 2 3))
> (copy 0 (list 5 6 7))
()
```

2. Elabore una función llamada *list-tails* que recibe como argumento una lista **L**, y lo que debe realizar dicha función es retornar en una lista todas las sublistas de los elementos consecutivos de la lista **L**. *Ejemplos:*

```
> (list-tails '(1 2 3 4 5))
((1 2 3 4 5) (2 3 4 5) (3 4 5) (4 5) (5))
> (list-tails '(1 a (e 4) 5 v))
((1 a (e 4) 5 v) (a (e 4) 5 v) ((e 4) 5 v) (5 v) (v))
```

3. Elabore una función llamada *list-set* que reciba tres argumentos: una lista **L**, un número **n** y un elemento **x**. La función debe retornar una lista similar a la que recibe (**L**), pero debe de tener en la posición ingresada **n** (indexando desde cero) el elemento **x**. *Ejemplos:*

```
> (list-set '(a b c d) 2 '(1 2))
(a b (1 2) d)
> (list-set '(a b c d) 3 '(1 5 10))
(a b c (1 5 10))
```

4. Elabore una función llamada *exists?* que debe recibir dos argumentos: un predicado **P** y una lista **L**. La función retorna #t si algún elemento de la lista **L** satisface el predicado **P**. Devuelve #f en caso contrario. *Ejemplos:*

```
> (exists? number? '(a b c 3 e))
#t
> (exists? number? '(a b c d e))
#f
```

5. Elabore una función llamada *list-index* que debe recibir dos argumentos: un predicado **P** y una lista **L**. La función retorna (desde una posición inicial 0) el primer elemento de la lista que satisface el predicado **L**. Si llega a suceder que ningún elemento satisface el predicado recibido, la función debe retornar **#f**. *Ejemplos:*

```
> (list-index number? '(a 2 (1 3) b 7))
1
> (list-index symbol? '(a (b c) 17 foo))
0
> (list-index symbol? '(1 2 (a b) 3))
#f
```

6. Elabore una función llamada *list-facts* que recibe como argumento un número entero **n**, y retorna una lista incremental de factoriales, comenzando desde 1! hasta $n!$. *Ejemplos:*

```
> (list-facts 5)
(1 2 6 24 120)
> (list-facts 8)
(1 2 6 24 120 720 5040 40320)
```

7. Elabore una función llamada *cartesian-product* que recibe como argumentos 2 listas de símbolos sin repeticiones **L1** y **L2**. La función debe retornar una lista de tuplas que representen el producto cartesiano entre **L1** y **L2**. Los pares pueden aparecer en cualquier orden. *Ejemplos:*

```
> (cartesian-product '(a b c) '(x y))
((a x) (a y) (b x) (b y) (c x) (c y))
> (cartesian-product '(p q r) '(5 6 7))
((p 5) (p 6) (p 7) (q 5) (q 6) (q 7) (r 5) (r 6) (r 7))
```

8. Elabore una función llamada *mapping* que debe recibir como entrada 3 argumentos: una función unaria (que recibe un argumento) llamada **F**, y dos listas de números **L1** y **L2**. La función debe retornar una lista de pares **(a,b)** siendo **a** elemento de **L1** y **b** elemento de **L2**, cumpliéndose la propiedad que al aplicar la función unaria **F** con el argumento **a**, debe arrojar el número **b**. Es decir, se debe cumplir que $\mathbf{F(a) = b}$. (Las listas deben ser de igual tamaño). *Ejemplos:*

```
>(mapping (lambda (d) (* d 2)) (list 1 2 3) (list 2 4 6))
((1 2) (2 4) (3 6))
> (mapping (lambda (d) (* d 3)) (list 1 2 2) (list 2 4 6))
((2 6))
> (mapping (lambda (d) (* d 2)) (list 1 2 3) (list 3 9 12))
()
```

9. Elabore una función llamada *inversions* que recibe como entrada una lista **L**, y determina el número de inversiones de la lista **L**. De manera formal, sea $A = (a_1 a_2 ... a_n)$ una lista de n números diferentes, si $i < j$ (posición) y $a_i > a_j$ (dato en la posición) entonces la pareja $(i\ j)$ es una inversión de A. *Ejemplos:*

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

10. Elabore una función llamada *up* que recibe como entrada una lista **L**, y lo que debe realizar la función es remover un par de paréntesis a cada elemento del nivel más alto de la lista. Si un elemento de este nivel no es una lista (no tiene paréntesis), este elemento es incluido en la salida resultante sin modificación alguna. *Ejemplos:*

```
> (up '((1 2) (3 4)))
(1 2 3 4)
> (up '((x (y)) z))
(x (y) z)
```

11. Elabore una función llamada *merge* que recibe como entrada dos listas de enteros ordenadas ascendentemente **L1** y **L2**. El procedimiento *merge* retorna una lista ordenada de todos los elementos de las listas **L1** y **L2**. *Ejemplos:*

```
> (merge '(1 4) '(1 2 8))
(1 1 2 4 8)
> (merge '(35 62 81 90 91) '(3 83 85 90))
(3 35 62 81 83 85 90 90 91)
```

12. Elabore una función llamada *zip* que recibe como entrada tres parámetros: una función binaria (función que espera recibir dos argumentos) **F**, y dos listas **L1** y **L2**, ambas de igual tamaño. El procedimiento *zip* debe retornar una lista donde la posición n-ésima corresponde al resultado de aplicar la función **F** sobre los elementos en la posición n-ésima en **L1** y **L2**. *Ejemplos:*

```
> (zip + '(1 4) '(6 2))
(7 6)
> (zip * '(11 5 6) '(10 9 8))
(110 45 48)
```

13. Elabore una función llamada *filter-acum* que recibe como entrada 5 parámetros: dos números **a** y **b**, una función binaria **F**, un valor inicial **acum** y una función unaria **filter**. El procedimiento *filter-acum* aplicará la función binaria **F** a todos los elementos que están en el intervalo $[a, b]$ y que a su vez todos estos elementos cumplen con el predicado de la función **filter**, el resultado se debe ir conservando en **acum** y debe retornarse el valor final de **acum**. *Ejemplos:*

```
> (filter-acum 1 10 + 0 odd?)
25
> (filter-acum 1 10 + 0 even?)
30
```

14. Elabore una función llamada *sort* que recibe como entrada dos argumentos: una lista de elementos **L** y una función de comparación **F**. La función debe retornar la lista **L** ordenada aplicando la función de comparación **F**. *Ejemplos:*

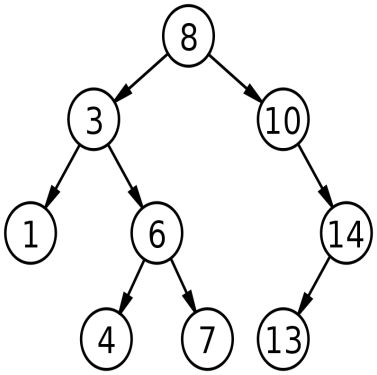
```
> (sort '(8 2 5 2 3) <)
(2 2 3 5 8)
> (sort '(8 2 5 2 3) >)
(8 5 3 3 2)
> (sort '("a" "c" "bo" "za" "lu") string>?)
("za" "lu" "c" "bo" "a")
```

15. Elabore una función llamada *path* que recibe como entrada dos parámetros: un número **n** y un árbol binario de búsqueda (representando con listas) **BST** (el árbol debe contener el número entero **n**). La función debe retornar una lista con la ruta a tomar (iniciando desde el nodo raíz del árbol), indicada por cadenas left y right, hasta llegar al número **n** recibido. Si el número **n** es encontrado en el nodo raíz, el procedimiento debe retornar una lista vacía. *Ejemplo:*

```
> (path 17 '(14 (7 () (12 () ()))
                (26 (20 (17 () ())
                        ())
                    (31 () ()))))
(right left left)
```

Nota aclaratoria: Para el ejercicio número 15, se utiliza la representación de Árbol Binario de Búsqueda con Listas en Racket, y podría representarse con la ayuda de la siguiente gramática BNF:

```
<árbol-binario> := (árbol-vacío) empty
                := (nodo) número <árbol-binario> <árbol-binario>
```



Es decir que este Árbol Binario de Búsqueda, representado en Racket con listas y usando la anterior gramática, sería:

```
'(8 (3 (1 () ()) (6 (4 () ()) (7 () ()))) (10 () (14 (13 () ()) ())))
```

Aclaraciones

1. El taller es en grupos de máximo tres (3) alumnos.

2. La solución del taller debe ser subida al campus virtual a más tardar el día **Viernes 7 de Junio a las 11:59 pm**. Se debe subir al campus virtual en el enlace correspondiente a este taller un archivo comprimido **.zip** que siga la convención *CódigodeEstudiante1-CódigodeEstudiante2-CódigodeEstudiante3-Taller1FLP20191.zip*. Este archivo debe contener el archivo **ejercicios-taller1.rkt** que contenga el desarrollo de los ejercicios.

3. En las primeras líneas del archivo **ejercicios-taller1.rkt** deben estar comentados los nombres y los códigos de los estudiantes participantes.

4. También deben de documentar los procedimientos que hayan implementado como solución a los problemas y de igual manera las funciones auxiliares que se utilicen, con ejemplos de prueba (mínimo 2 pruebas). Por ejemplo, si se pide un procedimiento *remove-first* debe ir así:

```
;; remove-first :
;; Proposito:
;; Procedimiento que remueve la primera ocurrencia de un simbolo
;; en una lista de simbolos.
;;

(define remove-first
   (lambda (s los)
      (if (null? los)
          '()
          (if (eqv? (car los) s)
              (cdr los)
              (cons (car los)
                    (remove-first s (cdr los)))))))

;; Pruebas
(remove-first 'a '(a b c))
(remove-first 'b '(e f g))
(remove-first 'a4 '(c1 a4 c1 a4))
(remove-first 'x '())
```

5. En caso de tener dudas, puede consultar con el profesor **Robinson Duque** los días Jueves de 4 pm a 5 pm (después de clase, en su oficina) o los días Martes de 10 am a 11 am (en su oficina, enviando correo previo para ser atendidos) o consultar con el monitor **Juan Marcos Caicedo** en el horario de atención de los Miércoles de 2 pm a 3 pm (en el Laboratorio del Grupo de Investigación Avispa, 3er piso) y los Jueves de 4 pm a 5 pm (en el mismo lugar).