Memoria Práctica I

Andrés Salas Peña (andres.salas@estudiante.uam.es)
Ricardo Riol González (ricardo.riol@estudiante.uam.es)

Grupo 2301 - Pareja 8

Inteligencia Artificial
Universidad Autónoma

12 de febrero de 2018

Introducción

Esta primera práctica ha sido un primer acercamiento al lenguaje de programación LISP, un lenguaje muy distinto a los que habíamos visto hasta ahora. La programación funcional, nos abre muchas posibilidades a la hora de implementar algoritmos, ya que en pocas líneas de código se obtienen resultados realmente útiles.

1. Similitud Coseno

1.1 Sc-Rec y Sc-Mapcar

Pseudocódigo

Entrada:
x: vector representado como lista
y: vector representado como lista
Salida:
Similitud coseno entre x e y
Procesamiento:
Si alguna lista es nil devuelve nil, en caso contrario calcula $\frac{x*y}{\|x\|_2\|y\|_2}$

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 1.1.1
;;; sc-rec (x y)
;;; Calcula la similitud coseno de un vector de forma recursiva
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: similitud coseno entre x e y
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun is-ok (x y)
  (cond ((or (equal nil x) (equal nil y)) nil)
        ((or (every #'zerop x) (every #'zerop y)) nil)
        (t t)))

(is-ok '(1 2 3) '(3 4 5)) ;; t
(is-ok '(1 2 3) '(3 4 -5)) ;; nil
(is-ok '(0 0 0) '(1 0 0));; nil
(is-ok '(1 0 0) '());; nil

;;Calcula el producto escalar de dos vectores representados como listas
```

```
(defun our-pesc-rec (x y)
  (if (or (equal nil x) (equal nil y)) ;; Si x o y es null devolvemos 0
      0
    ;; Calculamos sum ( x(i) * y(i))
    (+ (* (first x) (first y)) (our-pesc-rec (rest x) (rest y)))))
    con 1 < i < long x

(our-pesc-rec '(1 2 3) '( 2 -5 6))

(defun sc-rec (lista1 lista2)
  ;;Comprobamos que a lista cumple las condiciones del enunciado
  (if (equal NIL (is-ok lista1 lista2))
      NIL
    (/ (our-pesc-rec lista1 lista2) (*(sqrt (our-pesc-rec lista1 lista1))
    (sqrt (our-pesc-rec lista2 lista2))))))
;; El producto escalar de un vector consigo mismo es su norma al cuadrado

(sc-rec '(1 0) '(0 1)) ;; 0.0
(sc-rec '() '(0 1)) ;; nil
(sc-rec '(1 2 3) '(1 0 0))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 1.1.2
;;; sc-mapcar (x y)
;;; Calcula la similitud coseno de un vector usando mapcar
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: similitud coseno entre x e y
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun our-pesc-map (lista1 lista2)
  (reduce '+ (mapcar #'* lista1 lista2))) ;; Sum (x(i) * y(i)) com 1< i <len x

;; Utilizamos las funciones recursivas del primer apartado.
(defun sc-mapcar (lista1 lista2)
  (if (equal NIL (is-ok lista1 lista2))
      nil
    (/ (our-pesc-map lista1 lista2) (*(sqrt (our-pesc-map lista1 lista1))
    (sqrt (our-pesc-map lista2 lista2))))))

(sc-mapcar '(1 2 3) '(1 0 0))
```

Comentarios

- En la función sc-rec se realiza una implementación basada en la definición recursiva, para ello hacemos uso de dos funciones auxiliares, is-ok y our-pesc-rec:
  - La primera comprueba si se puede calcular la similitud coseno desechando si alguno de los vectores es nil o está compuesto por ceros.
  - La segunda calcula el producto escalar de los dos vectores pasados.
  - Tras esto, sc-rec solo llama a la primera para ver si se cumplen las condiciones del enunciado y a la segunda varias veces acorde con la fórmula $\frac{x*y}{\|x\|_2\|y\|_2}$
- En la función sc-rec se realiza una implementación basada en el uso del mapcar. La función principal es igual que la anterior y llama también al mismo is-ok pero llama a our-pesc-map que calcula el producto escalar basándose en el mapcar.

1.2 Sc-Conf

Pseudocódigo

Entrada:
x: vector representado como lista
vs: vector de vectores representado como lista de listas
conf: nivel de confianza
Salida:
Vectores ordenados con similitud mayor que confianza
Procesamiento:
Si la similitud de una lista es menor que la confianza pedida se rechaza
Si no, se añade a la lista y se ordenan de mayor a menor

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 1.2
;;;     sc-conf (x vs conf)
;;; Devuelve aquellos vectores similares a una categoria
;;;
;;; INPUT: x: vector, representado como una lista
;;; vs: vector de vectores, representado como una lista de listas
;;; conf: Nivel de confianza
;;;
;;; OUTPUT: Vectores cuya similitud es superior al nivel de confianza, ordenados
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun our-conf (x vs conf)
  ;; Eliminamos de vs las listas cuyo cos con x sea menor que una constante dada
```

```
  (remove-if #'(lambda (y) (< (sc-mapcar x y) conf)) vs))

(our-conf '(1 2 3) '((1 2 3) (2 3 4) (1 0 0)) 0.5)
(sc-mapcar '(1 2 3) '(1 0 0))

(defun sc-conf (x vs conf)
   ;; Ordenamos de mayor a menor el vector en funcion de su cos con la lista x.
  (sort (our-conf x vs conf) #'(lambda (z y) (> (sc-mapcar x z) (sc-mapcar x y)))))

(sc-conf '(1 2 3) '((1 2 3) (3 4 5) (1 0 0) (1 1 1)) 0.9)
```

Comentarios

- En la función sc-conf realizamos la ordenación de mayor a menor de los vectores en función de su similarity-cos, para descartar aquellos que tienen un similarity-cos menor que el nivel de confianza usamos la función auxiliar our-conf:
    - La función our-conf simplemente elimina a través de una función lambda aquellos vectores cuyo valor sea menor que el nivel de confianza.

1.3 Sc-Classifier

Pseudocódigo

Entrada:
cats: vector de vectores representado como lista de listas
vs: vector de vectores representado como lista de listas
func: función que evaluará la similitud coseno
Salida:
Pares identificadores de categoría con el resultado de similitud coseno
Procesamiento:
Calcula cada similitud coseno de cada vs con todas las cats y guarda los pares identificador similitud coseno en una lista y los ordena según su similitud coseno para después quedarse con el mayor y finalmente se añaden a la lista.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 1.3
;;; sc-classifier (cats texts func)
;;; Clasifica a los textos en categorias.
;;;
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;; vs: vector de vectores, representado como una lista de listas
;;; func: referencia a funcion para evaluar la similitud coseno
;;;
;;; OUTPUT: Pares identificador de categoria con resultado de similitud coseno
```

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

;; Forma una lista de pares de la forma (vector , cos <lista y>) con y
;; perteneciente a cats
(defun our-similarity-cos (cats lista func)
  (mapcar #'(lambda (y) (append(list(first y) (funcall func (rest lista) (rest y)))

(our-similarity-cos '((1 2 3) ( 2 3 4) (6 6 8)) '(3 3 3) #'sc-rec)

;; Ordena los pares en funcion de la segunda coordenada.
(defun our-max-similarity (cats lista func)
  (first (sort(our-similarity-cos cats lista func)
  #'(lambda (z y) (> (second z) (second y))))))

(our-max-similarity '((1 2 3) ( 2 3 5) (6 6 8)) '( 3 6 8) #'sc-rec)

;;Aplica las funciones anteriores a un conjunto de vectores.
(defun sc-classifier (cats texts func)
  (mapcar #'(lambda (z) (our-max-similarity cats z func)) texts))


(sc-classifier '((1 2 3) (2 3 5) (3 6 8)) '((1 3 5) (2 6 8)) #'sc-rec)
(sc-classifier '((1 2 3) (2 3 5) (3 6 8)) '((1 3 5) (2 3 6) (3 2 3)) #'sc-rec)
(sc-classifier '((1 2 3) (2 3 5) (3 6 8)) '((1 3 5) (2 6 8)) #'sc-mapcar)
(sc-classifier '((1 2 3) (2 3 5) (3 6 8)) '((1 3 5) (2 3 6) (3 2 3)) #'sc-mapcar)
(sc-classifier '((1 43 23 12) (2 33 54 24)) '((1 3 22 134) (2 43 26 58)) #'sc-rec)
```

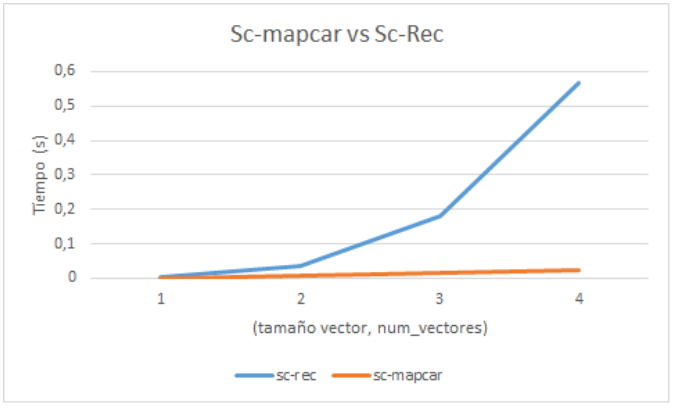
Comentarios

- Para la implementación de esta función nos hemos apoyado en un par de llamadas a otras funciones de tal forma que esta función principal solo se encarga de llamar a las otras tantas veces como vectores en texts tengamos:
  - La función our-max-similarity ordena una lista calculada en our-similarity-cos de mayor a menor según los segundos elementos de las sublistas y se queda con la primera de las sublistas.
    - La función our-similarity-cos realiza mediante un mapcar una lista de pares de la forma (vector, similitud-coseno de cada elemento de la lista)

1.4 Tiempos

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Estudio de tiempos
(time (sc-classifier '((1 2 3 4 5 6 7 3 4 4 4 4 4 4 5)
(2 3 5 5 6 7 8 2 2 2 2 2 3 4 5) (3 6 8 6 7 7 7 1 2 1 2 3 5 6 7))
```

```
'((1 3 5 2 2 2 2 6 8 4 2 1 9 9 9) (2 6 8 4 3 4 6 1 3 4 6 7 2 2 2)) #'sc-rec))
;;real time  0.002000 sec
(time (sc-classifier '((1 2 3 4 5 6 7 3 4 4 4 4 4 4 5)
(2 3 5 5 6 7 8 2 2 2 2 2 3 4 5) (3 6 8 6 7 7 7 1 2 1 2 3 5 6 7))
'((1 3 5 2 2 2 2 6 8 4 2 1 9 9 9) (2 6 8 4 3 4 6 1 3 4 6 7 2 2 2)) #'sc-mapcar))
;;real time  0.000000 sec
(time (sc-classifier (make-list 8 :initial-element
(make-list 20 :initial-element 1)) (make-list 8 :initial-element
(make-list 20 :initial-element 2)) #'sc-rec))
;;real time  0.035000 sec
(time (sc-classifier (make-list 8 :initial-element (make-list 20
:initial-element 1)) (make-list 8 :initial-element (make-list 20
:initial-element 2)) #'sc-mapcar))
;;real time  0.006000 sec
(time (sc-classifier (make-list 12 :initial-element (make-list 50
:initial-element 1)) (make-list 12 :initial-element (make-list 50
:initial-element 2)) #'sc-rec))
;;real time  0.182000 sec
(time (sc-classifier (make-list 12 :initial-element (make-list 50
:initial-element 1)) (make-list 12 :initial-element (make-list 50
:initial-element 2)) #'sc-mapcar))
; real time  0.014000 sec
(time (sc-classifier (make-list 15 :initial-element (make-list 100
:initial-element 1)) (make-list 15 :initial-element (make-list 100
:initial-element 2)) #'sc-rec))
;;real time  0.568000 sec
(time (sc-classifier (make-list 15 :initial-element (make-list 100
:initial-element 1)) (make-list 15 :initial-element (make-list 100
:initial-element 2)) #'sc-mapcar))
;; real time  0.025000 sec
```



En la gráfica anterior definimos los siguientes números que se corresponden con un par (tamaño del vector, número de vectores)
1 := (15,4); 2 := (20,8); 3 := (50,10); 4 := (100,15);
Como se puede observar en la gráfica la utilización de mapcar es más eficiente que la acudir a la recursión. Esto se debe a que la recursión es cotosa si nos es de cola.

Comentarios

2. Raíces de una Función

2.1 Bisect

Pseudocódigo

Entrada:
f: función cuyas raíces queremos encontrar
a: extremo inferior del intervalo en el que buscamos
b: extremo superior del intervalo en el que buscamos
tol: tolerancia que define el criterio de parada
Salida:
Devuelve $\frac{a+b}{2}$ como solución.
Procesamiento:
Subdivide en dos el intervalo y se queda con el que al aplicar $f(inf) * f(sup) < 0$ y lo hace recursivamente.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 2.1
;;; bisect (f a b tol)
;;; Encuentra una raiz de f entre los puntos a y b usando biseccion
;;;
;;; Si f(a)f(b)>0 no hay garantia de que vaya a haber una raiz en el
;;; intervalo, y la funcion devolvera NIL.
;;;
;;; INPUT: f: funcion de un solo parametro real con valores reales cuya
;;; raiz queremos encontrar
;;; a: extremo inferior del intervalo en el que queremos buscar la raiz
;;; b: b>a extremo superior del intervalo en el que queremos buscar la raiz
;;; tol: tolerancia para el criterio de parada: si b-a < tol de la funcion
;;;
;;; OUTPUT: devuelve (a+b)/2 como solucion
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun our-distance (a b) ;;Calcula la distancia entre dos reales
```

```
  (abs (- b a)))

(defun our-medium-point (a b) ;;Calcula el punto medio entre dos extremos reales
  (/ (+ a b) 2))

 ;; Aplica el algoritmo de la biseccion para encontrar soluciones de f (f(a) = 0).
 (defun bisect (f a b tol)
  (let ((c (our-medium-point a b))) ;; Definimos c = punto medio de (a , b)
    (cond ((equal (funcall f a) 0) a) ;; Si f(a) = 0 -> a
          ((equal (funcall f b) 0) b) ;; Si f(b) = 0 -> b
          ;; Si f(a) * f(b) > 0 -> nil (ambas son positvas o negativas)
          ((> (* (funcall f a) (funcall f b)) 0) nil)
          ;; Si el tamanyo de (a , b) es menor que tol -> punto medio (a , b)
          ((< (our-distance a b) tol) c)
          ((= (funcall f c) 0) c) ;; Si f(c) = 0 -> c
          ;; Aplicamos la recursividad en uno de los dos intervalos
          ((< (* (funcall f a) (funcall f c)) 0) (bisect f a c tol))
          ((< (* (funcall f b) (funcall f c)) 0) (bisect f c b tol)))))

(bisect #'(lambda (z) (* z z z)) -4 5 0.5)
(bisect #'(lambda(x) (sin (* 6.26 x))) 0.0 0.7 0.001)
(bisect #'(lambda(x) (sin (* 6.28 x))) 1.1 1.5 0.001)
(bisect #'(lambda(x) (sin (* 6.28 x))) 1.1 2.1 0.001)
```

Comentarios

- En este apartado usamos la función principal para ir desechando todos los casos de error y especiales:
  - Si $f(a) = 0$ devuelve a.
  - Si $f(b) = 0$ devuelve b.
  - Si $f(a) * f(b) > 0$ devuelve nil porque o ambas son positivas o ambas negativas.
  - Condición de parada, si el tamaño del intervalo (a b) ¡tol se devuelve el punto medio.
  - Si $f(c) = 0$ devuelve c, hemos encontrado la raíz.

  Si no se cumple ninguno de estos casos, usamos la recursividad para llamar al subintervalo que siga cumpliendo la condición de $f(a) * f(b) < 0$.
- Usamos dos funciones auxiliares: una our-distance que simplemente calcula la distancia entre los extremos de los intervalos; y otra, our-medium-point que calcula el punto medio de un intervalo.

2.2 Allroot

Pseudocódigo

Entrada:
f: función cuyas raíces queremos encontrar

lst: lista ordenada de valores reales
tol: tolerancia que define el criterio de parada

Salida:

Devuelve lista con valores que contienen las raíces en los subintervalos dados

Procesamiento:

Si la lista tiene un solo elemento o ninguno devuelve nil, si no llama al método bisectriz con cada par de valores de la lista ordenados y elimina los que no tienen solución (nil).

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 2.2
;;; allroot (f lst tol)
;;; Encuentra todas las raices localizadas entre dos valores consecutivos
;;; de una lista de valores
;;;
;;; INPUT: f: funcion de un solo parametro real con valores reales cuya
;;; raiz queremos encontrar
;;; lst: lista ordenada de valores reales (lst[i] < lst[i+1])
;;; tol: tolerancia para el criterio de parada: si b-a < tol de la funcion
;;;
;;; OUTPUT: una lista o valores reales que contienen las raices de la funcion
;;; en los subintervalos dados.
;;;
;;; Cuando sgn(f(lst[i])) != sgn(f(lst[i+1])) esta funcion busca
;;; una raiz en el correspondiente intervalo
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun allroot (f lst tol)
  ;; Caso base: Si la lista no tiene elementos o solo 1.
  (if (or (equal lst nil) (equal (rest lst) nil))
      nil
    ;;Lo utilizamos para eliminar los nil de la lista
    (mapcan #'(lambda (x) (unless (null x) (list x)))
    (append (list (bisect f (first lst) (second lst) tol))  ;; lista de soluciones
            (allroot f (rest lst) tol)))))

(allroot #'(lambda(x) (sin (* 6.28 x))) '(0.25 0.75 1.25 1.75 2.25) 0.0001)
(allroot #'(lambda(x) (sin (* 6.28 x))) '(0.25 0.9 0.75 1.25 1.75 2.25) 0.0001)
```

Comentarios

- Esta función simplemente comprueba que ningún elemento de la lista es nil y si no lo son, hacemos un mapcan para filtrar y eliminar aquellos valores nil que puede devolver la función

bisectriz y se llama a la función bisección con los dos primeros valores de la lista recursivamente.

2.3 Allind

Pseudocódigo

Entrada:
f: función cuyas raíces queremos encontrar
a: extremo inferior del intervalo en el que buscamos
b: extremo superior del intervalo en el que buscamos
N: exponente del número de intervalos en el que [a, b] será dividido (2^N)
tol: tolerancia que define el criterio de parada
Salida:
Devuelve $\frac{a+b}{2}$ como solución.
Procesamiento:
Subdividimos en 2^N intervalos y los introducimos en la lista junto con el extremo superior b y llamamos a la función del apartado anterior.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 2.3
;;; allind (f a b N tol)
;;;     Divide en un numero 2^N de invervalos y encuentra todas las raices
;;; de la funcion f en los intervalos obtenidos
;;;
;;; INPUT: f: funcion de un solo parametro real con valores reales cuya
;;; raiz queremos encontrar
;;; a: extremo inferior del intervalo en el que buscamos la raiz
;;; b: b>a extremo superior del intervalo en el que queremos buscar la raiz
;;; N: exponente del numero de intervalos en el que [a, b] va a ser dividido
;;; [a, b] es dividido en 2^N intervalos
;;; tol: tolerancia para el criterio de parada: si b-a < tol de la funcion
;;;
;;; OUTPUT: devuelve (a+b)/2 como solucion
;;;
;;; El intervalo (a, b) es dividido en intervalos (x[i], x[i+1]) con
;;; x[i] = a + i*dlt; una raiz es buscada en cada intervalo, y todas las
;;; raices encontradas se juntan en una lista que se devuelve
;;;
;;; Pista:
;;; Uno puede encontrar una manera de usar allroot para implementar esta funcion.
;;; Esto es posible por supuesto, pero hay una forma simple de hacerlo recursivo
;;; sin usar allroot.
;;;
```

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

(defun div-int (a b N) ;; Dividimos un intervalo (a , b) en 2^ns intervalos
  (let ((c (our-medium-point a b))) ;;Definimos c = punto intermedio
    (if (equal N 0) ;;Caso base: N = 0 -> 1 intervalo
        (list a)
      ;; dividimos cada intervalo por la mita hasta que n = 0
      (append (div-int a c (- N 1)) (div-int c b (- N 1))))))

;; La funcion anterior nos devuelve una lista con todos las divisiones del interval

(div-int 0 8 3)

(defun allind (f a b N tol)
  ;;Anyadimos b y calculamos todas las soluciones con la funcion del apartado 2.2
  (allroot f (append (div-int a b N) (list b)) tol))

(allind #'(lambda(x) (sin (* 6.28 x))) 0 3 2 0.01) ;
(allind #'(lambda(x) (sin (* 6.28 x))) 0.1 2.25 3 0.0001)
(allind #'(lambda(x) (sin (* 6.28 x))) 0.25 2.25 3 0.01) ;
```

Comentarios

- La función principal solo se encarga de añadir a la lista el extremo superior b y llama a la función auxiliar div-int.
- La función div-int llama a la función del punto medio del apartado anterior y va añadiendo a la lista ordenadamente cada extremo hasta llegar a completar los 2^N intervalos.

3. Combinación de Listas

3.1 Combine-elt-lst

Pseudocódigo

Entrada:
elem: elemento que combinaremos con los de la lista
lst: lista con la que se combina el elemento
Salida:
Devuelve la lista con las combinaciones.
Procesamiento:
Si la lista está vacía devolvemos nil, si no se van añadiendo combinaciones recursivamente.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 3.1
```

```
;;; combine-elt-lst (elem lst)
;;; Combina un elemento dado con todos los elementos de una lista
;;;
;;; INPUT: elt: elemento que se combinara con los de la lista
;;; lst: lista con la que se combinara el elemento
;;;
;;; OUTPUT: devuelve la lista con las combinaciones del elemento y la lista dadas.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun combine-elem-lst (elem lst)
  ;; Con esta definicion no repetimos codigo
  (let ((k (list(list elem (first lst)))))
  (cond ((equal lst nil) nil) ;; Caso base: lst nil -> nil
        ;; Caso base: si la lista tiene un elemento -> k
        ((equal (rest lst) nil) k)
        ( t (append k (combine-elem-lst elem (rest lst)))))))

(combine-elem-lst 'a nil)
(combine-elem-lst 'a '(1))
```

Comentarios

- La función comprueba:
  - Si la lista está vacía devuelve nil.
  - Si la lista tiene solo un elemento lo devuelve.
  - Si la lista posee varios elementos añade el primero y llama recursivamente a la misma función con el resto de la lista.

3.2 Combine-lst-lst

Pseudocódigo

Entrada:
lst1: primera lista del producto cartesiano
lst2: segunda lista del producto cartesiano
Salida:
Devuelve la lista resultado del producto cartesiano.
Procesamiento:
Devuelve nil si ambas listas están vacías, si no se llama a la función anterior con cada elemento de la primera lista y la segunda lista entera.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 3.2
```

```
;;; combine-lst-lst (lst1 lst2)
;;; Calcula el producto cartesiano de dos listas
;;;
;;; INPUT: lst1: primera lista sobre la que se realizara el producto cartesiano
;;; lst2: segunda lista sobre la que se realizara el producto cartesiano
;;;
;;; OUTPUT: devuelve la lista resultado del producto cartesiano de las anteriores
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun combine-lst-lst (lst1 lst2)
  ;; Caso base: Si ambas lista son nil -> nil nil
  (if (or (equal lst1 nil) (equal lst2 nil))
    ;; Combinamos el elemento primero de la primera lista con la segunda
    ;; y aplicamos recursividad sobre la lst1
    (append (combine-elem-lst (first lst1) lst2) (combine-lst-lst (rest lst1) lst2)

(combine-lst-lst nil nil)
(combine-lst-lst '(a b c) nil)
(combine-lst-lst NIL '(a b c))
(combine-lst-lst '(a b c) '(1 2))
(combine-lst-lst '(a b c d e f) '(1 2))
```

Comentarios

- Comprueba si alguna lista es nil y si no lo son, llama a la función del apartado anterior para el primer elemento de la lista 1 y por recursividad a esta propia función pasándole como primer argumento el resto de la lista.

3.3 Combine-list-of-lsts

Pseudocódigo

Entrada:
lstolsts: todas las listas a combinar
Salida:
Devuelve la lista resultado de la combinación de todas las dadas.
Procesamiento:
Devuelve nil si la lista o alguna de sus sublistas está vacía, si la lista solo tiene un vector cada uno de sus elementos se convierten en listas independientes y s i no se devuelve la combinación de todas las listas dadas.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 3.3
;;; combine-list-of-lsts (lstolsts)
```

```
;;; Calcula todas las posibles disposiciones pertenecientes a N listas
;;; de forma que en cada disposicion aparezca solo un elemento de cada lista
;;;
;;; INPUT: lstolsts: todas las listas que se combinaran
;;;
;;; OUTPUT: devuelve una lista resultado de la combinacion de todas las dadas
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun flatten (lst)  ;; Funcion que sirve para quitar parentesis a las listas
  (cond
   ((null lst) NIL)
   ((atom (first lst))
    (cons (first lst) (flatten (rest lst))))
   (t (append (flatten (first lst)) (flatten (rest lst))))))

(defun cl (lstolsts)
  (if (equal lstolsts nil) ;; Si lstolsts es nil devolvemos (nil)
      (return-from cl nil))
  (let ((comb (cl (rest lstolsts)))) ;;Definimos (cl (rest lstoslts)) = comb
    ;;Si ha finalizado de recorrer la lista de listas devolvemos todo lo anterior
    (if (eql comb nil)
       (first lstolsts)
      (combine-lst-lst (first lstolsts) comb)))) ;; LLamada recursiva

(defun combine-list-of-lsts (lstolsts)
  (cond ((equal lstolsts nil) '(nil)) ;; Si el vector de listas es nil -> nil
        ;; Si alguna lista de vector es nil -> nil
        ((some #'(lambda (z) (equal z nil)) lstolsts) nil)
        ;; Si el vector tiene solo una lista
        ;; -> Convertimos cada elemento en una lista independiente
        ((equal (rest lstolsts) nil) (mapcar #'list (first lstolsts)))
        ;; En caso contrario, llamamos a la funcion recursciva cl
        ( t (mapcar #'(lambda (x) (flatten x)) (cl lstolsts)))))

(combine-list-of-lsts '((a b c) (+ -) (1 2 3 4) (m o r e)))
(combine-list-of-lsts '(() (+ -) (1 2 3 4)))
(combine-list-of-lsts '((a b c) () (1 2 3 4)))
(combine-list-of-lsts '((a b c) (1 2 3 4) ()))
(combine-list-of-lsts '((1 2 3 4)))
(combine-list-of-lsts '((a b c) (+ -) (1 2 3 4)))
```

Comentarios

- La función principal contempla todos los casos posibles:
  - Si el vector de listas es nil devuelve nil.

    - Si alguna lista de vectores es nil devuelve nil.
    - Si el vector tiene una sola lista, convierte cada elemento suyo en una lista.
    - Si no está en ningún caso anterior llama a la función auxiliar recursiva cl y le aplica otra función auxiliar flatten.
- La función cl comprueba si la lista pasada es nil y si no recorre la lista de listas de manera recursiva hasta que termina y devuelve todas las combinaciones.
- La función flatten se usa para quitar los paréntesis a las listas que devuelve la función recursiva anterior, con el objetivo de aplanar y simplificar las expresión.

Conclusiones

En esta primera entrega parcial, hemos encontrado muchísimas dificultades al no estar familiarizados con el lenguaje de LISP ni con todas las funciones internas que este posee ni tampoco con el uso adecuado de los paréntesis.

Por tanto, le hemos dedicado muchísimo tiempo para ir entendiendo la lógica de Lisp y la profundidad y potencia del lenguaje. También queremos reseñar que no ha ayudado para nada el cambiar los entornos en los que trabajar con tan poco tiempo, el tener la primera clase de prácticas solo 3 días antes de la entrega y el cambio del enunciado tres veces durante los últimos dos/tres días previos a la entrega parcial.

Como es bien sabido, los inicios siempre cuestan, ahora solo falta irse acostumbrando cada vez más al lenguaje de cara a la entrega final.