Ejercicio 1.1:

- **PSEUDOCÓDIGO**

Entrada: Dos vectores (x e y).

Salida: El coseno del ángulo formado entre los vectores x e y.

Función general:

Comprobar que los vectores no están vacíos y si lo son, devolver 0;

División entre:

Producto escalar (x y)

Raíz de multiplicación de:

Producto escalar (x x)

Producto escalar (y y)

Función producto escalar recursivo (a b):

Caso base: Los dos vectores están vacíos: devolver 0.

Comprobar condiciones de error

Multiplicar a[0] por b[0] y sumar el producto escalar de (resto de a, resto de b)

Función producto escalar por mapcar (a b):

Crea una nueva lista con la multiplicación a pares de los elementos de a y b

Suma todos los elementos

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; opera-con-error(x y op)
;;; Funcion auxiliar que opera comprobando errores
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;; op: operacion a realizar
;;;
;;; OUTPUT: resultado de operacion o NIL en caso de error
;;;
(defun opera-con-error (op x y)
  (unless (or (null x) (null y))
    (funcall op x y)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; caso-error(x y)
;;; Funcion auxiliar que comprueba los casos de error
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: T (algo esta mal) o NIL (todo correcto)
;;;
```

```
(defun caso-error (x y)
 (when (or (null x)
       (null y)
       (minusp (first x)) ;; Si alguno es negativo, devolver NIL
       (minusp (first y)))
        t))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; pe-rec (x y)
;;; Funcion auxiliar que calcula el producto escalar entre dos vectores
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: Producto escalar de ambos o NIL si hay algun error
;;;
(defun pe-rec (x y)
 (cond ((and (null x) (null y)) 0) ;; Si estan vacías, devolver cero (caso base)
      ((caso-error x y) NIL) ;; Comprobamos errores
      (t (opera-con-error #'+
                  (* (first x) (first y))
                  (pe-rec (rest x) (rest y)))))) ;; Recursion

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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)
 (if (or (null x) (null y)) ;; Comprobamos que no esten vacios para evitar divisiones entre 0.
    0
  (opera-con-error #'/
            (pe-rec x y)
            (sqrt (* (pe-rec x x)
                  (pe-rec y y))))))

;;; Pruebas
(sc-rec '() '())  ;; 0
(sc-rec '(0 1) '(1 1))   ;; 0.7071...
(sc-rec '(0 1) '(1 0))   ;; 1
(sc-rec '(0 1) '(0 1))   ;; 0

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; pe-mapcar (x y)
;;; Calcula el producto escalar usando mapcar
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: similitud coseno entre x e y
;;;
(defun pe-mapcar (x y)
 (apply #'+ (mapcar #'* x y)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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)
  (if (or (null x) (null y)) ;; Comprobamos que no esten vacios para evitar divisiones entre 0
      0
    (/ (pe-mapcar x y)
       (sqrt (* (pe-mapcar x x)
                (pe-mapcar y y))))))

;;; Pruebas
(sc-mapcar '() '())   ;; 0
(sc-mapcar '(0 1) '(1 1))   ;; 0.7071...
(sc-mapcar '(0 1) '(1 0))   ;; 0
(sc-mapcar '(0 1) '(0 1))   ;; 1
```

- **COMENTARIOS**

En nuestro caso hemos necesitado una operación llamada opera-con-error. Detecta que una operación puede provocar un error debido a los argumentos y devuelve NIL. De esta manera podemos arriesgarnos a tener operandos nulos y evitar errores de programa.

También hemos creado una función caso-error para separar la comprobación de errores y hacer el código más legible.

Ejercicio 1.2:

- **PSEUDOCÓDIGO**

Entrada: Un vector categoría, una lista de vectores a evaluar y un nivel de confianza

Salida: Vector de vectores ordenados de mayor a menor confianza, teniendo todos mayor confianza que el nivel establecido.

Función:

Obtener las confianzas de todos los vectores y eliminar los que no tengan confianza suficiente

Ordenar los vectores resultantes por confianza

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; filtrar-vectores (x vs conf)
;;; Funcion para seleccionar los vectores a analizar posteriormente
;;;
;;; INPUT: x: vector, representado como una lista
;;; vs: vector de vectores, representado como una lista de listas
;;; conf: Nivel de confianza
;;;
;;; OUTPUT: lista de tupas con la confianza y el vector al que pertenece
;;;
(defun filtrar-vectores (x vs conf)
  (mapcan (lambda (y)
          (let ((cos (sc-rec x y))) ;; coseno entre los vectores
            (when (opera-con-error #'>
                                   cos
                                   conf)
              (list (cons cos
                          (list y)))))) ;; Lista con el coseno y el vector asociado al mismo
      vs))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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 sc-conf (x vs conf)
  (mapcar #'second ;; Nos quedamos solo con el segundo
    (sort ;; De la lista ordenada de mayor a menor por el coseno
     (filtrar-vectores x vs conf) #'> :key #'first)))

;; Pruebas
(sc-conf '(1 0) '() 0.5) ;; NIL
(sc-conf '(1 0) '(()) 0.5) ;; NIL
(sc-conf '(1 0) '((0 1) (1 1) (1 0)) 0.5) ;; ((1 0) (1 1))
(sc-conf '(1 2 3) '((1 2 3) (1 2) (0 3 1) (0 0 1) (12 0 1)) 0.7) ;; ((1 2 3) (0 0 1) (0 3 1))
```

- **COMENTARIOS**

Para el filtrado de vectores hemos decidido crear una función lambda que cree una tupla con el vector y su confianza. En caso de que la confianza sea menor que la necesitada, eliminará ese vector del resultado.

Tras ordenar los vectores hemos necesitado quedarnos únicamente con los vectores, por ello usaremos la función second en cada tupla.

Ejercicio 1.3:

- **PSEUDOCÓDIGO**

Entrada: Un vector de vectores categoría, una lista de vectores a evaluar y una función a usar

Salida: Vector de tuplas, donde aparece cada vector y la categoría asignada

Función:

Para cada vector:

Calcular los cosenos con todas las categorías y representarlos como una tupla: Categoría y confianza

Obtener la tupla con mayor confianza

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; calcular-cosenos (cats text func)
;;; Funcion auxiliar para obtener todas las tuplas posibles
;;;
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;; vs: vector
;;; func: referencia a función para evaluar la similitud coseno
;;;
;;; OUTPUT: Pares identificador de categoría con resultado de similitud coseno
;;;
(defun calcular-cosenos (cats text func)
  (mapcar #'(lambda (x)
            (cons (first x)
                (funcall func (rest x) (rest text))))
    cats))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; coger-mayor (valores)
;;; Funcion auxiliar para obtener la mayor tupla
;;;
;;; INPUT: valores: lista de tuplas con los valores a analizar
;;;
;;; OUTPUT: tupla cuyo segundo elemento es el mayor de la lista
;;;
(defun coger-mayor (valores)
  (first (sort valores #'> :key #'rest)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-classifier (cats texts func)
;;; Clasifica a los textos en categorías.
;;;
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;; vs: vector de vectores, representado como una lista de listas
;;; func: referencia a función para evaluar la similitud coseno
;;; OUTPUT: Pares identificador de categoría con resultado de similitud coseno
;;;
(defun sc-classifier (cats texts func)
  (mapcan #'(lambda (x)
            (list
             (coger-mayor (calcular-cosenos cats x func))))
    texts))

;; Ejercicio 1.4
```

```
(setf cats '((1 43 23 12) (2 33 54 24)))
(setf texts '((1 3 22 134) (2 43 26 58)))
(sc-classifier cats texts #'sc-rec) ;;((2 . 0.48981872) (1 . 0.81555086))
(sc-classifier cats texts #'sc-mapcar) ;;((2 . 0.48981872) (1 . 0.81555086))

(setf cats '((1 43 23 12 1 5 3) (2 33 54 24 52 68 84)
        (3 2 3 4 5 6 7) (4 8 12 34 53 10 53)
        (5 1 1 1 1 1 1) (6 68 35 111 54 65 87)
        (7 95 84 75 87 95 76) (8 1 1 1 1 1 1)
        (9 32 64 15 97 68 2) (10 54 87 91 56 57 30)
        (11 64 97 84 62 35 45) (12 88 99 54 61 55 32)))
(setf texts '((1 3 22 134 53 65 75) (2 43 26 58 65 78 45)
         (3 4 6 5 12 34 15) (4 5 6 7 12 34 65)
         (5 5 24 87 91 448 35) (6 5 78 35 94 68 54)
         (7 95 84 75 87 95 76) (8 1 1 1 1 1 1)))
(time (sc-classifier cats texts #'sc-rec))
(time (sc-classifier cats texts #'sc-mapcar))
```

- **COMENTARIOS**

Al principio nos planteamos utilizar las funciones desarrolladas en el apartado anterior, pero no lo vimos viable debido a que, como quedan ordenados, dejamos de saber a qué categoría pertenece cada confianza. Por ello hemos tenido que crear una nueva función que sigue un esquema similar pero la forma de mostrar los datos es diferente.

Aprovechamos para explicar también el ejercicio 1.4:

Podemos apreciar una clara diferencia entre las dos implementaciones de la similitud coseno, siendo la recursiva mucho más lenta que la implementada mediante mapcar.

Tiempo de CPU de mapcar: 0.015625

Tiempo de CPU de recursivo: 0.109375

Pensamos que se debe a que la función mapcar paraleliza el trabajo, siendo mucho más eficiente en máquinas multithread.

Ejercicio 2.1:

Entrada: f (función), a, b (elementos entre los que realizar la bisección), tol (tolerancia)

Salida: La raíz encontrada de la función en el intervalo dado. NIL si no hay raíces (o no se pueden hallar por este método

- **PSEUDOCÓDIGO**

```
Def bisect (f, a, b, tol):
        If (signo(a) * signo(b) > 0):
                return NIL
        medio = (a + b) / 2
        if (es_raiz(a)):
                return a
        else if (es_raiz(b)):
                return b
        else if (supera_tolerancia(a, b, tol)):
                return medio
        else:
                if ( signo(a) * signo(medio) <= 0):
                        return bisect(f, a, medio, tol)
                else if (signo(medio) * signo(b) <= 0):
                        return bisect(f, medio, b, tol)
```

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Returns true if the tolerance has been reached
;;
;; a: lower extremum of the interval in which we search for the root
;; b: b>a upper extremum of the interval in which we search for the root
;; tol: tolerance for the stopping criterion: if b-a < tol the function
;; returns true if the tolerance has been reached.

(defun test (a b tol)
    (when (< (- b a) tol)
        T))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Finds a root of f between the points a and b using bisection.
;;
;; If f(a)f(b)>0 there is no guarantee that there will be a root in the
;; interval, and the function will return NIL.
;;
;; f: function of a single real parameter with real values whose root
;; we want to find
;; a: lower extremum of the interval in which we search for the root
;; b: b>a upper extremum of the interval in which we search for the root
;; tol: tolerance for the stopping criterion: if b-a < tol the function
;; returns (a+b)/2 as a solution.
;;

(defun bisect (f a b tol)
    (let ((med (/ (+ a b) 2)))
        (cond ((<= b a)
            nil)
        ((= 0 (funcall f a))
            a)
        ((= 0 (funcall f b))
```

```
                b)
        ((test a b tol)
                med)
        ((>= (* (funcall f a) (funcall f b)) 0)
                nil)
        ((<= (* (funcall f a) (funcall f med)) 0)
                (bisect f a med tol))
        ((<= (* (funcall f med) (funcall f b)) 0)
                (bisect f med b tol)))))

;; Use cases

(bisect #'(lambda (x) (sin (* 6.26 x))) 0.1 0.7 0.0001) ;; -> 0.50184333
(bisect #'(lambda (x) (sin (* 6.26 x))) 0.1 0.1 0.0001) ;; -> NIL
(bisect #'(lambda (x) (sin (* 6.26 x))) 0.2 0.1 0.0001) ;; -> NIL
(bisect #'(lambda (x) (sin (* 6.26 x))) 0.1 0.7 1) ;; -> 0.4
```

- **COMENTARIOS**

Para este apartado decidimos utilizar un cond debido a la cantidad de casos diferentes que había, tanto casos de error como posibles condiciones de parada. Aunque no lo especificase así el enunciado, decidimos tener en cuenta los extremos del intervalo como posibles soluciones, por lo que comprobamos si los parámetros a y b de la función son soluciones. También añadimos como caso de error aquél en el que b <= a, ya que en este caso no se aplica el algoritmo.

También decidimos implementar una función auxiliar que comprueba si se ha alcanzado la tolerancia (test).

Por lo demás, la función se basa en una simple recursión, por lo que no tuvimos que tomar más decisiones de diseño.

Ejercicio 2.2:

Entrada: f (función), lst (lista de puntos entre los que buscar raíces), tol (tolerancia)

Salida: Una lista con las raíces encontradas en el intervalo. NIL si no hay soluciones

- **PSEUDOCÓDIGO**

```
Def allroot (f, lst, tol):
        unless (rest(lst) = NIL):
                raíz = bisect(f, first(lst), second(lst), tol)
                append(raíz, allroot(f, rest(lst), tol)
```

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Finds all the roots of f between the elements of a list using bisection.
;;
;;
;; f: function of a single real parameter with real values whose root
;; we want to find
;; lst: a list of the values we want to explore
;; tol: tolerance for the stopping criterion: if b-a < tol the function
;; returns a list with the found roots
;;
(defun allroot (f lst tol)
        (unless (null (rest lst))
                (remove nil
                        (append
                                (list
                                        (bisect f
                                                (first lst)
                                                (second lst) tol))
                                (allroot f (rest lst) tol)))))

;; Use cases

(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)

(allroot #'(lambda(x) (sin (* 6.28 x))) NIL 0.0001)
;; -> NIL

(allroot #'(lambda(x) (sin (* 6.28 x))) '(0.25 2 1.25 1.75 2.25) 0.0001)
;; -> (1.5007629 2.001007)
;; En el caso de que b <= a, se devuelve NIL y no se incluye en la lista
```

- **COMENTARIOS**

En este apartado decidimos ir aplicando bisect a los puntos de la lista, de dos en dos, empezando por el principio. Para ello, cada iteración de la recursión realiza un append entre la salida del bisect de los dos primeros puntos de la lista, y la salida de allroot (recursivo) al que le pasamos rest(lst), en vez de la lista entera. De esta forma, cuando sólo quede un punto (rest(lst) = NIL) acabará la recursión. Para llevar a cabo dicha implementación, tuvimos que borrar los nil que se introducían en la lista en caso de no encontrarse soluciones.

Ejercicio 2.3

Entrada: f (función), a, b (puntos entre los que buscar soluciones), n (número de franjas en las que subdividir el intervalo), tol (tolerancia)

Salida: Una lista con las raíces encontradas en el intervalo. NIL si no hay soluciones.

- **PSEUDOCÓDIGO**

```
Def allind (f, a, b, n, tol):
    If (n = 0):
        return bisect (f, a, b, tol)
    else:
        half = (a + b) / 2
        left = allind (f, a, half, (n–1), tol)
        right = allind(f, half, b, (n-1), tol)
        return append(left, right)
```

- ***CÓDIGO***

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Divides the interval [a, b] into 2^n parts. Then searches for roots
;; in every part.
;;
;; f: function of a single real parameter with real values whose root
;; we want to find
;; a: lower extremum of the interval in which we search for the root
;; b: b>a upper extremum of the interval in which we search for the root
;; n: number of parts
;; tol: tolerance for the stopping criterion: if b-a < tol the function
;; returns a list with the found roots
;;

(defun allind (f a b n tol)
    (if (= n 0)
        (remove nil (list (bisect f a b tol)))
        (let ((half (/ (+ b a) 2)))
            (append
                (allind f a half (- n 1) tol)
                (allind f half b (- n 1) tol)))))

;; Use cases

(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)

(allind #'(lambda (x) (sin (* 6.28 x))) 0.1 0.1 2 0.0001)  ;; -> NIL
```

- ***COMENTARIOS***

La primera implementación que se nos vino a la cabeza fue la de crear una lista con los (n+1) puntos que conforman los n intervalos, y pasarle esa lista a *allroot.* Sin embargo, una forma más eficiente consistía en usar la recursividad en ambas mitades del intervalo, de

forma que, al entrar en la función por n-ésima vez, ya tendremos la lista dividida en 2^n partes. De este modo, vamos decrementando la n hasta llegar a cero, y en ese momento se calcula la bisección y se devuelve una lista con la solución, que va a ser introducida en una lista mediante un append.

En este caso nos encontramos dos problemas. El primero era que, al calcular la bisección, debíamos guardar el resultado en una lista para que funcionase el append. El segundo y más grave, era que en caso de no encontrar soluciones, obteníamos ((nil), (nil)). Una forma de arreglarlo fue usando un *remove nil* al devolver el resultado de la bisección para que, en caso de no haber soluciones, se devuelva *nil* en vez de *(nil)*.

Ejercicio 3.1:

- **PSEUDOCÓDIGO**

Entrada: Un elemento y una lista

Salida: Una lista de listas donde se realizan las combinaciones de ambos

Función:

Para todos los elementos de la lista

Combinar dicho elemento con el elemento suministrado

Combinación:

Si los dos elementos son átomos, crear una lista con ellos

Si son un elemento y una lista, crear una lista con el elemento suministrado y el elemento de la lista

Si son dos listas, concatenar ambas.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst (elt lst)
;;; Combina un elemento con todos los elementos de una lista.
;;;
;;; INPUT: elt: elemento a combinar
;;; lst: lista de elementos a combinar
;;;
;;; OUTPUT: lista de listas con todas las combinaciones
;;;
(defun combine-elt-lst (elt lst)
  (mapcar #'(lambda (x)
          (cond ((and (atom x) (atom elt)) ;; Elementos individuales
                 (list elt x))
                ((atom elt) ;; LST es lista de listas
                 (cons elt x))
                ((atom x) ;; ELT es una lista
                 (append elt (list x)))
                (t (append elt x)))) ;; Ambos son listas
    lst))

;; Pruebas
(combine-elt-lst 'a NIL) ;; NIL
(combine-elt-lst 1 '(1 2 3 4)) ;; ((1 1) (1 2) (1 3) (1 4))
(combine-elt-lst 'b '(a c)) ;; ((b a) (b c))
```

- **COMENTARIOS**

En nuestro caso hemos realizado esta implementación más completa para simplificar los apartados posteriores. Estuvimos dudosos de si la adición de un elemento a una lista debería realizarse con un append o dándole la vuelta a la lista, añadiéndolo al principio y volviéndole a dar la vuelta. Hemos optado por el append por simplicidad y una mejor comprensión del código.

Ejercicio 3.2:

- **PSEUDOCÓDIGO**

Entrada: Dos listas

Salida: Todas las posibles combinaciones de los elementos de ambas listas

Función:

Para cada elemento de una de las listas:

Combinarlo con la otra lista mediante la función del apartado anterior.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst (lst1 lst2)
;;; Realiza el producto cartesiano entre dos listas
;;;
;;; INPUT: lst1: lista 1
;;; lst2: lista 2
;;;
;;; OUTPUT: lista de listas con todas las combinaciones
;;;
(defun combine-lst-lst (lst1 lst2)
  (mapcan #'(lambda (x)
            (combine-elt-lst x lst2))
    lst1))

;; Pruebas
(combine-lst-lst nil nil) ;; --> NIL
(combine-lst-lst '(a b c) nil) ;; --> NIL
(combine-lst-lst NIL '(a b c)) ;; --> NIL
(combine-lst-lst '(a b c) '(1 2)) ;; --> ((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))
```

- **CÓMENTARIOS**

En este caso hemos utilizado la función del apartado anterior, que nos facilita la implementación.

Ejercicio 3.3:

- **PSEUDOCÓDIGO**

Entrada: Una lista de listas

Salida: Lista de listas con todas las posibles combinaciones de los elementos de entrada.

Función:

Comprobar que la lista no está vacía y devolver una lista vacía.

Si la lista contiene una sola lista, crear una lista donde cada elemento es una lista.

En el resto de casos, utilizar la función del apartado anterior de forma recursiva

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts-aux (lstolsts acc)
;;; Calcula todas las posibles combinaciones entre n listas
;;;
;;; INPUT: lstolsts: lista de listas a combinar
;;; acc: lista de listas acumulada
;;; OUTPUT: lista de listas con todas las combinaciones
;;;
(defun combine-list-of-lsts-aux (lstolsts acc)
  (if (null lstolsts) acc
    (combine-list-of-lsts-aux (rest lstolsts)
                              (combine-lst-lst acc
                                               (first lstolsts)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts (lstolsts)
;;; Calcula todas las posibles combinaciones entre n listas
;;;
;;; INPUT: lstolsts: lista de listas a combinar
;;;
;;; OUTPUT: lista de listas con todas las combinaciones
;;;
(defun combine-list-of-lsts (lstolsts)
  (if (null lstolsts) '(())
    (if (null (rest lstolsts))
        (mapcar #'list (first lstolsts))
      (combine-list-of-lsts-aux (cddr lstolsts)
                                (combine-lst-lst (first lstolsts)
                                                 (second lstolsts))))))

;; Pruebas
(combine-list-of-lsts NIL)  ;; --> (NIL)
(combine-list-of-lsts '(() (+ -) (1 2 3 4))) ;; --> NIL
(combine-list-of-lsts '((a b c) () (1 2 3 4))) ;; --> NIL
(combine-list-of-lsts '((a b c) (1 2 3 4) ())) ;; --> NIL
(combine-list-of-lsts '((1 2 3 4))) ;; --> ((1) (2) (3) (4))
(combine-list-of-lsts '((a b c) (+ -) (1 2 3 4)))
```

- **COMENTARIOS**

En este apartado nos resulta muy útil la implementación realizada en el apartado 1 ya que nos permite combinar listas con elementos lista de una manera sencilla.

Además se puede apreciar que utilizamos una recursión de cola, recurriendo a un acumulador en cada etapa de la recursión.