

# PRÁCTICA 1

## INTELIGENCIA ARTIFICIAL

Jorge Santisteban Rivas  
Javier Martínez Rubio

# Ejercicio 1

## Apartado 1.1

### Prod-esc-rec

#### Pseudocódigo

**Entradas** x: vector, representado como una lista

y: vector, representado como una lista

**Salida:** producto escalar entre x e y

#### Procesamiento

si  $x = 0$  o  $y = 0$

devuelve 0

en caso contrario,

suma el resultado de multiplicar las primeras coordenadas de cada lista con el resultado de aplicar la misma función al resto de cada lista

#### Código

```
;;; producto-escalar-rec (x y)
;;; Calcula el producto escalar de forma recursiva
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
```

```
(defun prod-esc-rec (x y)
  (if (or (null x) (null y))
      0
      (+ (* (first x) (first y))
         (prod-esc-rec (rest x) (rest y)))))
```

#### Comentarios

Es una función que calcula el producto escalar de dos vectores de forma recursiva.

### Cosine-distance-rec

#### Pseudocódigo

**Entradas** x: vector, representado como una lista

y: vector, representado como una lista

**Salida:** distancia coseno entre x e y

#### Procesamiento

si  $x = 0$  o  $y = 0$

devuelve 0

si la norma de x o la norma de y son 0,

devuelve 0

en caso contrario,

calcula la distancia coseno según su fórmula haciendo

uso de la función auxiliar que calcula el producto escalar entre dos vectores.

## Código

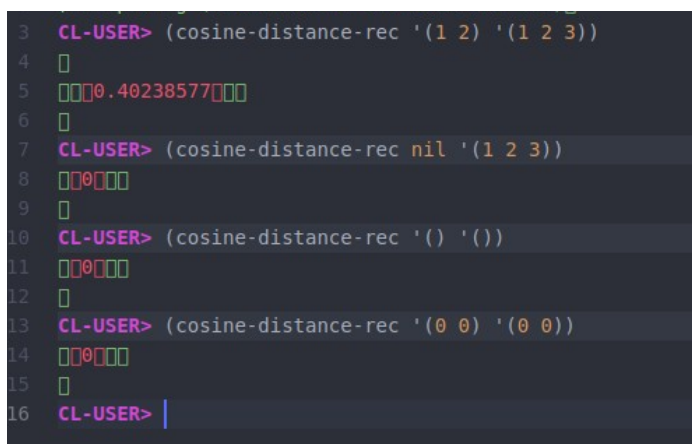
```
;;; cosine-distance-rec (x y)
;;; Calcula la distancia coseno de un vector de forma recursiva
;;; Se asume que los dos vectores de entrada tienen la misma longitud.

(defun cosine-distance-rec (x y)
  (cond ((or (null x) (null y)) 0)
        ((= 0 (* (prod-esc-rec x x) (prod-esc-rec y y))) 0)
        (t (- 1 (/ (prod-esc-rec x y)
                    (* (sqrt(prod-esc-rec x x))
                      (sqrt(prod-esc-rec y y))))))))
```

## Comentarios

Esta función calcula la distancia coseno entre dos listas de manera recursiva. Esta recursividad la vemos en el uso de una función recursiva de cálculo del producto escalar, que hemos programado anteriormente.

## Ejemplos



```
3 CL-USER> (cosine-distance-rec '(1 2) '(1 2 3))
4
5 0.40238577
6
7 CL-USER> (cosine-distance-rec nil '(1 2 3))
8 0
9
10 CL-USER> (cosine-distance-rec '() '())
11 0
12
13 CL-USER> (cosine-distance-rec '(0 0) '(0 0))
14 0
15
16 CL-USER> |
```

## Prod-esc-mapcar

### Pseudocódigo

**Entradas** x: vector, representado como una lista

y: vector, representado como una lista

**Salida:** producto escalar entre x e y

### Procesamiento

si  $x = 0$  o  $y = 0$

devuelve 0

en caso contrario,

suma el resultado de multiplicar las primeras coordenadas de cada lista con el resultado de aplicar la misma función al resto de cada lista

## Código

```
;;; producto-escalar-rec (x y)
;;; Calcula el producto escalar usando mapcar
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
```

```
(defun prod-esc-mapcar (x y)
  (if (or (null x) (null y))
      0
      (apply #'+ (mapcar #'* x y))))
```

## Comentarios

Es una función que calcula el producto escalar de dos vectores sin recursividad, utilizando la función mapcar.

## Cosine-distance-mapcar

### Pseudocódigo

**Entradas** x: vector, representado como una lista  
y: vector, representado como una lista

**Salida:** distancia coseno entre x e y

### Procesamiento

si  $x = 0$  o  $y = 0$   
devuelve 0  
si la norma de x o la norma de y son 0,  
devuelve 0  
en caso contrario,  
calcula la distancia coseno según su fórmula haciendo  
uso de la función auxiliar que calcula el producto escalar  
entre dos vectores.

## Código

```
;;; cosine-distance-mapcar (x y)
;;; Calcula la distancia coseno de un vector usando mapcar
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
```

```
(defun cosine-distance-mapcar (x y)
  (cond ((or (null x) (null y)) 0)
        ((= 0 (* (prod-esc-mapcar x x) (prod-esc-mapcar y y))) 0)
        (t (- 1 (/ (prod-esc-mapcar x y)
                    (* (sqrt(prod-esc-mapcar x x))
                      (sqrt(prod-esc-mapcar y y))))))))
```

## Comentarios

Esta función calcula la distancia coseno entre dos listas utilizando la función mapcar, ya que hace uso de una función que calcula el producto escalar usando mapcar.

## Ejemplos

```
CL-USER> (cosine-distance-mapcar '(1 2) '(1 2 3))
0.40238577
CL-USER> (cosine-distance-mapcar nil '(1 2 3))
0
CL-USER> (cosine-distance-mapcar '() '())
0
CL-USER> (cosine-distance-mapcar '(0 0) '(0 0))
0
```

## Apartado 1.2

### Order-lst-vectors

Función auxiliar que hemos creado para que pasándole la lista de vectores ordenados inserte un nuevo vector correctamente

#### **Pseudocódigo**

**Entrada:** vector-ref: vector que representa a una categoría, representado como una lista

vector-insert: vector a insertar

ord-lst-of-vectors: nuevo vector de vectores ordenados

**Salida:** la nueva lista ordenada con el vector insertado

#### **Procesamiento:**

Si la lista de vectores ordenados es vacía

Insertamos el vector a insertar en la lista

Si la distancia coseno del vector es menor que la del primer elemento de la lista ordenada

Devolvemos la lista ordenada con el vector como primer elemento

En otro caso

Devolvemos la lista con el primer elemento de la lista ordenada y la llamada recursiva de order-lst-vectors aplicada al vector y al *rest* de la lista ordenada.

#### **Código**

```
;;;; order-lst-vectors
```

```
(defun order-lst-vectors(vector-ref vector-insert ord-lst-of-vectors)
  (cond ((null ord-lst-of-vectors) (cons vector-insert ord-lst-of-vectors))
        ((< (cosine-distance-mapcar vector-ref vector-insert)
             (cosine-distance-mapcar vector-ref (first ord-lst-of-vectors)))
         (cons vector-insert ord-lst-of-vectors))
        (t (cons (first ord-lst-of-vectors)
                   (order-lst-vectors vector-ref vector-insert (rest ord-lst-of-vectors))))))
```

#### **Comentarios**

En este caso hemos utilizado recursividad para ir insertando elementos de manera ordenada a una lista.

## Order-vectors-cosine-distance

### Pseudocódigo

**Entrada:** vector: vector que representa a una categoría, representado como una lista  
lst-of-vectors vector de vectores  
confidence-level: Nivel de confianza (parametro opcional)

**Salida:** Vectores cuya semejanza con respecto a la categoría es superior al nivel de confianza

### Procesamiento:

Si  $1 - \text{nivel de confianza} \geq \text{distancia del primero de la lista}$

Si *rest* == null

Insertamos el primer elemento en una lista vacia (con order-lst-vectors)

En otro caso

Recurción de order-lst-vectors para el resto de elementos

En otro caso

Si *rest* == null

Nil

En otro caso

Recurción de la función para aplicarla al resto de los vectores que queremos comparar con el vector referencia

### Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;; order-vectors-cosine-distance
```

```
;;; Devuelve aquellos vectores similares a una categoria
```

```
(defun order-vectors-cosine-distance (vector lst-of-vectors &optional (confidence-level 0))
```

```
  (if (>= (- 1 confidence-level)
```

```
      (cosine-distance-mapcar vector (first lst-of-vectors)))
```

```
      (if (null (rest lst-of-vectors))
```

```
          (order-lst-vectors vector (first lst-of-vectors) '())
```

```
          (order-lst-vectors vector (first lst-of-vectors) (order-vectors-cosine-distance  
                                                                    vector (rest lst-of-vectors) confidence-level)))
```

```
      (if (null (rest lst-of-vectors))
```

```
          nil
```

```
          (order-vectors-cosine-distance vector (rest lst-of-vectors) confidence-level))))
```

### Comentarios

Hemos utilizado la función auxiliar para ir insertando (si cumple la condición) uno a uno en la lista y de manera ordenada, aplicando recursividad sobre la propia función.

## Ejemplos

```
7 CL-USER> (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2))) 0.5)
8
9
10 CL-USER> (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2))) 0.3)
11
12 CL-USER> (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2))) 0.99)
13
14 CL-USER> (order-vectors-cosine-distance '(1 2 3) '())
15
16 CL-USER> (order-vectors-cosine-distance '() '((4 3 2) (1 2 3)))
17
```

## Apartado 1.3

### Get-text-category

Función auxiliar que a partir de un text y una lista de categorías devuelve el id de la categoría que mejor lo aproxima y la distancia a ella.

### Pseudocódigo

**Entrada:** categories: vector de vectores, representado como una lista de listas

text: vector, representado como una lista

distance-measure: funcion de distancia

min-category: la categoria minima para comenzar la iteracion (la primera categoria por defecto)

**Salida:** Par formado por el vector que identifica la categoria de menor distancia , junto con el valor de dicha distancia.

### Procesamiento:

Si lista\_categorias == null

devolver una lista con el primer elemento de min-category y la distancia del texto con esa categoria

En otro caso

Si la distancia entre la primera categoria y el texto es menor que la distancia entre la min-category y el texto

Esta pasa a ser la min-category y hacemos una llamada recursiva

En otro caso

Llamada recursiva manteniendo min-category

### Código

```
.....
;;; get-text-category (categories text distance-measure min-category)
;
;; A partir de un texto devolvemos el identificador de la categoria que lo aproxima y su distancia.
(defun get-text-category (categories text distance-measure min-category)
  (if (null categories)
      (list (first min-category) (funcall distance-measure (rest text) (rest min-category)))
      (if (< (funcall distance-measure (rest (first categories)) (rest text))
          (funcall distance-measure (rest min-category) (rest text)))
          (get-text-category (rest categories) text distance-measure (first categories))
          (get-text-category (rest categories) text distance-measure min-category)))))
```

## Comentarios

En esta función utilizamos recursividad para ir manteniendo siempre la menor categoría, siendo el primer valor de esta el primer elemento de la lista de categorías. Tras haber pasado por todas las categorías devolvemos simplemente el identificador y la distancia a esta.

### Get-vectors category

#### Pseudocódigo

**Entrada:** categories: vector de vectores, representado como una lista de listas

texts: vector de vectores, representado como una lista de listas

**Salida:** Pares formados por el vector que identifica la categoría de menor distancia , junto con el valor de dicha distancia

#### Procesamiento:

Con mapcar y una función lambda ir aplicando a todos los elementos de texts la función get-text-category, devolviendo una lista final con todos los ids y las distancias.

#### Código

```
.....  
;;; get-vectors-category (categories vectors distance-measure)  
(defun get-vectors-category (categories texts distance-measure)  
  (mapcar #'(lambda(x) (get-text-category categories x distance-measure (first categories))) texts))
```

#### Comentarios

Una vez ya programada la función que averigua la mejor categoría, al final en esta simplemente recurriendo a la función mapcar y una lambda conseguimos una lista con todos los pares de id y distancia.

## Apartado 1.4

Utilizamos la función para los siguientes casos, midiendo el tiempo y usando tanto la implementada recursivamente como la que usa mapcar:

### 1. (get-vectors-categories '() '()) #'cosine-distance)

```
110 CL-USER> (time (get-vectors-category '() '()) #'cosine-distance-rec))  
111 [Evaluation took:  
112   0.000 seconds of real time  
113   0.000010 seconds of total run time (0.000008 user, 0.000002 system)  
114   100.00% CPU  
115   8,468 processor cycles  
116   0 bytes consed  
117 ]  
118 [[(NIL 0)]]  
119 ]  
120 CL-USER> (time (get-vectors-category '() '()) #'cosine-distance-mapcar))  
121 [Evaluation took:  
122   0.000 seconds of real time  
123   0.000009 seconds of total run time (0.000009 user, 0.000000 system)  
124   100.00% CPU  
125   8,300 processor cycles  
126   0 bytes consed  
127 ]  
128 [[(NIL 0)]]
```

Vemos que comparando tanto categorías como textos vacíos ambas funciones se comportan igual, devolviendo NIL como id y 0 como distancia.



## 2. (get-vectors-categories '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance)

```
260 CL-USER> (time (get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-rec))
261 []
262 Evaluation took:
263   0.000 seconds of real time
264   0.000024 seconds of total run time (0.000021 user, 0.000003 system)
265   100.00% CPU
266   49,604 processor cycles
267   0 bytes consed
268
269 []((2 0.40238577))[]
270 []
271 CL-USER> (time (get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-mapcar))
272 []Evaluation took:
273   0.000 seconds of real time
274   0.000027 seconds of total run time (0.000027 user, 0.000000 system)
275   100.00% CPU
276   54,492 processor cycles
277   0 bytes consed
278
279 []((2 0.40238577))[]
280 []
```

Vemos que para ambos da el mismo resultado pero la recursiva tarda ligeramente menos tiempo que la que utiliza mapcar.

## 3. (get-vectors-categories '() '((1 1 2 3) (2 4 5 6)) #'cosine-distance)

```
281 CL-USER> (time (get-vectors-category '() '((1 1 2 3) (2 4 5 6)) #'cosine-distance-rec))
282 []Evaluation took:
283   0.000 seconds of real time
284   0.000009 seconds of total run time (0.000009 user, 0.000000 system)
285   100.00% CPU
286   10,524 processor cycles
287   0 bytes consed
288
289 []((NIL 0) (NIL 0))[]
290 []
291 CL-USER> (time (get-vectors-category '() '((1 1 2 3) (2 4 5 6)) #'cosine-distance-mapcar))
292 []Evaluation took:
293   0.000 seconds of real time
294   0.000002 seconds of total run time (0.000002 user, 0.000000 system)
295   100.00% CPU
296   2,968 processor cycles
297   0 bytes consed
298
299 []((NIL 0) (NIL 0))[]
300 []
```

Para esos dos textos vemos que al no haber categorías, los resultados son NIL y 0 para cada uno de los pares. Vemos que en este caso la recursiva tarda mucho mas tiempo que la que usa mapcar.

# Ejercicio 2

## Apartado 2.1

### Newton

### Pseudocódigo

**Entradas:** f: funcion cuyo cero se desea encontrar  
df: derivada de f  
max-iter: maximo numero de iteraciones  
x0: estimacion inicial del cero (semilla)  
tol: tolerancia para convergencia (parametro opcional)

**Salida:** estimacion del cero de f o NIL si no converge

### Procesamiento

si max-iter = -1  
devuelve nil  
en caso contrario,  
se vuelve a llamar a la misma función con max-iter = max-iter -1  
y con x0 como la nueva semilla calculada según la fórmula.

### Código

```
;;; newton
;;; Estima el cero de una funcion mediante Newton-Raphson

(defun newton (f df max-iter x0 &optional (tol 0.001))
  (if (= max-iter 0) NIL
      (if (< (abs (funcall f x0)) tol) x0
          (newton f df (- max-iter 1) (- x0 (/ (funcall f x0) (funcall df x0))) tol))))
```

### Comentarios

Esta función estima (con una cierta tolerancia), a partir de una semilla dada, la raíz de una función pasada como parámetro utilizando el método de Newton. Este es un método iterativo en el cual la semilla va cambiando en cada iteración de acuerdo a una cierta fórmula. En cada una de estas iteraciones, se evalúa la función en la semilla, y si este es 0, se devuelve la semilla como la raíz. Si no lo es, se evalúa la función en la nueva raíz. Además, hay un número máximo de iteraciones, y, en caso de no encontrar ninguna raíz en este periodo, se devuelve NIL.

La función que hemos programado es, por tanto, recursiva. Primero comprobamos si ya se han superado las iteraciones máximas. Ya que hacemos esta comprobación la primera, aparece en el código como max-iter = -1, y no como max-iter = 0. Después miramos si  $f(x_0) = 0$ , y en caso de serlo, devolvemos  $x_0$ . En otro caso, hacemos la recursividad, llamando a newton con los nuevos valores en sus argumentos.

### Ejemplos

```
CL-USER> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 3.0)
0.0004.0000
CL-USER> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 0.6)
0.999999946000
CL-USER> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 -2.5)
-3.0000203000
CL-USER> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 10 100)
NIL
```

## Apartado 2.2

### One-root-newton

#### Pseudocódigo

**Entradas:** f: función cuyo cero se desea encontrar  
df: derivada de f  
max-iter: máximo número de iteraciones  
semillas: semillas con las que invocar a Newton  
tol: tolerancia para convergencia (parámetro opcional)

**Salida:** el primer cero de f que se encuentre, o NIL si se diverge para todas las semillas

#### Procesamiento

si semillas = nil  
devuelve nil  
si one-root-newton != nil  
devuelve newton f df max-iter (first semillas) tol  
en caso contrario,  
se hace one-root-newton f df max-iter (rest semillas) tol

#### Código

```
;;; one-root-newton  
;;; Prueba con distintas semillas iniciales hasta que Newton  
;;; converge
```

```
(defun one-root-newton (f df max-iter semillas &optional (tol 0.001))  
  (cond ((null semillas) nil)  
        ((newton f df max-iter (first semillas) tol)  
         (newton f df max-iter (first semillas) tol))  
        (t (one-root-newton f df max-iter (rest semillas) tol))))
```

#### Comentarios

Esta función es también recursiva, ya que tiene que evaluar la función newton en cada una de las semillas de la lista pasada como argumento hasta que se encuentra una que no sea nil. Simplemente se comprueba esta condición, y si no se cumple se vuelve a llamar a la misma función con el resto de la lista de semillas.

#### Ejemplos

```
CL-USER> (one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda(x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5))  
0  
0.99999946  
0  
CL-USER> (one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda(x) (- (* x (- (* x 3) 4)) 11)) 20 '(3.0 -2.5))  
0  
4.0  
0  
CL-USER> (one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda(x) (- (* x (- (* x 3) 4)) 11)) 1 '(3.0 -2.5))  
0  
NIL  
0  
CL-USER>
```

## Apartado 2.3

### ALL-roots-newton

#### Pseudocódigo

**Entradas:** f: funcion cuyo cero se desea encontrar  
df: derivada de f  
max-iter: maximo numero de iteraciones  
semillas: semillas con las que invocar a Newton  
tol: tolerancia para convergencia (parametro opcional)

**Salida:** las raices que se encuentren para cada semilla o nil  
si para esa semilla el metodo no converge

#### Procesamiento

aplicamos newton pasando como semilla cada elemento de la lista  
pasada como argumento

#### Código

```
;;; all-roots-newton  
;;; Prueba con distintas semillas iniciales y devuelve las raices  
;;; encontradas por Newton para dichas semillas
```

```
(defun all-roots-newton (f df max-iter semillas &optional ( tol 0.001))  
  (mapcar #'(lambda(x) (newton f df max-iter x tol)) semillas))
```

#### Comentarios

En esta función aplicamos un mapcar para así simplificar. También tuvimos en cuenta implementarlo de manera recursiva, pero decidimos que esta manera era mucho más simple.

#### Ejemplos

```
CL-USER> (all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5))  
[]  
[][(0.99999946 4.0 -3.0000203)][]  
CL-USER> (all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0))  
[][(0.99999946 4.0 NIL)][]  
CL-USER> (all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(0.6 3.0 10000.0))  
[(NIL NIL NIL)][]  
CL-USER>
```

## Apartado 2.3.1

### list-not-nil-roots-newton

#### Pseudocódigo

**Entradas:** f: funcion cuyo cero se desea encontrar  
df: derivada de f  
max-iter: maximo numero de iteraciones

semillas: semillas con las que invocar a Newton  
tol: tolerancia para convergencia (parametro opcional)

**Salida:** las raices que se encuentren para cada semilla

### Procesamiento

aplicamos all roots newton pero solo seleccionamos las que el resultado no sea nil

### Código

```
.....  
;;; list-not-nil-newton
```

```
(defun list-not-nil-roots-newton (f df max-iter semillas &optional (tol 0.001))  
  (mapcan #'(lambda(x) (unless (null x) (list x))) (all-roots-newton f df max-iter semillas tol))))
```

### Comentarios

En esta función aplicamos un mapcar para así simplificar. Además gracias al unless solo seleccionamos los resultados que no sean nil. También tuvimos en cuenta implementarlo de manera recursiva, pero decidimos que esta manera era mucho más simple.

### Ejemplos

```
CL-USER> (list-not-nil-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0))  
[]  
[](0.99999946 4.0)[]  
[]  
CL-USER> (all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(0.6 3.0 10000.0))  
[](0.99999946 4.0 NIL)[]  
[]  
CL-USER> |
```

## Ejercicio 3

### Apartado 3.1

#### Combine-elt-lst

##### Pseudocódigo

**Entrada:** elt: elemento a combinar con la lista

lst: lista a combinar con el elemento

**Salida:** Devuelve las listas resultantes de combinar el elemento con cada uno de los elementos de la lista.

##### Procesamiento:

Si el elemento o la lista es null

Nil

En otro caso, usando mapcar combinamos uno a uno cada uno de los elementos de la lista con el elemento pasado como argumento.

### Código

```
.....  
;;; combine-elt-lst  
;;; Combina un elemento dado con todos los elementos de una lista
```

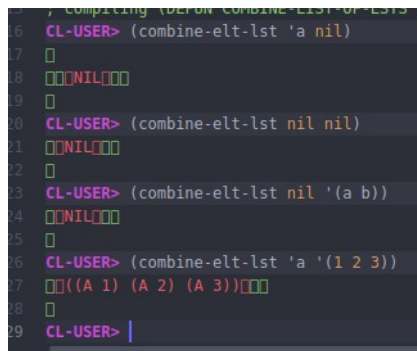
```
(defun combine-elt-lst (elt lst)
```

```
(cond ((or (null elt) (null lst))
      nil)
      (t (mapcar #'(lambda(x) (list elt x)) lst))))
```

## Comentarios

En lugar de utilizar la recursión, a partir de la función mapcar y una función lambda que crea una lista a partir de dos elementos vamos recorriendo la lista elemento a elemento y generando todas las listas demandadas.

## Ejemplos



```
CL-USER> (combine-elt-lst 'a nil)
()
CL-USER> (combine-elt-lst nil nil)
()
CL-USER> (combine-elt-lst nil '(a b))
()
CL-USER> (combine-elt-lst 'a '(1 2 3))
((A 1) (A 2) (A 3))
```

## Apartado 3.2

### Combine-lst-lst

#### Pseudocódigo

**Entrada:** lst1: primera lista

lst2: segunda lista

**Salida:** lista con las combinaciones del elemento con cada uno de los de la lista

#### Procesamiento:

Si alguna de las listas es null

Nil

En otro caso, utilizando mapcan y una función lambda que va elemento a elemento de la primera lista aplicando combine-elt-lst sobre la segunda lista

#### Código

```
;;;;
;;; combine-lst-lst
;;; Calcula el producto cartesiano de dos listas
```

```
(defun combine-lst-lst (lst1 lst2)
  (cond ((or (null lst1) (null lst2))
        nil)
        (t (mapcan #'(lambda(x) (combine-elt-lst x lst2)) lst1))))
```

## Comentarios

De igual manera que en el 3.1 aplicamos mapcan y vamos creando todos los pares de combinaciones de las dos listas.

## Ejemplos

```

CL-USER> (combine-lst-lst '(a b c) '(1 2))
((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))
CL-USER> (combine-lst-lst '(a b c) nil)
NIL
CL-USER> (combine-lst-lst nil nil)
NIL
CL-USER> (combine-lst-lst nil '(a b c))
NIL
CL-USER>

```

## Apartado 3.3

### Combine-list-of-lsts

(utilizamos dos funciones auxiliares iguales que las explicadas en el 3.1 y 3.2 pero en lugar de construir los pares con *list* los construimos con *cons*)

#### Pseudocódigo

**Entrada:** lstolsts: lista de listas

**Salida:** lista con todas las posibles combinaciones de elementos

#### Procesamiento:

Si la lista de listas es null

Nil

En otro caso, llamamos a la función que combina dos listas, cuyos argumentos serán la primera lista y la recursión de la función pero respecto al *rest* de la lista de listas.

#### Código

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts
;;; Calcula todas las posibles disposiciones de elementos
;;; pertenecientes a N listas de forma que en cada disposicion
;;; aparezca unicamente un elemento de cada lista
(defun combine-list-of-lsts (lstolsts)
  (cond ((null lstolsts)
        (list nil))
        (t (combine-cons-lst-lst (first lstolsts)
                                   (combine-list-of-lsts (rest lstolsts))))))

```

#### Comentarios

En este caso hemos utilizado la recursión para poder ir combinando todas las listas a partir de la función que da todas las combinaciones de dos listas.

#### Ejemplos

```

110 CL-USER> (combine-list-of-lsts '())
111 NIL
112 CL-USER> (combine-list-of-lsts '((a b c) (1 2 3 4)))
113 NIL
114 CL-USER> (combine-list-of-lsts '((a b c) (1 2 3 4) ()))
115 NIL
116 CL-USER> (combine-list-of-lsts '((a b c) (1 2 3 4) ()))
117 NIL
118 CL-USER> (combine-list-of-lsts '((1 2 3 4)))
119 ((1) (2) (3) (4))
120 CL-USER> (combine-list-of-lsts '(nil))
121 NIL
122 CL-USER> (combine-list-of-lsts nil)
123 NIL
124 CL-USER>

```