Taller 1
Fundamentos de Lenguajes Programación

Carlos Andres Delgado S, Ing *

Octubre de 2015

Implemente en Scheme los siguientes procedimientos:

1. (**2pts.**)`(list-tails lista)` que recibe como parámetro una lista y retorna una lista de todas las sublistas de elementos consecutivos en `lista`.

```
> (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))
```

2. (**4pts.**)`(exists? pred lst)` recibe como parametros un predicado (number? symbol? bool? list?) y retorna #t si al menos un elemento de la lista satisface el predicado

```
> (exists? number? ?(a b c 3 e))
#t
> (exists? number? ?(a b c d e))
#f
```

3. (**3pts.**)`(list-facts n)` que recibe como parámetro un entero `n` y retorna una lista incremental de factoriales, comenzando en *1!* hasta *n!*

```
> (list-facts 5)
(1 2 6 24 120)
```

4. (**4pts.**)`(simpson-rule f a b n)` que cálcula la integral de una función `f` entre los valores `a` y `b` mediante la regla de Simpson. Dicha regla utiliza la siguiente aproximación:

$$\frac{h}{3} \cdot \left(y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n\right)$$

donde $h = \frac{(b-a)}{n}$ para algún entero par `n` y $y_k = f(a + kh)$.

```
> (simpson-rule (lambda (x) (* x (* x x))) 1 5 8)
156
> (simpson-rule (lambda (x) x) 1 5 12)
12
```

5. (**4pts.**)`(inversions lista)` que determina el número de inversiones de `lista`. Sea $A = (a_1 a_2 \ldots a_n)$ una lista de n números diferentes, si $i < j$ y $a_i > a_j$ entonces la pareja $(i\ j)$ es una inversión de A.

*carlos.andres.delgado@correounivalle.edu.co

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

6. (**4pts.**)`(up lista)` que remueve un par de paréntesis de cada elemento del nivel más alto de `lista`. Si un elemento de este nivel no es una lista (no tiene paréntesis), este elemento es incluido en el resultado sin modificaciones.

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

7. (**4pts.**)`(merge lista1 lista2)`, donde `lista1` y `lista2` son listas de enteros ordenadas ascendentemente. El procedimiento `merge` retorna una lista ordenada de todos los elementos en `lista1` y `lista2`.

```
> (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)
```

8. (**4pts.**)`(zip f lista1 lista2)` que retorna una lista donde la posición n-sima corresponde al resultado de aplicar la función `f` sobre los elementos en la posición n-sima en `lista1` y `lista2`. Puede asumir que `f` es una función que recibe dos argumentos y que ambas listan tienen igual tamaño.

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

9. (**2pts.**)`(max-list lista)` que recibe una lista de números y devuelve el número mayor en dicha lista.

```
> (max-list '(12 4 21 11 8))
21
> (max-list '(19 12 34 8 14 45))
45
```

10. (**4pts.**)`(foldl lista f acum)` que recibe una lista `lista`, una función `f` y un valor inicial `acum`. La función `f` es una función que recibe dos parámetros. Este procedimiento aplicará la función `f` a cada elemento de la lista y a un valor acumulado inicialmente correspondiente al valor inicial `acum`.

```
> (foldl '(12 4 21 11 8) + 0)
56
> (foldl '(19 12 34 8 14 45) max 0)
45
```

11. (**4pts.**)`(filter-acum a b f acum filter)` que recibe dos números `a` y `b`, una función binaria `f`, un valor inicial `acum` y una función unaria `filter`. Este procedimiento aplicará la función `f` a cada valor entre `a` y `b` y a un valor acumulado que inicialmente corresponde al valor inicial `acum`.

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

12. (**4pts.**)`(sort lista f)` que retorna la lista `lista` ordenada ascendentemente aplicando la función de comparación `f`.

```
> (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")
```

13. (**4pts.**)`(path num arbol)` que toma un entero `num` y un árbol binario de búsqueda que contiene el entero `num` y retorna una lista de `lefts` y `rights` indicando como encontrar el nodo que contiene el entero `num` en el árbol. Si `num` es encontrado en la raíz, el procedimiento retorna una lista vacia.

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

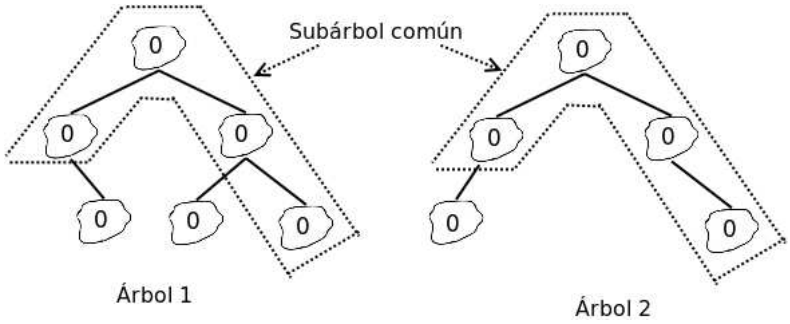
14. (**5pts.**)`(prod-scalar-matrix mat vec)` que recibe una matriz `mat` representada como una lista de listas y un vector `vec` representado como una lista y retorna el resultado de realizar la multiplicación matriz por vector.

```
> (prod-scalar-matriz '((1 1) (2 2)) '(2 3))
'((2 3) (4 6))
> (prod-scalar-matriz '((1 1) (2 2) (3 3)) '(2 3))
'((2 3) (4 6) (6 9))
```

Nota: Puede asumir que los tamaños de la matriz y el vector son coherentes.

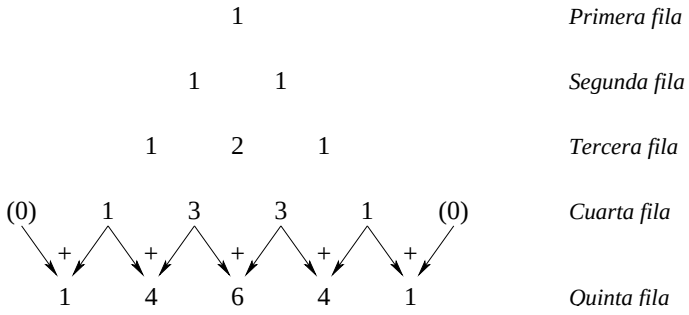
15. (**5pts.**)`(common-subtree arbol1 arbol2)` que recibe dos arboles n-arios y retorna el subarbol común (estructuralmente). El subarbol común se compone de las ramas que existen en ambos árboles. En este ejercicio el dato existente en cada nodo es irrelevante por lo que puede asumir que siempre será 0.

    A continuación se muestra un ejemplo del subarbol común para dos árboles binarios:



16. (**5pts.**)`(pascal N)` que retorna la fila `N` del triangulo de Pascal.

    A continuación se muestra las primeras cinco filas del triangulo de Pascal:



Ayuda: Note que la fila `N` depende de la anterior, por ejemplo la quinta fila:

```
(1 4 6 4 1) =
(0 1 3 3 1) +
(1 3 3 1 0)
```

```
> (pascal 5)
(1 4 6 4 1)
> (pascal 1)
(1)
```

17. (**6pts.**)`(remove-bin-stree arbol val)` que toma un árbol binario de búsqueda `arbol` y retorna un nuevo árbol binario de búsqueda donde se ha removido el elemento `val`.

```
> (remove-bin-stree '(14 (7 () (12 () ()))
                         (26 (20 (17 () ())
                                 ())
                             (31 () ())))
                    20)
(14 (7 () (12 () ()))
    (26 (17 () ())
        (31 () ())))
> (remove-bin-stree '(14 (7 () (12 () ()))
                         (26 (20 (17 () ())
                                 ())
                             (31 () ())))
                    26)
(14 (7 () (12 () ()))
    (31 (20 (17 () ())
        ()))
```

Aclaraciones

1. El taller es en grupos de mínimo (2) estudiantes y máximo tres (3) estudiantes.
2. La solución del taller debe ser subida al campus virtual en la fecha especificada por el docente
3. Las primeras líneas de cada archivo deben contener los nombres y códigos de los estudiantes.
4. En ese mismo archivo, vendrán comentados los procedimientos que llevan al código de la declaración, las operaciones, la expresión BNF de las estructuras que se están utilizando, y algunos ejemplos de prueba. Por ejemplo, si se pidiera construir el procedimiento remove-first, deberá existir un código como:

```
;; <lista-de-simbolos> := ({<exp-simbolo>}*)
;; <exp-simbolo> := <simbolo> | <lista-de-simbolos>
;;
;; remove-first : simbolo * lista-de-simbolos -> lista-de-simbolos
;;
;; 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 '())
```