Inteligencia Artificial
Entrega parcial práctica 1

Martín Selgas, Blanca (blanca.martins@estudiante.uam.es)
Villar Gómez, Fernando (fernando.villarg@estudiante.uam.es)
Grupo 2302 - Pareja 3

10/02/2018

Índice

1. Introducción

En el presente documento se detalla la realización de los tres primeros problemas pertenecientes a la primera práctica de Inteligencia Artificial. En ellos se requiere la resolución de determinadas tareas en el lenguaje funciónal LISP, aportando el pseudo-código, la codificación, la documentación y los comentarios de las funciónes utilizadas para tales fines.

El desarrollo de la práctica se ha realizado en la aplicación **Allegro CL 6.2 ANSI** corriendo sobre un sistema Windows 10, pero el código utilizado es compatible con otros entornos LISP como **SBCL**.

2. Ejercicio 1

2.1. Apartado 1.1

Para calcular la similitud coseno entre dos vectores necesitamos realizar varias operaciones, que se pueden resumir en tres:

- **Producto escalar**, en concreto tres de ellos: $x \cdot y$, $x \cdot x$ y $y \cdot y$.
- **Raíz cuadrada**, función nativa en LISP: `sqrt`
- **División**, funcón nativa en LISP: `/`

Por tanto es necesario codificar el producto escalar. Tal y como se dice en el enunciado hay dos formas diferentes de implementarlo: a través de recursión o a través de una función `mapcar`.

Producto escalar usando recursión

PSEUDOCÓDIGO

```
Entrada: x (vector)
         y (vector)
Salida:  x·y (producto escalar)

Procesamiento:
         Si x o y están vacíos,
             evalúa a 0
         en caso contrario,
             evalúa a x[0]·y[0] + producto_escalar(x[1:n],y[1:n])
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; prod-esc-rec (x y)
;;; Calcula el producto escalar de dos vectores de forma recursiva
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: producto escalar entre x e y
;;;
(defun prod-esc-rec (x y)
   (if (or (null x) (null y))
         0
      (+ (* (first x) (first y))
         (prod-esc-rec (rest x) (rest y)))))
;;;
;;; EJEMPLOS:
;;; (prod-esc-rec '() '()) ;-> 0 ; caso base
;;; (prod-esc-rec '(1 2) '(3 4)) ;-> 11 ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Producto escalar usando mapcar

PSEUDOCÓDIGO

```
Entrada: x (vector)
         y (vector)
Salida:  x·y (producto escalar)

Procesamiento:
         inicializa ret=0
         Para i desde 1 a n,
             ret <= ret + x[i]·y[i]
         evalúa a ret
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; prod-esc-mapcar (x y)
;;; Calcula el producto escalar de dos vectores usando mapcar
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: producto escalar entre x e y
;;;
(defun prod-esc-mapcar (x y)
   (apply #'+ (mapcar #'* x y)))
;;;
;;; EJEMPLOS:
;;; (prod-esc-mapcar '() '()) ;-> 0 ; caso base
;;; (prod-esc-mapcar '(1 2) '(3 1)) ;-> 5 ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Además, entre los requisitos para el correcto funciónamiento del programa para calcular la similitud coseno se encuentra la condición de que se retorne `NIL` en caso de que se introduzcan dos vectores cero como argumentos. Por tanto necesitamos crear otra función que realice esta tarea:

Comprobación de vectores cero

PSEUDOCÓDIGO

```
Entrada: x (vector)
Salida:  t si x es vector cero o vacío, nil en caso contrario

Procesamiento:
         Si x está vacío,
             evalúa a t
         en caso contrario,
             evalúa a x[0]==0 and comprobar_cero(x[1:n])
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; check-zero (x)
;;; Comprueba si x es el vector cero
;;;
;;; INPUT: x: vector, representado como una lista
;;;
;;; OUTPUT: t si x es el vector cero o nil, nil en caso contrario
;;;
(defun check-zero (x)
   (if (null x)
        t
     (and (= (first x) 0)
          (check-zero (rest x)))))
```

```
;;;
;;; EJEMPLOS:
;;; (check-zero '()) ;-> t ; caso base
;;; (check-zero '(0 0 0)) ;-> t ; caso tipico
;;; (check-zero '(1 0 0)) ;-> nil ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Una vez tenemos estas funciónes podemos codificar la función principal para el cálculo de la similitud coseno, con comprobación de errores incluida.

Cálculo de similitud coseno

PSEUDOCÓDIGO

Entrada: x (vector)
y (vector)

Salida: nil si se introducen argumentos no permitidos,
$cos\_similarity(x,y) = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$ en caso contrario

Procesamiento:
Si x o y están vacíos o son cero,
evalúa a nil
en caso contrario,
evalúa a prod_esc(x,y) entre raíz(prod_esc(x,x))*raíz(prod_esc(y,y))

CÓDIGO 1 (recursión)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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 sc-rec (x y)
   (unless (or (check-zero x) (check-zero y))
      (/ (prod-esc-rec x y)
         (* (sqrt (prod-esc-rec x x))
            (sqrt (prod-esc-rec y y))))))
;;;
;;; EJEMPLOS:
;;; (sc-rec '() '()) ;-> nil ; caso no permitido
;;; (sc-rec '(0 0) '(0 0)) ;-> nil ; caso no permitido
;;; (sc-rec '(1 2) '(2 3)) ;-> 0.99227786 ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

CÓDIGO 2 (mapcar)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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 sc-mapcar (x y)
   (unless (or (check-zero x) (check-zero y))
      (/ (prod-esc-mapcar x y)
```

```
        (* (sqrt (prod-esc-mapcar x x))
           (sqrt (prod-esc-mapcar y y))))))
;;;
;;; EJEMPLOS:
;;; (sc-rec '() '()) ;-> nil ; caso no permitido
;;; (sc-rec '(0 0) '(0 0)) ;-> nil ; caso no permitido
;;; (sc-rec '(1 2) '(2 4)) ;-> 1.0 ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- En la comprobación de la función `check-zero` ya se comprueba si se ha introducido un vector vacío (y devuelve `t` en tal caso), por lo que no es necesario incluir las cláusulas `(null x)` o `(null y)`.
- No hay ahorro evidente entre ambas funciónes por el uso de recursión o iteratividad, ya que en cualquiera de los casos se tienen que procesar todas las componentes de los vectores sin excepción.

2.2. Apartado 1.2

Para la implementación de la función `sc-conf` requerida en este apartado se han codificado dos funciones auxiliares, para gestionar diccionarios (con pares vector-similitud) en los que se puedan ordenar los vectores según su similitud.

Creación del diccionario

PSEUDOCÓDIGO

```
Entrada: x (vector)
         vs (lista de vectores)
Salida:  diccionario con pares (vector, similitud)

Procesamiento:
         Para cada elemento de vs,
             calcula cos_similarity(x,elemento)
             añade (cos_similarity(x,elemento), elemento) a lst
         evalúa a lst
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf-gendict (x vs)
;;; Genera un diccionario con la similitud coseno entre un vector y todos los vectores
;;; de una lista de vectores
;;;
;;; INPUT: x: vector, representado como una lista
;;; vs: vector de vectores, representado como una lista de listas
;;;
;;; OUTPUT: diccionario formado por las similitudes y los vectores
;;;
(defun sc-conf-gendict (x vs)
   (mapcar #'(lambda (y) (list (sc-rec x y) y)) vs))
;;;
;;; EJEMPLOS:
;;; (sc-conf-gendict '(1 2) '((1 2) (1 3))) ;-> ((1.0 (1 2)) (0.98994946 (1 3)))
;;; (sc-conf-gendict '(2 3) '((1 1) (1 2))) ;-> ((0.9805807 (1 1)) (0.99227786 (1 2)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Posteriormente, se realizarían las modificaciones pertinentes a este diccionario según los requisitos, en este caso habría que eliminar todos los vectores con una similitud inferior al argumento `conf` y posteriormente ordenarlos de mayor a menor (más adelante se mostrará con qué implementación de las funciones `sort` y `remove`). Finalmente, para obtener la salida que buscamos se usa la siguiente función.

Descomposición del diccionario

PSEUDOCÓDIGO

Entrada: *dic* (diccionario)
Salida: lista de vectores ordenados del diccionario

Procesamiento:
Para cada elemento de dic,
obtener elemento[1], vector correspondiente
añade elemento[1] a lst
evalúa a lst

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf-selvec (lst)
;;; De un diccionario obtiene el vector correspondiente a cada similitud
;;;
;;; INPUT: lst: diccionario de similitudes-vectores
;;;
;;; OUTPUT: lista de vectores procedentes del diccionario en el mismo orden
;;;
(defun sc-conf-selvec (lst)
   (mapcar #'(lambda (y) (first (second y))) lst))
;;;
;;; EJEMPLOS:
;;; (sc-conf-selvec '((1.0 (1 2)) (0.98994946 (1 3)))) ;-> ((1 2) (1 3))
;;; (sc-conf-selvec '((0.9805807 (1 1)) (0.99227786 (1 2)))) ;-> ((1 1) (1 2))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Y finalmente la función que implementa todo el proceso:

Obtención de vectores ordenados por similitud mayores a un nivel de confianza

PSEUDOCÓDIGO

Entrada: *cat* (categoría)
vs (lista de vectores)
conf (nivel de confianza)
Salida: lista de vectores ordenados con similitud mayor a conf

Procesamiento:
Generar diccionario de pares (cos_similarity(cat,x), x) con x de vs
eliminar pares con cos_similarity(cat,x)<conf
ordenar diccionario de mayor a menor según cos_similarity(cat,x)
descomponer diccionario y obtener lst, lista de vectores
evalúa a lst

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf (cat vs conf)
;;; Devuelve aquellos vectores similares a una categoria
;;;
;;; INPUT: cat: vector que representa a una categoria, 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 sc-conf (cat vs conf)
   (sc-conf-selvec
      (sort
         (remove conf (sc-conf-gendict cat vs) :test #'> :key #'first)
         #'> :key #'first)))
```

```
;;;
;;; EJEMPLOS:
;;; (sc-conf '(1 2) '((1 2) (1 3) (1 4) (2 30)) 0.95) ;-> ((1 2) (1 3) (1 4))
;;; (sc-conf '(2 5) '((1 5) (4 10)) 0.9) ;-> ((4 10) (1 5))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- `sort` es una función destructiva, pero en este caso no es importante mantener la lista previa (la desordenada) puesto que sólo estamos interesados en la ordenada. Si quisiéramos mantener la lista previa bastaría con aplicar un `copy-list` a la salida de la instrucción `remove`.
- La utilización de las funciones crear/descomponer diccionario viene justificada por la necesidad de ordenar una serie de vectores en base a un parámetro que no se encuentra en los mismos vectores, sino que se calcula a partir de ellos y es un valor externo.
- Otra posible solución sería crear una lista con las similitudes de los vectores paralelamente a la propia lista de vectores, y posteriormente realizar las mismas operaciones de reordenación en la lista de vectores a medida que se va ordenando la lista de similitudes. Se ha considerado más simple y entendible la opción del uso de diccionarios.

2.3. Apartado 1.3

En este apartado nos piden una tarea similar a la del apartado anterior, pero en esta ocasión sólo tenemos que devolver la categoría de mayor similitud por cada vector. Además, aquí también se incluyen identificaciones por cada vector y categoría, y se puede introducir por argumento qué función usar para calcular la similitud coseno; por lo tanto la implementación de los diccionarios usados en el apartado 1.2 ha de cambiar necesariamente. Las nuevas funciones son las siguientes:

Creación del diccionario

PSEUDOCÓDIGO

```
Entrada: x (vector)
         vs (lista de vectores)
         func (función para cálculo de cos_similarity)
Salida:  diccionario con pares (similitud, identificador)

Procesamiento:
         Para cada elemento de vs,
             calcula cos_similarity(x[1:n],elemento[1:n])
             añade (cos_similarity(x[1:n],elemento[1:n]), elemento[0]) a lst
         evalúa a lst
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-class-gendict (x vs func)
;;; Realiza la misma funcion que sc-conf-gendict pero ahora tenemos en cuenta
;;; que los vectores tienen un identificador, y la funcion para calcular la
;;; similitud se pasa como parametro
;;;
;;; INPUT: x: vector, representado como una lista
;;; vs: vector de vectores, representado como una lista de listas
;;; func: funcion con la que calcular la similitud
;;;
;;; OUTPUT: diccionario formado por las similitudes y los identificadores de los vectores
;;;
(defun sc-class-gendict (x vs func)
   (mapcar #'(lambda (y) (list (funcall func (rest x) (rest y)) (car y))) vs))
;;;
;;; EJEMPLOS:
```

```
;;; (sc-class-gendict '(1 1 2) '((1 1 2) (2 1 3)) #'sc-rec) ;-> ((1.0 1) (0.98994946 2))
;;; (sc-class-gendict '(1 2 3) '((1 1 1) (2 1 2)) #'sc-mapcar) ;-> ((0.9805 1) (0.9922 2))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Descomposición del diccionario

PSEUDOCÓDIGO

Entrada: *dic* (diccionario)
Salida: lista de vectores ordenados del diccionario

Procesamiento:
Para cada elemento de dic,
obtener elemento[0], identificador, y elemento[1], similitud
añade (elemento[1] . elemento[0]) a lst
evalúa a lst

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-class-selvec (lst)
;;; Realiza la misma funcion que sc-conf-selvec pero ahora en vez de devolver el
;;; vector, devuelve su identificacion junto con la similitud en un par
;;;
;;; INPUT: lst: diccionario de similitudes-vectores
;;;
;;; OUTPUT: lista de pares con identificadores de vectores y similitudes
;;;
(defun sc-class-selvec (lst)
   (mapcar #'(lambda (y) (cons (second y) (first y))) lst))
;;;
;;; EJEMPLOS:
;;; (sc-class-selvec '((1.0 1) (0.98994946 2))) ;-> ((1 . 1.0) (2 . 0.98994946))
;;; (sc-class-selvec '((0.9805807 1) (0.99227786 2))) ;-> ((1 . 0.9805807) (2 . 0.99227786))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Por tanto, nuestro programa final debe realizar lo mismo que el `sc-conf` del apartado anterior, pero sin usar el comando `remove` y seleccionando únicamente el primer elemento de la lista que nos devuelva `sc-class-selvec`, puesto que será el de mayor similitud. Puesto que vamos a tener que usar un `mapcar` para cada texto, nos ayudaremos de una función intermedia que realice la ordenación y gestione los diccionarios.

Obtención de categorías ordenadas por similitud para un texto

PSEUDOCÓDIGO

Entrada: x (categoría)
vs (lista de textos)
$func$ (función con la que obtener cos_similarity)
Salida: pares similitud-identificador ordenados

Procesamiento:
Generar diccionario con pares similitud-identificador
ordenar diccionario según similitud para cada identificador
descomponer diccionario y devolver pares (identificador . similitud)

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf-alt (x vs func)
;;; Devuelve lista de pares identificador-similitud ordenados por similitud a uno dado
;;;
;;; INPUT: x: vector, representado como una lista
;;; vs: vector de vectores, representado como una lista de listas
;;; func: funcion para calcular similitud
```

```
;;; OUTPUT: Pares identificador-similitud ordenados
;;;
(defun sc-conf-alt (x vs func)
   (sc-class-selvec
      (sort (sc-class-gendict x vs func) #'> :key #'first)))
;;;
;;; EJEMPLOS:
;;; (sc-conf-alt '(2 1 2) '((1 1 2) (2 1 3) (3 1 4) (4 2 30)) #'sc-rec) ;-> ((1 . 1.0) (2
. 0.98994946) (3 . 0.97618705) (4 . 0.9221943))
;;; (sc-conf-alt '(3 2 5) '((1 1 5) (2 4 10)) #'sc-mapcar) ;-> ((2 . 1.0) (1 . 0.98328197))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Y finalmente nuestra función principal:

Obtención de categoría de máxima similitud para lista de textos

PSEUDOCÓDIGO

```
Entrada: cats (lista de categorías)
         texts (lista de textos)
         func (función con la que obtener cos_similarity)
Salida:  pares identificador-similitud máximos para cada texto

Procesamiento:
         Para cada texto en texts,
             obtener lista de pares identificador-similitud de cada categoría ordenados
             seleccionar pares[0], el máximo
             añadir pares[0] a lst
         evalúa a lst
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-classifier (cats texts func)
;;; Clasifica a los textos en categorias.
;;;
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;; texts: vector de vectores, representado como una lista de listas
;;; func: función para evaluar la similitud coseno
;;; OUTPUT: Pares identificador de categoría con resultado de similitud coseno
;;;
(defun sc-classifier (cats texts func)
   (mapcar #'(lambda (y) (car (sc-conf-alt y cats func))) texts))
;;;
;;; EJEMPLOS:
;;; (sc-classifier '((1 1 2) (2 5 10)) '((1 1 3) (2 1 4) (3 2 5)) #'sc-rec) ;-> ((1 . 0.98994946)
(2 . 0.9761871) (1 . 0.9965458))
;;; (sc-classifier '((1 43 23 12) (2 33 54 24)) '((1 3 22 134) (2 43 26 58)) #'sc-mapcar)
;-> ((2 . 0.48981872) (1 . 0.81555086))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Mismo comentario respecto al tratamiento de los diccionarios que en el apartado anterior, pero además en este caso teníamos que incluir la similitud en el resultado final, por lo que hemos considerado esta opción como la más factible.
- Se ha codificado `sc-conf-alt` como si el primer argumento `x` fuera una categoría, pero luego en `sc-classifier` se ha iterado sobre los textos como primer argumento. Esto se ha realizado por comodidad al definir las funciones, y el resultado final no varía dado que $cos\_similarity(x, y) = cos\_similarity(y, x)$ y es indistinto, por tanto, tratar texto como categoría o al revés.

2.4. Apartado 1.4

Método | Dimensión	3	4	5	6
Recursión	32580	17169	38199	36240
Mapcar	39612	26774	48076	50721

La tabla superior expone los ciclos de reloj de CPU que se han utilizado para aplicar la función `sc-classifier` a vectores de dimensión 3, 4, 5 y 6. En general el número de ciclos para los mismos argumentos puede variar por diversos motivos (como fallos de caché), pero se puede apreciar que hay una tendencia a que la versión iterativa de `mapcar` consuma más ciclos de reloj.

Este hecho puede venir debido a que desde un punto de vista de número de operaciones, conviene más realizar llamadas recursivas que la creación y el manejo de estructuras internas que utilice `mapcar` para funcionar como funciona. Además, es más compatible con la naturaleza del lenguaje funcional, por lo que podríamos también pensar que por cómo funciona LISP se favorezcan más las codificaciones iterativas.

3. Ejercicio 2

A lo largo del Ejercicio 2 se emplea la función `clean`, cuya motivación es el filtrado de las listas formadas únicamente por elementos que son NIL.

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; clean (lst)
;;;
;;; Función no destructiva que devuelve una lista vacía si todos
;;; los elementos de lst son NIL.
;;;
;;; INPUT:
;;;
;;;   lst: lista que se quiere evaluar.
;;;
;;; OUTPUT:
;;;
;;;   NIL si todos los elementos de lst son NIL;
;;;   lst en caso contrario.
;;;
(defun clean (lst)
   (unless (every #'null lst)
      lst))
;;;
;;; EJEMPLOS:
;;;
;;; (clean '(NIL NIL NIL)) ;->NIL
;;; (clean '(NIL 1 NIL)) ;->(NIL 1 NIL)
;;; (clean '(1 2 3)) ;->(1 2 3)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

3.1. Apartado 2.1

Método de la bisección

PSEUDOCÓDIGO

Entrada: f (función)
a (número real)
b (número real)

```
            tol (número real)
    Salida: nil si no se han encontrado raíces,
            la raíz encontrada en caso contrario.

    Procesamiento:
             Si f(a)*f(b)<0
                 Si (b-a)<tol
                     evalúa a (a+b)/2
                 en caso contrario
                     Si f(a)*f((a+b)/2)>=0
                         evalúa a la bisección entre (a+b)/2 y b con tolerancia tol
                     en caso contrario
                         evalúa a la bisección entre a y (a+b)/2 con tolerancia tol
              en caso contrario
                 evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; bisect (f a b tol)
;;;
;;; Encuentra una raíz de f entre los puntos a y b utilizando bisección.
;;;
;;; Si f(a)f(b)>=0, no hay garantía de que exista una raíz en el
;;; intervalo y la función devuelve NIL.
;;;
;;; INPUT:
;;;
;;;   f: función real de un solo parámetro con valores reales
;;;      de la que queremos encontrar la raíz.
;;;   a: límite inferior del intervalo en el que queremos buscar la raíz.
;;;   b: b>a límite superior del intervalo en el que queremos buscar la raíz.
;;;   tol: tolerancia para el criterio de parada: si b-a <tol la función
;;;      devuelve (a+b)/2 como solución.
;;;
;;; OUTPUT:
;;;
;;;   raíz de la función, o NIL si no se ha encontrado ninguna.
;;;
(defun bisect (f a b tol)
   (let ((pto-medio (/ (+ a b) 2)))
      (unless (>= (* (funcall f a) (funcall f b)) 0)
         (if (<(- b a) tol)
               pto-medio
            (if (>= (* (funcall f a) (funcall f pto-medio)) 0)
                  (bisect f pto-medio b tol)
               (bisect f a pto-medio tol))))))
;;;
;;; EJEMPLOS:
;;;
;;; (bisect #'(lambda(x) (sin (* 6.26 x))) 0.0 0.7 0.001) ;-> NIL
;;; (bisect #'(lambda(x) (sin (* 6.26 x))) 0.1 0.7 0.001) ;-> 0.5016602
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Se ha empleado la definición recursiva:

```
bisect(f a b tol) = bisect(f (b+a)/2 b tol) [Si f(a)*f((b+a)/2)>=0]
```

```
bisect(f a b tol) = bisect(f a (b+a)/2 tol) [Si f((b+a)/2)*f(b)>0]
```

- No se ha escogido la definición del método:

```
While (b-a)<tol
    c = (b-a)/2
    Si f(a)*f(c)>=0
        a = c
    en caso contrario
        b = c
```

  pues requiere uso de variables y bucles y, por lo tanto, no es funcional. Por ello, se ha optado por una implementación recursiva.

3.2. Apartado 2.2

Función allroot

PSEUDOCÓDIGO

```
Entrada: f (función)
         lst (lista)
         tol (número real)
Salida:  nil si no se han encontrado raíces,
        lista de raíces encontradas en caso contrario.

Procesamiento:
         Si length(lst)!= 1
             evalúa a una lista compuesta por
               bisección entre los dos primeros elementos de la lista
               allroot de la lista sin el primer elemento
         en caso contrario
             evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; allroot (f lst tol)
;;;
;;; Encuentra todas las raíces situadas entre valores consecutivos
;;; de una lista ordenada.
;;;
;;; Siempre que sgn(f(lst[i])) != sgn(f(lst[i+1])) la función buscará
;;; una raíz en el correspondiente intervalo.
;;;
;;; INPUT:
;;;
;;;   f: función real de un solo parametro con valores reales
;;;      de la que queremos encontrar la raíz.
;;;   lst: lista ordenada de valores reales (lst[i] <lst[i+1]).
;;;   tol: tolerancia para el criterio de parada: si b-a <tol la función
;;;      devuelve (a+b)/2 como solución.
;;;
;;; OUTPUT:
;;;
;;;   Una lista de valores reales conteniendo las raíces de la
;;;   función en los sub-intervalos dados.
;;;
(defun allroot-aux (f lst tol)
   (unless (and (null (rest lst)) (not (null (first lst))))
      (cons (bisect f (first lst) (second lst) tol)
```

```
            (allroot-aux f (rest lst) tol))))

(defun allroot (f lst tol)
   (clean (allroot-aux f lst tol)))
;;;
;;; EJEMPLOS:
;;;
;;; (allroot #'(lambda(x) (sin (* 6.28 x))) '(0.1 2.25) 0.0001) ;-> NIL
;;; (allroot #'(lambda(x) (sin (* 6.28 x))) '(0.25 0.75 1.25 1.75 2.25) 0.0001)
;;; ;-> (0.50027466 1.0005188 1.5007629 2.001007)
;;;
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- `allroot` se basa en la idea de concatenar el resultado del método de la bisección del intervalo definido por los dos primeros números de la lista, y el de repetir el proceso con el resto de la lista. Siguiendo el mismo esquema que en la bisección, por definición el algoritmo *allroot* sería:

```
ret[]
for i := 0 TO length(lst)-1
    ret[i] = bisección entre lst[i],lst[i+1]
```

  pero se ha descartado por el uso de variables y bucles.

- La función auxiliar `clean` permite evitar salidas del tipo `(NIL [..] NIL)` de forma no destructiva. Sin embargo, incluirla en la función recursiva `allroot` supondría posibles comportamientos indeseados (eliminación de un conjunto de NIL cuando más adelante se iba a insertar un número en la lista), por lo que se ha sacrificado la recursión de cola en `allroot`:

```
(defun allroot (f lst tol)
    (unless (and (null (rest lst)) (not (null (first lst))))
        (cons (bisect f (first lst) (second lst) tol)
            (allroot f (rest lst) tol))))
```

3.3. Apartado 2.3

Función allind

PSEUDOCÓDIGO

```
Entrada: f (función)
         a (número real)
         b (número real)
         N (número real)
         tol (número real)
Salida:  nil si no se han encontrado raíces,
        lista de raíces encontradas en caso contrario.

Procesamiento:
         c = a + (b-a)/(2^N)
         Si c <b
             evalúa a una lista compuesta por
               bisección entre a y c
               allind desde c en adelante
```

CÓDIGO

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

```
;;; allind(f a b N tol)
;;;
;;; Divide un intervalo en cierto numero de sub-intervalos y encuentra
;;; todas las raíces de la función f en los mismos.
;;;
;;; El intervalo [a,b] es dividido en intervalos [x[i],x[i+1]] con
;;; x[i] = a + i*dlt; en cada intervalo se busca una raíz, y todas
;;; las raíces encontradas son devueltas en una lista.
;;;
;;; INPUT:
;;;
;;;   f: función real de un solo parámetro con valores reales
;;;      de la que queremos encontrar la raíz.
;;;   a: límite inferior del intervalo en el que queremos buscar la raíz.
;;;   b: b>a límite superior del intervalo en el que queremos buscar la raíz.
;;;   N: Exponente del número de intervalos en el que se divide [a,b]:
;;;      [a,b] se divide en 2^N intervalos
;;;   tol: tolerancia para el criterio de parada: si b-a <tol la función
;;;      devuelve (a+b)/2 como solución.
;;;
;;; OUTPUT:
;;;
;;;      Lista con todas las raíces encontradas.
;;;
(defun allind-aux (f x incr tol max)
   (let ((y (+ x incr)))
      (unless (>y max)
         (cons (bisect f x y tol) (allind-aux f y incr tol max)))))

(defun allind (f a b N tol)
   (clean (allind-aux f a (/ (- b a) (expt 2 N)) tol b)))
;;;
;;; EJEMPLOS:
;;;
;;; (allind #'(lambda(x) (sin (* 6.28 x))) 0.1 2.25 1 0.0001) ;-> NIL
;;; (allind #'(lambda(x) (sin (* 6.28 x))) 0.1 2.25 2 0.0001)
;;; ;-> (0.50027084 1.0005027 1.5007347 2.0010324);
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- La definición de *allind* corresponde al siguiente pseudocódigo:

```
ret[]
incr = a + (b-a)/(2^N)
for i := 0 TO (2^N)-1
    ret[i] = bisección entre a y (a + incr)
    a = a + incr
```

  Sin embargo, debido al uso de variables y bucles, en nuestro caso se ha realizado una implementación funcional que aprovecha la recursión.

- La función `allind` con recursión de cola realiza el cómputo de la exponencial en cada llamada recursiva, por lo que se ha decidido descartar esta idea, y así aprovechar además la oportunidad para llamar a la función auxiliar `clean` de manera segura.

4. Ejercicio 3

4.1. Apartado 3.1

PSEUDOCÓDIGO

Entrada: *elt* (átomo)
lst (lista)
Salida: lista con todas las posibles combinaciones.

Procesamiento:
Si lst no vacía
evalúa a una lista compuesta por:
sublista conteniendo elt y el primer elemento de lst
resultado de combine-elt-lst(elt , lst sin el primer elemento)
en caso contrario
evalúa a NIL

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst (elt lst)
;;; Genera una lista con las combinaciones de un elemento con los elementos de una lista
;;;
;;; INPUT: elt: elemento a combinar
;;; lst: lista de elementos a combinar
;;;
;;; OUTPUT: Lista con todas las posibles combinaciones
;;;
(defun combine-elt-lst (elt lst)
   (unless (null lst)
      (cons (list elt (first lst))
            (combine-elt-lst elt (rest lst)))))
;;;
;;; EJEMPLOS:
;;; (combine-elt-lst 'a nil) ;-> nil ; caso base
;;; (combine-elt-lst nil '(1 2)) ;-> ((NIL 1) (NIL 2)) ; caso atípico
;;; (combine-elt-lst 'a '(1 2)) ;-> ((A 1) (A 2)) ; caso típico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Por la simpleza de esta función se ha decidido usar la recursión en lugar de un método iterativo. En el fondo el número de operaciones es el mismo puesto que hace falta inevitablemente recorrer toda la lista `lst`, pero se ha elegido este método que es más natural de la programación funcional.
- Importante considerar el caso base en el que devolvemos un `nil` en caso de recibir una lista vacía, dado que es necesario para que posteriormente, en la recursión, la unión de una lista con un `nil` sea la propia lista, y a partir de ahí se empiece a construir.
- Si se recibe un `nil` como primer argumento la función sencillamente tratará a ese `nil` como un elemento cualquiera y realizará la misma acción, como se puede apreciar en el ejemplo.

4.2. Apartado 3.2

PSEUDOCÓDIGO

Entrada: *lst*1 (lista)
*lst*2 (lista)
Salida: lista con todas las posibles combinaciones.

Procesamiento:

```
            Si lst1 y lst2 distintas de NIL
                evalúa a una lista compuesta por:
                  combine-elt-lst(primer elemento de lst1 con lst2)
                  combine-lst-lst(lst1 sin el primer elemento con lst2)
            en caso contrario
                evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst (lst1 lst2)
;;; Genera el producto cartesiano lst1 X lst2 (todas las combinaciones de
;;; elementos entre ambas listas)
;;;
;;; INPUT: lst1: primera lista de elementos a combinar
;;; lst2: segunda lista de elementos a combinar
;;;
;;; OUTPUT: Lista con todas las posibles combinaciones
;;;
(defun combine-lst-lst (lst1 lst2)
   (unless (or (null lst1) (null lst2))
      (append (combine-elt-lst (first lst1) lst2)
              (combine-lst-lst (rest lst1) lst2))))
;;;
;;; EJEMPLOS:
;;; (combine-lst-lst '() '()) ;-> nil ; caso no permitido
;;; (combine-lst-lst '() '(1 2)) ;-> nil ; caso base
;;; (combine-lst-lst '(1 2) '(a b)) ;-> ((1 A) (1 B) (2 A) (2 B)) ; caso típico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Al igual que en el apartado anterior y por los mismos motivos se ha elegido una implementación recursiva en lugar de iterativa.
- La comprobación `(null lst2)` es, en realidad, innecesaria, puesto que se realizaría igualmente en la llamada a la función `combine-elt-lst`, por lo que se podría eliminar. Se ha mantenido por motivos de eficiencia, dado que será menos costoso para la máquina realizar una comprobación sobre `lst2` que realizar modificaciones en la pila de funciones con más llamadas.
- El hecho de que siempre se realice la comprobación `(null lst1)` hace que nunca se vaya a llamar a la función `combine-elt-lst` con primer argumento `nil`; por tanto, pierde importancia el tratamiento que se le dio a dicho caso, dado que siempre que no se llame a la función individualmente, nunca va a darse.
- A efectos de codificación, el caso no permitido y el caso base contenidos en los ejemplos no son diferentes; sin embargo, para la naturaleza del algoritmo sí lo son, puesto que pasar dos listas vacías es claramente un caso no permitido pero pasar la primera lista vacía puede ser uno de los pasos de la recursión cuando se haya terminado de llamar con todos los elementos de `lst1` y `(rest lst1)` esté vacía.

4.3. Apartado 3.3

Dado que venimos de implementar dos funciones que combinan un elemento con una lista y una lista con una lista, para realizar la tarea de combinar entre sí todos los elementos de una lista de listas lo lógico es pensar en una recursión que aplique el siguiente pseudocódigo:

```
combinacion_listas(lista-de-listas):
  combinacion_lista_lista(lista-de-listas[0],
                          combinacion_listas(lista-de-listas[1:n]))
```

Sin embargo, con las funciones anteriormente codificadas no se puede hacer esta tarea, dado que tal y como están programadas, en uno de los pasos intermedios de la recursión podríamos encontrarnos con la necesidad de combinar, por ejemplo, la lista `(1 2)` con la lista de listas `((a +) (a -) (b +) (b-))`, cuyo resultado deseado sería `((1 a +) (1 a -) (1 b +) (1 b -) (2 a +)...)`; pero lo que realmente devolvería dicha función en este paso sería `((1 (a +)) (1 (a -)) ...)`. Por tanto surge la necesidad de programar funciones específicas que traten con listas de listas, que explicamos a continuación:

Combinación elemento-lista de listas

PSEUDOCÓDIGO

Entrada: *elt* (átomo)
lol (lista de listas)
Salida: lista de combinaciones

Procesamiento:
Para cada lista de lol,
añadir elt al principio de lista
añadir lista a ret
evalúa a ret

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lol (elt lol)
;;; Genera una lista con las combinaciones de un elemento con los elementos de una
;;; lista de listas
;;;
;;; INPUT: elt: elemento a combinar
;;; lol: lista de listas de elementos a combinar
;;;
;;; OUTPUT: Lista con todas las posibles combinaciones
;;;
(defun combine-elt-lol (elt lol)
   (mapcar #'(lambda (y) (cons elt y)) lol))
;;;
;;; EJEMPLOS:
;;; (combine-elt-lol á nil) ;-> nil ; caso no permitido
;;; (combine-elt-lol á '((1 2) (3 4))) ;-> ((A 1 2) (A 3 4)) ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- A diferencia de las funciones implementadas en los apartados anteriores, se elige aquí una implementación iterativa a través de `mapcar` en lugar de utilizar la recursión, que daría exactamente el mismo resultado con una codificación siguiendo el siguiente pseudocódigo:

```
combine-elt-lol(elt lol):
     si lol está vacío,
       devolver lista vacía
     en caso contrario,
       añadir elt al principio de lol[0],
       devolver unión de lol[0] y combine-elt-lol(elt lol[1:n])
```

- El caso de pasar `nil` como segundo argumento viene contemplado en los ejemplos. ¿Qué ocurre si usamos `nil` como primer argumento? Lo que ocurre es que se trata a `nil` como un elemento más, y por ejemplo, al combinarlo con `((1 2) (3 4))` surgiría `((NIL 1 2) (NIL 3 4))`.

Combinación lista-lista de listas

PSEUDOCÓDIGO

Entrada: *lst* (lista)
lol (lista de listas)
Salida: lista de combinaciones

```
Procesamiento:
         Para cada elemento de lst,
             combine-elt-lol(elemento lol)
             se añade la combinación a ret
         evalúa a ret
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lol (lst lol)
;;; Genera el producto cartesiano de una lista con todas las listas de una
;;; lista de listas
;;;
;;; INPUT: lst: lista de elementos a combinar
;;; lol: lista de listas de elementos a combinar
;;;
;;; OUTPUT: Lista con todas las posibles combinaciones
;;;
(defun combine-lst-lol (lst lol)
   (mapcan #'(lambda (z) (combine-elt-lol z lol)) lst))
;;;
;;; EJEMPLOS:
;;; (combine-lst-lol nil '((1 2) (3 4))) ;-> nil ; caso no permitido
;;; (combine-lst-lol '(1 2) nil) ;-> nil ; caso no permitido
;;; (combine-lst-lol '(a b) '((1 2) (3 4))) ;-> ((A 1 2) (A 3 4) (B 1 2) (B 3 4))
;;; ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Al igual que en la función anterior se ha elegido el método iterativo para la implementación, aunque existe su versión recursiva, similar a la utilizada en la función del apartado 3.2; sin embargo, dado que no hay diferencia de operaciones porque en cualquier caso hay que iterar sobre todos los elementos de todas las listas, no hay un beneficio muy cuantioso en utilizar la recursión, y sin embargo la versión iterativa es mucho más simple de codificar.
- Importante la utilización de `mapcan` en lugar de `mapcar`. En este caso, usamos `mapcan` para que todas las combinaciones se encuentren en una misma lista. Si usáramos `mapcar`, la salida del último ejemplo expuesto en el código sería `(((A 1 2) (A 3 4)) ((B 1 2) (B 3 4)))`, que no es lo que buscamos.

Con estas dos funciones definidas, ahora sí que podemos plantear la recursión mencionada anteriormente para terminar de construir nuestro combinador de listas:

Combinación de un número arbitrario de listas

PSEUDOCÓDIGO

```
Entrada: lstolsts (lista de listas)
Salida:  lista de combinaciones

Procesamiento:
         Si lstolsts está vacío,
             devuelve ()
         en caso contrario,
             juntar combine-lst-lol (lstolsts[0]) con combine-lstolsts(lstolsts[1:n])
             devolver lista con la unión anterior
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts (lstolsts)
;;; Genera una lista con todas las disposiciones de elementos pertenecientes a listas
```

```
;;; contenidas en una lista de listas
;;;
;;; INPUT: lstolsts: lista de listas de la que sacar todas las combinaciones
;;;
;;; OUTPUT: Lista con todas las posibles combinaciones
;;;
(defun combine-list-of-lsts (lstolsts)
   (if (null lstolsts)
         '(nil)
      (append (combine-lst-lol (first lstolsts)
                               (combine-list-of-lsts (rest lstolsts))))))
;;;
;;; EJEMPLOS:
;;; (combine-list-of-lsts '()) ;-> (nil) ; caso base
;;; (combine-list-of-lsts '(() (+ -) (1 2 3 4))) ;-> nil ; caso no permitido
;;; (combine-list-of-lsts '((a b c) () (1 2 3 4))) ;-> nil ; caso no permitido
;;; (combine-list-of-lsts '((a b c) (+ -) ())) ;-> nil ; caso no permitido
;;; (combine-list-of-lsts '((a b c))) ;-> ((a) (b) (c)) ; caso particular
;;; (combine-list-of-lsts '((a b c) (+ -) (1 2 3 4))) ;-> ; caso tipico
;;; ((A + 1) (A + 2) (A + 3) (A + 4) (A - 1) (A - 2) (A - 3) (A - 4)
;;; (B + 1) (B + 2) (B + 3) (B + 4) (B - 1) (B - 2) (B - 3) (B - 4)
;;; (C + 1) (C + 2) (C + 3) (C + 4) (C - 1) (C - 2) (C - 3) (C - 4))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- En esta función es extremadamente importante el caso base de la recursión. Gracias a que cuando se recibe una lista vacía se devuelve `(nil)` conseguimos que todos los pasos se realicen correctamente, ya que `(combine-lst-lol '(1 2 3) '(nil))`, por ejemplo, nos devuelve `((1) (2) (3))`, porque en LISP concatenar un `nil` detrás de un elemento cualquiera en una lista es como no hacer nada. A partir de esta primera lista de listas con un elemento es como se van generando todas las demás en los pasos consecutivos, dando lugar a la salida deseada.
- Cabe destacar también el hecho de que cuando una de las listas de `lstolsts` es vacía, el programa devuelve `nil`. Esto viene como consecuencia de que `combine-lst-lol` devuelve `nil` si `lst` está vacía, que es exactamente lo que ocurre en uno de los pasos de la recursión cuando se llega a la lista vacía dentro de `lstolsts`.