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

Fundamentos de Lenguajes de Programación / 750095M / Grupo 01 / Profesor Robinson Andrey Duque / Monitora Sara Jazmín Maradiago Calderón / 2021-I

1. Elabore una función llamada *sublistas* que reciba como argumento una lista **lst** y que retorne la cantidad de sublistas que posee. *Por ejemplo:*

```
> (sublistas '(4 5 8 (1 2) 8 (7 8 9 (2 4))))
3

> (sublistas '(((a b) c) (1 (2 (3 4)) (d e f (g))) 5 ((6 (7) 8) 9)))
10
```

2. Elabore una función llamada *filtro* que reciba dos argumentos: un predicado **pred** y una lista **lst**. La función debe retornar una lista con los elementos de **lst** que cumplen con el predicado **pred**. *Por ejemplo:*

```
> (filtro number? '((1 2 a (b)) 3 4 c 5 d))
'(3 4 5)

> (filtro symbol? '((1 2 a (b)) 3 4 c 5 d))
'(c d)

> (filtro list? '((1 2 a (b)) 3 4 c 5 d))
'((1 2 a (b)))
```

3. Elabore una función llamada *inversion-listas* que recibe como argumento una lista **lst**, que a su vez contiene como elementos listas 2-list (listas de 2 elementos). La función debe retornar cada 2-list invertida. *Por ejemplo:*

```
> (inversion-listas '((2 1) (4 3) (b a) (d c)))
'((1 2) (3 4) (a b) (c d))

> (inversion-listas '((es Racket) (genial muy) (21 20)))
'((Racket es) (muy genial) (20 21))
```

4. Elabore una función llamada *situar-en-lista* que debe recibir tres argumentos: una lista **lst**, una posición **pos** y un elemento **elem**. La función debe retornar una lista similar a **lst**, pero en la que se ha reemplazado el elemento en la posición **pos** con el elemento **elem**, comenzando los índices desde cero. *Por ejemplo:*

```
> (situar-en-lista '(1 2 3 4 5 6 7 8 9) 0 'Comienzo)
'(Comienzo 2 3 4 5 6 7 8 9)

> (situar-en-lista '(1 2 3 4 5 6 7 8 9) 4 'Mitad)
'(1 2 3 4 Mitad 6 7 8 9)

> (situar-en-lista '(1 2 3 4 5 6 7 8 9) 8 'Final)
'(1 2 3 4 5 6 7 8 9 Final)
```

5. Elabore una función llamada *ordenar* que debe recibir dos argumentos: un predicado **pred** y una lista de números **lst-num**. La función debe ordenar la lista **lst-num** según el predicado **pred**, que puede tomar el valor de **mayor que** ó **menor qué**. *Por ejemplo:*

```
> (ordenar < '(58 41 67 54 32 10))
'(10 32 41 54 58 67)

> (ordenar > '(58 41 67 54 32 10))
'(67 58 54 41 32 10)
```

6. Elabore una función llamada *indice-lista* que retorna la posición (en índice en base cero) del primer elemento en la lista **lst** que cumpla con el predicado **pred**. Utilice *lambda* y *letrec* para este ejercicio. *Por ejemplo:*

```
> (indice-lista (lambda (x) (eqv? x 'd)) '(a b c d e f g))
3

> (indice-lista (lambda (x) (> x 3)) '(0 1 2 3 4 5 6))
4
```

7. Elabore una función llamada *contar-ocurrencias* que recibe como argumentos un elemento símbolo **elem** y una lista **S-list**, acorde a la siguiente gramática:

```
<S-list> ::= ({<S-exp>}*)

<S-exp> ::= <Symbol> | <S-list>
```

La función debe retornar el número de ocurrencias de **elem** en **S-list**. *Por ejemplo:*

```
> (contar-ocurrencias 'x '((f x) y (((x z) x))))
3

> (contar-ocurrencias 'x '((f x) y (((x z) () x))))
3

> (contar-ocurrencias 'w '((f x) y (((x z) x))))
0
```

8. Elabore una función llamada *intercambio* que recibe tres argumentos: dos elementos **elem1** y **elem2**, que pertenecen a una lista **S-list**. La función debe retornar una lista similar a **S-list**, sólo que cada ocurrencia anterior de **elem1** será reemplazada por **elem2** y cada ocurrencia anterior de **elem2** será reemplazada por **elem1**. Por ejemplo:

```
> (intercambio 'a 'd '(a b c d))
(d b c a)

> (intercambio 'a 'd '(a d () c d))
(d a () c a)

> (intercambio 'x 'y '((x) y (z (x))))
((y) x (z (y)))
```

9. Elabore una función llamada *producto* que recibe como argumento dos listas **lst1** y **lst2**, y retorna una lista de listas 2-list que represente el producto cartesiano de **lst1** y **lst2**. Las listas 2-list pueden aparecer en cualquier orden. *Por ejemplo:*

```
> (producto '(a b c) '(x y))
((a x) (a y) (b x) (b y) (c x) (c y))
```

10. Elabore una función llamada *filter-acum* que recibe como entrada cinco 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**. *Por ejemplo:*

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

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

11. Elabore una función llamada *list-append* que recibe como argumentos dos listas *lst1* y *lst2*. Debe retornar una nueva lista con los elementos de *lst2* añadidos a los de *lst1*, sin usar la función *append* de Racket. *Por ejemplo:*

```
> (list-append '(1 2 3) '(4 5))
(1 2 3 4 5)
```

12. Elabore una función llamada *operate* que recibe dos argumentos: donde *lrators* es una lista de funciones binarias de tamaño *n* y *lrands* es una lista de números de tamaño *n + 1*. La función retorna el resultado de aplicar sucesivamente las operaciones en *lrators* a los valores en *lrands*.

```
> (operate (list + * + - *) '(1 2 8 4 11 6))
102

> (operate (list *) '(4 5))
20
```

13. 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**. *Por ejemplo:*

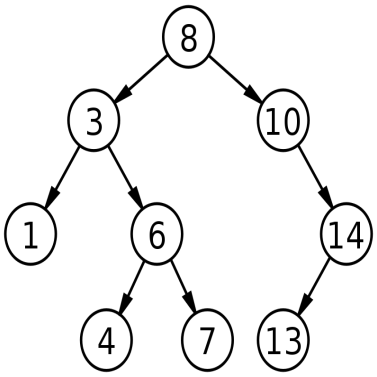
```
> (zip + '(1 4) '(6 2))
(7 6)

> (zip * '(11 5 6) '(10 9 8))
(110 45 48)
```

14. 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. *Por ejemplo:*

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

Nota aclaratoria: Para el ejercicio número 14, 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 () ()) ())))
```

15. Elabore un procedimiento llamado *compose*, que recibe dos argumentos **proc1** y **proc2** que son prodecimientos de un argumento, y un valor **val**. Debe retornar la composición de ambos procedimientos aplicados sobre **val**. Utilice dos *lambda*. **Por ejemplo:**

```
> ((compose car cdr) '(a b c d))
b

> ((compose number? car) '(a b c d))
#f
```

```
> ((compose symbol? (compose car cdr)) '(a b c d))
#t

> ((compose boolean? even?) 5)
#t

> ((compose list? cdr) '(+ 2 6))
#t

> ((compose number? (compose car cdr)) '(+ 2 6))
#t
```

Recordemos que la composición de funciones se expresa:

```
 (g o f) (x) = g[f(x)]
```

Donde **f** y **g** son funciones.

Sean *Dom f* el dominio de **f** y *Dom g*, el de **g**, entonces el dominio de la función composición es:

```
 Dom (g o f) (x) = { x ∈ Dom f | f(x) ∈ Dom g }
```

16. Elabore una funcion llamado *carCdr*, que recibe tres argumentos: un elemento **elem**, una lista **lst** y un valor de error **errvalue**. Debe retornar una expresión que cuando se evalúa, produce el código para un procedimiento que toma una lista con la misma estructura que **lst** y devuelve el valor en la misma posición que la primera aparición de **elem**. Si **elem** no existe en **lst**, se retorna **errvalue**. Haga esto para que genere composiciones de procedimiento. *Por ejemplo:*

```
> (carCdr 'a '(a b c) 'fail)
car

> (carCdr 'c '(a b c) 'fail)

(compose (compose car cdr) cdr)
```

```
> (carCdr 'dog '(cat lion (fish dog ()) pig) 'fail)
(compose
  (compose (compose (compose car cdr) car) cdr)
  cdr)

> (carCdr 'dog '((fish dog ()) dog) 'fail)
(compose (compose car cdr) car)

> (carCdr 'dog '((dog) fish) 'fail)
(compose car car)

> (carCdr 'dog '(whale (cat) ((mouse ((dog) snake) fish) cow) (dog fish)) 'fail)
(compose
  (compose
    (compose
      (compose
        (compose (compose (compose car car) car) cdr)
        car)
      car)
    cdr)
  cdr)

> (carCdr 'a '() 'fail)
fail
```

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 **primero de marzo de 2021, a las 23:59**. Se debe subir al campus virtual en el enlace correspondiente a este taller un archivo comprimido **.zip** que siga la convención *CódigodeEstudiante1-CódigodeEstudiante2-CódigodeEstudiante3-Taller1FLP2021-I.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 Andrey Duque** los días Martes de 10 a 11, confirmando su asistencia un día antes por correo. O consultar con la monitora **Sara Jazmín Maradiago Calderón** en el horario de atención los Lunes de 14 a 16 en su reunión de Meet:

   https://meet.google.com/bpu-dagg-ivj?authuser=3