MEMORIA PRÁCTICA 1

Victoria Pelayo e Ignacio Rabuñal.

-**Ejercicio1**

1.1 Para este ejercicio, a parte de la función pedida por el enunciado, hemos creado dos auxiliares. Una que realiza el producto escalar entre dos vectores y otra que aplica la formula, una vez que ya se tienen los productos escalares.

Pseudocódigo:

Scalar-product

Entrada: x (vector), y (vector)

Salida: res (producto escalar de x e y)

Proceso:

Para cada i <= dimensión(x)

Res[i] = x[i] + y[i]

Formula coseno distancia:

Entrada: xx(norma vector x), yy(norma vector y), xy (producto escalar de x e y)

Salida: res (coseno-distancia entre x e y)

Proceso:

Res = 1 – (xy / (xx * yy))

Cosine-distance-rec

Entrada: x(vector), y(vector)

Salida: res (coseno distancia entre x e y)

Proceso:

Si (norma(x) = 0 o norma(y) = 0):

Res = NIL

Si no:

Res = Formula-coseno-distancia (procuto-escalar(x,x), producto-escalar(y,y) , producto-escalar(x,y))

Scalar-product-mapcar

Entrada: x(vector), y(vector)

Salida: res (producto escalar de x e y)

Proceso:

Lista [i] = x[i] * y[i]

Para cada i <= dimension(lista)

Res = res + lista[i]

Cosine-distance-mapcar

Entrada: x(vector), y(vector)

Salida: res(coseno-distancia entre x e y)

Proceso:

Si x = 0 o y = 0:

Res = NIL

Si no:

Res = formula-coseno-distancia (producto-escalar-mapcar(x,x) , producto-escalar-mapcar(y,y), producto-escalar-mapcar(x,y))

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; scalar-product (x y)
;;; Calcula el producto escalar de dos vectores.
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT: x: vector, representado como una lista
;;;        y: vector, representado como una lista
;;; OUTPUT: producto escalar entre x e y
;;;
(defun scalar-product (x y)
  (if (or(null x)(null y)) 0
    (+(* (car x) (car y))
    (scalar-product (cdr x) (cdr y)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; formula-cos-dis
;;; Aplica la formula de la distancia coseno
;;;
;;; INPUT: xx: norma del vector x
;;;       yy: norma del vector y
;;;                                  xy: producto escalar de x e y
;;; OUTPUT: coseno distancua entre x e y
;;;
(defun formula-cos-dis (xx yy xy)
                  (- 1 (/ xy
    (* (sqrt xx)
    (sqrt yy)))))

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

```
;;; 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.
;;; Si uno de los dos es el vector 0, se devuelve NIL
;;;
;;; INPUT: x: vector, representado como una lista
;;;        y: vector, representado como una lista
;;; OUTPUT: distancia coseno entre x e y
;;;
(defun cosine-distance-rec (x y)
  (let ((xy  (scalar-product x y)) (xx (scalar-product x x)) (yy (scalar-product y y)))
    (if (or (= 0 xx) (= 0 yy))
    nil
    (formula-cos-dis xx yy xy))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; scalar-product-mapcar (x y)
;;; Calcula el producto escalar de dos vectores.
;;; En vez de recursivamente, utilizando "mapcar"
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT: x: vector, representado como una lista
;;;        y: vector, representado como una lista
;;; OUTPUT: producto escalar entre x e y
;;;
(defun scalar-product-mapcar (x y)
                (if (or (null x) (null y)) 0
                (apply #'+ (mapcar #'* x y))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; cosine-distance-mapcar
;;; Calcula la distancia coseno de un vector usando mapcar
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;;
;;; INPUT:  x: vector, representado como una lista
;;;        y: vector, representado como una lista
;;; OUTPUT: distancia coseno entre x e y
;;;
(defun cosine-distance-mapcar (x y)
                (let((xy (scalar-product-mapcar x y)) (xx (scalar-product-mapcar x x)) (yy (scalar-
product-mapcar y y)))
                (if (or (= 0 xx) (= 0 yy))
                nil
                                (formula-cos-dis xx yy xy))))
```

Ejemplos de ejecución, con la función recursiva (hemos comprobado con una calculadora que el resultado es el correcto):

- (cosine-distance-rec '(1 2) '(1 2 3)) → 0,40238577
- (cosine-distance-rec nil '(1 2 3)) → nil
- (cosine-distance-rec '() '()) → nil
- (cosine-distance-rec '(0 0) '(0 0)) → nil

Ejemplos de ejecución, con la función que utiliza mapcar:

- (cosine-distance-mapcar '(1 2) '(1 2 3)) → 0,40238577
- (cosine-distance-mapcar nil '(1 2 3)) → nil

- (cosine-distance-mapcar '() '()) → nil
- (cosine-distance-mapcar '(0 0) '(0 0)) → nil

1.2 Para este apartado hemos creado una función auxiliar que devuelve la lista de tuplas de la confianza y el vector correspondiente.

Como no se especifica nada en el enunciado, para calcular la distancia coseno utilizamos la función implementada con mapcar.

Además, hemos creado una función que devuelve la copia de una lista para trabajar sobre ella, a la hora de ordenar por ejemplo, y no sobre la original. Esta función la utilizaremos más adelante también.

Pseudocódigo:

Simil-vector-lst

Entrada: categoria(vector), lista(vector de vectores), confianza

Salida: Res(lista de tuplas vector-confianza)

Proceso:

    Para cada i <= dimensión(lista)

    Sim = coseno-distancia(vector, lista[i])

    Si (1 - confianza) >= sim:

        Añado a res [vector, sim]

Copia-lista

Entrada: lista

Salida: copia

Proceso

    Si llista es un átomo:

        Copia = lista

    Si no:

        Copia = (lista , copia-lista(resto(lista)))

Order-vectors-cosine-distance

Entrada: categoria(vector), lista(vector de vectores), confianza

Salida: res (lista de vectores)

Proceso:

    Si (lista = null) o (categoría = null) o (confianza < 1)

        Salida = null

Else:

Aux = simil-vector-lst(categoría,lista,confianza)

Res = ordenar-de-menor-a-mayor(aux)

Código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; simil-vector-lst
;;; Lista de vectores y su similaridad.
;;; Solo van a pertenecer a esta lista los que superen el niveld e confianza
;;; INPUT:  vector: vector que representa a una categoria,
;;;              representado como una lista
;;;       lst-of-vectors vector de vectores
;;;       confidence: Nivel de confianza
;;; OUTPUT: Lista de tuplas (vector similitud)
;;;
(defun simil-vector-lst (x lst confidence)
              (mapcan #'(lambda(y)
                                                  (let ((sim (cosine-distance-mapcar x y)))
                                                  (if (>= (- 1 confidence) sim) (list (list y sim))))) lst))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; copia-lista
;;; Devuelve una copia de una lista
;;; INPUT:  lst: lista que queremos copiar
;;; OUTPUT: Una lista que es una copia de lst
;;;
(defun copia-lista (lst)
              (if (atom lst)
                            lst
                            (cons (car lst) (copia-lista (cdr lst)))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; order-vectors-cosine-distance
;;; Devuelve aquellos vectores similares a una categoria
;;; INPUT:  vector: vector que representa a una categoria,
;;;              representado como una lista
;;;       lst-of-vectors vector de vectores
;;;       confidence-level: Nivel de confianza (parametro opcional)
;;; OUTPUT: Vectores cuya semejanza con respecto a la
;;;       categoria es superior al nivel de confianza ,
;;;       ordenados
;;;
(defun order-vectors-cosine-distance (vector lst-of-vectors &optional (confidence-level 0))
              (if (or (null vector) (null lst-of-vectors) (> confidence-level 1) (< confidence-level 0))  nil
                            (mapcar #'first (sort (copia-lista (simil-vector-lst vector lst-of-vectors
confidence-level)) #'< :key #'second))))
```

Ejemplos de ejecución:

- (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.99)
  NIL

- (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.3)
  ((4 2 2) (32 454 123) (133 12 1))
- (order-vectors-cosine-distance '(1 2 3) '((32 454 123) (133 12 1) (4 2 2)) 0.5)
  ((4 2 2) (32 454 123))
- (order-vectors-cosine-distance '(1 2 3) '()) NIL
- (order-vectors-cosine-distance '() '((4 3 2) (1 2 3))) NIL

1.3

Para esta última función hemos creado dos funciones auxiliares. Una para calcular la distancia coseno entre cada una de las categorías y un vector; y otra que elige la mejor categoría para un vector.

Pseudocódigo:

Cats-simil

Entrada: vector, categories(lista de categorías(vectores)), función-distancia

Salida: Res (lista de tuplas categoría-distancia)

Proceso:

Para cada i <= dimensión(categorías)

Res [i] = (categoría[i], función-distancia(vector, categoría[i]))

Best-cat

Entrada: Entrada: vector, categories(lista de categorías(vectores)), función-distancia

Salida: Res(Tupla(lista) numero-categoria, distancia-coseno)

Proceso:

Lista aux = ordenar-demenor-a-mayor (cats-simil (vector, categorías, función-distancia))

Res = aux[0]

Get-vectors-category

Entrada: categorías, texts (lista de vectores), función-distancia

Salida: Res (Lista de tuplas de la forma categoría que minimiza distancia- dicha distancia, para cada vector)

Proceso:

Para cada i <= dimensión(texts)

Res[i] = Best-cat(categorías , texts[i], función-distancia)

Código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; cats-simil
;;; Calcula la distancia coseno de un vector y una lista de categorias
;;; INPUT:  vector: vector
;;;       categories: lista de categorias
;;;       distance-measure: funcion a usar para la distancia coseno
;;; OUTPUT: Lista de tuplas (categoria distancia)
;;;
(defun cats-simil (vector categories distance-measure)
                (mapcar #'(lambda(x) (list (car x) (funcall distance-measure (cdr vector) (cdr x))))
categories))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; best-cat
;;; Busca la mejor categoria para un vector
;;; INPUT:  vector: vector
;;;       categories: lista de categorias
;;;       distance-measure: funcion a usar para la distancia coseno
;;; OUTPUT: Tupla (numero-de-categoria distancia-coseno)
;;;
(defun best-cat (vector categories distance-measure)
                (car (sort (copia-lista (cats-simil vector categories distance-measure)) #'< :key
#'second)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; get-vectors-category (categories vectors distance-measure)
;;; Clasifica a los textos en categorias .
;;;
;;; INPUT : categories: vector de vectores, representado como
;;;                 una lista de listas
;;;       texts:      vector de vectores, representado como
;;;                 una lista de listas
;;;       distance-measure: funcion de distancia
;;; OUTPUT: Pares formados por el vector que identifica la categoria
;;;       de menor distancia , junto con el valor de dicha distancia
;;;
( defun get-vectors-category (categories texts distance-measure)
                (unless (null (car texts))
                                (mapcar #'(lambda(x) (best-cat x categories distance-measure)) texts)))
```

Pruebas de ejecución:

- (get-vectors-category '(()) '(()) #'cosine-distance-rec)
  NIL
- (get-vectors-category '(()) '(()) #'cosine-distance-mapcar)
  NIL
- (get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-mapcar)
  ((2 0.40238577))
- (get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-rec)
  ((2 0.40238577))
- (get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance-rec)
  ((NIL NIL) (NIL NIL))

- (get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance-mapcar)
  ((NIL NIL) (NIL NIL))
- (setf categories '((1 43 23 12) (2 33 54 24)))
  ((1 43 23 12) (2 33 54 24))

  (setf texts '((1 3 22 134) (2 43 26 58)))

  ((1 3 22 134) (2 43 26 58))

  (get-vectors-category categories texts #'cosine-distance-rec)

  ((2 0.5101813) (1 0.18444914))

  (get-vectors-category categories texts #'cosine-distance-mapcar)
((2 0.5101813) (1 0.18444914))

1.4

Para el ejemplo que nos piden, como tarda tan poco el tiempo nos sale 0.

```
CG-USER(2): (setf categories '((1 43 23 12) (2 33 54 24)))
((1 43 23 12) (2 33 54 24))
CG-USER(3): (setf texts '((1 3 22 134) (2 43 26 58)))
((1 3 22 134) (2 43 26 58))
CG-USER(4): (time (get-vectors-category categories texts #'cosine-distance-rec))
; cpu time (non-gc) 0.000000 sec user, 0.000000 sec system
; cpu time (gc)     0.000000 sec user, 0.000000 sec system
; cpu time (total)  0.000000 sec user, 0.000000 sec system
; real time  0.000000 sec
; space allocation:
;  28 cons cells, 512 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
((2 0.5101813) (1 0.18444914))

CG-USER(7): (time (get-vectors-category categories texts #'cosine-distance-mapcar))
; cpu time (non-gc) 0.000000 sec user, 0.000000 sec system
; cpu time (gc)     0.000000 sec user, 0.000000 sec system
; cpu time (total)  0.000000 sec user, 0.000000 sec system
; real time  0.000000 sec
; space allocation:
;  64 cons cells, 512 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
((2 0.5101813) (1 0.18444914))
CG-USER(8):
```

Ahora vamos a poner los resultados al evaluar los siguientes casos:

>>(get-vectors-category ’(()) ’(()) #’cosine-distance):

Hemos sustituido cosine-distance por la recursiva y la que utiliza mapcar, y en ambas nos pasa lo de arriba, que como tarda tan poco aparece en tiempo 0.000000sec

>> (get-vectors-categories ’((1 4 2) (2 1 2)) ’((1 1 2 3)) #’cosine-distance):

Hemos sustituido cosine-distance por la recursiva y la que utiliza mapcar, y en ambas nos pasa lo de arriba, que como tarda tan poco aparece en tiempo 0.000000sec

>> (get-vectors-categories ’(()) ’((1 1 2 3) (2 4 5 6)) #’cosine-distance)

Hemos sustituido cosine-distance por la recursiva y la que utiliza mapcar, y en ambas nos pasa lo de arriba, que como tarda tan poco aparece en tiempo 0.000000sec

EJERCICIO 2.

2.1

En este apartado no hemos necesitado crear ninguna función auxiliar.

Pseudocódigo:

Newton

Enrada: f (función), df (derivada de la función), max-iter, x0 (estimación inicial), tol (tolerancia para la convergencia).

Salida: Estimación del cero de la función o nil si no converge.

Proceso:

Si max-iter es 0:

Devolver nil.

Si |f(x0)| < tol:

Devolver x0.

New_x0 = x0 – f(x0)/df(x0)

Devolver newton(f, df, max-iter – 1, New_x0, tol).

Código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; newton
;;; Estima el cero de una funcion mediante Newton-Raphson
;;;
;;; INPUT : 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)
;;; OUTPUT: estimacion del cero de f o NIL si no converge
;;;
(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))))
```

Ejemplos de ejecución:

```
[2]>
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 3.0)
4.0
[3]> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 0.6)
0.99999946
[4]> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 30 -2.5)
-3.0000203
[5]> (newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 10 100)
NIL
```

2.2

En este apartado no hemos necesitado crear ninguna función auxiliar.

Pseudocódigo:

One-root-newton

Entrada: f (función), df (derivada de la función), max-iter, semillas (estimaciónes iniciales), tol (tolerancia para la convergencia).

Salida: Primera estimación del cero de la función o nil si no converge.

Proceso:

Si semillas es nulo:

Devolver nil

First = primer elemento de semillas

Rest = resto de semillas

newton(f, df, max-iter, first, tol)

one-root-newton(f, df, max-iter, rest, tol)

Código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; one-root-newton
;;; Prueba con distintas semillas iniciales hasta que Newton
;;; converge
;;;
;;; INPUT: f: funcion de la que se desea encontrar un cero
;;;        df: derivada de f
;;;        max-iter: maximo numero de iteraciones
;;;        semillas : semillas con las que invocar a Newton
;;;        tol : tolerancia para convergencia ( parametro opcional )
;;;
;;; OUTPUT: el primer cero de f que se encuentre, o NIL si se diverge
;;;          para todas las semillas
;;;
(defun one-root-newton (f df max-iter semillas &optional ( tol 0.001))
  (cond
    ((null semillas) nil)
    (t (newton f df max-iter (first semillas) tol))
    (t (one-root-newton f df max-iter (rest semillas) tol))))
```

Ejemplos de ejecución:

```
[7]>
(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))
0.99999946
[8]>
(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
                 #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(3.0 -2.5))
4.0
[9]>
(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
                #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(3.0 -2.5))
NIL
```

2.3

En este apartado no hemos necesitado crear ninguna función auxiliar. Hemos decidido usar mapcar en vez de recursión porque nos parecía más simple.

Pseudocódigo:

Entrada: f (función), df (derivada de la función), max-iter, semillas (estimaciónes iniciales), tol (tolerancia para la convergencia).

Salida: Lista de estimaciones para cada semilla.

Proceso:

Lista ls

Para cada semilla “s” de semillas:

Añadir a ls newton(f, df, max-iter, s, tol)

Devolver ls

Código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; all-roots-newton
;;; Prueba con distintas semillas iniciales y devuelve las raices
;;; encontradas por Newton para dichas semillas
;;;
;;; INPUT: f: funcion de la que se desea encontrar un cero
;;;        df: derivada de f
;;;        max-iter: maximo numero de iteraciones
;;;        semillas: semillas con las que invocar a Newton
;;;        tol : tolerancia para convergencia ( parametro opcional )
;;;
;;; OUTPUT: las raices que se encuentren para cada semilla o nil
;;;          si para esa semilla el metodo no converge
;;;
(defun all-roots-newton (f df max-iter semillas &optional ( tol 0.001))
    (mapcar #'(lambda(x) (newton f df max-iter x tol)) semillas))
```

Ejemplos de ejecución:

```
[2]> (defun all-roots-newton (f df max-iter semillas &optional ( tol 0.001))
  (mapcar #'(lambda(x) (newton f df max-iter x tol)) semillas))
ALL-ROOTS-NEWTON
[3]>
(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)
[4]>
(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)
[5]>
(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3)))
                  #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(0.6 3.0 -2.5))
(NIL NIL NIL)
```

2.3.1

En este apartado no hemos necesitado crear ninguna función auxiliar. Hemos decidido usar mapcan y unless en vez de recursión porque nos parecía más simple.

Pseudocódigo:

Entrada: f (función), df (derivada de la función), max-iter, semillas (estimaciónes iniciales), tol (tolerancia para la convergencia).

Salida: Lista de estimaciones no nulas para cada semilla.

Proceso:

Lista ls

Para cada semilla “s” de semillas:

Añadir a ls newton(f, df, max-iter, s, tol)

Eliminar nil de ls

Devolver ls

Código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; list-not-nil-roots-newton
;;; Elimina de la lista de salida de all-roots-newton los
;;; elementos que sean nil
;;;
;;; INPUT: f : funcion de la que se desea encontrar un cero
;;;        df : derivada de f
;;;        max-iter : maximo numero de iteraciones
;;;        semillas : semillas con las que invocar a Newton
;;;        tol : tolerancia para convergencia ( parametro opcional )
;;;
;;; OUTPUT: Las raices no nil que se encuentran para las semillas
;;;
(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)))
```

Ejemplos de Ejecución:

```
[4]> (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)
```

EJERCICIO3.

3.1

Para este apartado no hemos necesitado crear ninguna función auxiliar.

Pseudocódigo:

Combine-elt-lst

Entrada: elem (atomo), lista

Salida: Res (lista de tuplas elemento-elemento_lista)

Proceso:

Para i <= dimensión(lista):

Res[i] = (elemento, lista[i])

Código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst
;;; Combina un elemento dado con todos los elementos de una lista
;;;
;;; INPUT: elem: elemento a combinar
;;;       lst: lista con la que se quiere combinar el elemento
;;;
;;; OUTPUT: lista con las combinacion del elemento con cada uno de los
;;;        de la lista
(defun combine-elt-lst (elt lst)
              (if (null lst) '()
              (mapcar #'(lambda(x) (list elt x)) lst)))
```

Ejemplos de ejecución:

- (combine-elt-lst 'a '(1 2 3)) → ((A 1) (A 2) (A 3))
- (combine-elt-lst nil nil) → NIL
- (combine-elt-lst nil '(a b)) → ((NIL A) (NIL B))

3.2

En este ejercicio teníamos que crear una función que hiciese el producto cartesiano de dos listas.

Hemos creado una función auxiliar llamada limpiar para eliminar los paréntesis sobrantes.

Pseudocódigo:

Limpiar

Entrada: lista

Salida: Res(lista)

Proceso:

Mientras i <= dimension(lista):

Res = lista[i] , limpiar(lista i:) //los elementos posteriores al i

Combine-lst-lst

Entrada: lst1(lista), lst2(lista)

Salida: Res (Lista de tuplas de elementos de las dos listas combinados)

Proceso:

Mientras i <= dimensión (lst1):

Mientras j <= dimensión(lst2):

Añadir-a-Res (lst[i] lst[j])

Código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; limpiar
;;; funcion auxiliar para evitar parentesis "extra"
;;;
;;; INPUT lst:lista quu queremos "limpiar"
;;; OUTPUT: lista de la manera pedida en el ejercicio 3.3
;;;
(defun limpiar (lst)
        (if (null (cdr lst))
                (car lst)
                (append (car lst) (limpiar (cdr lst)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst
;;; Calcula el producto cartesiano de dos listas
;;;
;;; INPUT: lst1: primera lista
;;;        lst2: segunda lista
;;;
;;; OUTPUT: producto cartesiano de las dos listas
(defun combine-lst-lst (lst1 lst2)
        (limpiar (mapcar #'(lambda(x) (combine-elt-lst x lst2)) lst1 )))
```

Ejemplo de ejecución:

- (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 '(1 2 3) '(a b c)) ➔((1 A) (1 B) (1 C) (2 A) (2 B) (2 C) (3 A) (3 B) (3 C))

Si una de las dos listas es NIL el producto cartesiano es NIL.

3.3

Pseudocódigo:

Combine-elt-lst-aux

Entrada: elem (atomo), lista

Salida: Res(lista)

Proceso:

```
Si (lista == null):
        Res = null
Si no:
        Res[i] = (elem lista[i])
```

Combine-list-of-lsts-aux

Entrada: lst1, lst2

Salida: Res

Proceso:

```
Si (lst1 = null):
        Res = lst2
Si (lst2 = null):
        Res = lst1
Si no:
        Mientras i <= dimensión (lst1):
        Mientras j <= dimensión(lst2):
                Añadir-a-Res (lst[i] lst[j]) //utilizar funciones de Lisp adecuadas para
                                             //evitar paréntesis extra
```

Combine-list-of-lsts

Entrada: lstolsts (lista de listas)

Salida: Res (lista, combinación de las anteriores)

Proceso:

Si (lstolsts = null):

Res = null

Si no:

Res = Combine-list-of-lsts-aux (lstolsts (1) , combine-list-of-lsts (Cdr (lstolsts)))

Código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst-aux
;;; Combina un elemento dado con todos los elementos de una lista
;;; Igual a combine-elt-lst pero utilizamos la funcion "cons" para evitar
;;; problemas de parentesis
;;; Creaada como funcion auxiliar de combine-list-of-lsts
;;;
;;; INPUT: elem: elemento a combinar
;;;       lst: lista con la que se quiere combinar el elemento
;;;
;;; OUTPUT: lista con las combinacion del elemento con cada uno de los
;;;        de la lista
(defun combine-elt-lst-aux (elt lst)
                (if (null lst) '()
                (mapcar #'(lambda(x)(cons elt x)) lst)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts-aux
;;; Funcion auxiliar que combina todos los elementos de dos listas
;;; Utiliza la funcion combine-elt-lst-aux, creada especificamente para
;;; este ejercicio y que no haya "problemas de parentesis"
;;;
;;; INPUT: lst1: lista 1
;;;       lst2: lista 2
;;;
;;; OUTPUT: lista con todas las posibles combinaciones de elementos
(defun combine-list-of-lsts-aux (lst1 lst2)
                (cond ((null lst1) lst2)
                ((null lst2) lst1)
                (T (limpiar (mapcar #'(lambda(x) (combine-elt-lst-aux x lst2)) lst1)))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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
;;;
;;; INPUT: lstolsts: lista de listas
;;;
;;; OUTPUT: lista con todas las posibles combinaciones de elementos
```

```
(defun combine-list-of-lsts (lstolsts)
        (if (null lstolsts) '(())
        (combine-list-of-lsts-aux (car lstolsts) (combine-list-of-lsts (cdr lstolsts)))))
```

En este apartado hay que crear una función que combine todos los elementos de varias listas dadas.

Para ello hemos creado una función auxiliar que es una modificación de la del ejercicio1, esta modificación evita los problemas de paréntesis.

Y otra función auxiliar que combina todos los elementos de dos listas.

Ejemplos de ejecución:

- (combine-list-of-lsts '(() (+ -) (1 2 3 4))) → ((+ 1) (+ 2) (+ 3) (+ 4) (- 1) (- 2) (- 3) (- 4))
  (Si una lista es nil, contamos como si no estuviese)
- (combine-list-of-lsts '((a b c) () (1 2 3 4))) → ((A 1) (A 2) (A 3) (A 4) (B 1) (B 2) (B 3) (B 4) (C 1) (C 2) (C 3) (C 4))
- (combine-list-of-lsts '((a b c) (1 2 3 4) ())) → ((A 1) (A 2) (A 3) (A 4) (B 1) (B 2) (B 3) (B 4) (C 1) (C 2) ...)
- (combine-list-of-lsts '((1 2 3 4))) → ((1) (2) (3) (4))
- (combine-list-of-lsts '(nil)) → NIL
- (combine-list-of-lsts '((a b c) (+ -) (1 2 3))) ➔ ((A + 1) (A + 2) (A + 3) (A - 1) (A - 2) (A - 3) (B + 1) (B + 2) (B + 3) (B - 1) ...)
- (combine-list-of-lsts '((a b c) (+ -) (1 2 3) (=) (hola caracola))) ➔ ((A + 1 = HOLA) (A + 1 = CARACOLA) (A + 2 = HOLA) (A + 2 = CARACOLA) (A + 3 = HOLA) (A + 3 = CARACOLA) (A - 1 = HOLA) (A - 1 = CARACOLA) (A - 2 = HOLA)
  (A - 2 = CARACOLA) ...)

EJERCICIO 4

Pseudocódigo:

---! = NOT

expand-bicond:

Entrada: fbf

Salida: res (FBF- no empiez por bicondicional)

Proceso:

Res = (fbf[2] and fbf[3]) or (! Fbf[2] and ! fbf[3])

Expand-cond

Entrada: fbf

Salida: res(fbf- no empieza por un condicional)

Proceso:

Res = (! Fbf[2]) or fbf[3]

Expand

Entrada:fbf

Salida: fbf(“expandida”)

Proceso:

Si (fbf[1] = ⇔):

Expand-bicond(fbf)

Si no:

Expand-cond(fbf)

Create-childs:

Entrada: lst(lista de literales), fbf

Salida: res(lista)

Proceso:

Si (fbf[1] = or):

Mientras i <= dimension(fbf)-1:

Res[i] = expand-truth-tree-aux(lst, fbf[I + 1])

Si no:

And-tree (expand-truth-tree-aux(rest(fbf)))

And-tree

Entrada: result (lista), fbf

Salida: Res (lista de listas de atomos)

Proceso:

Si (lst = null):

Res = result

Si no:

Para cada elemento “x” de lst:

List (Append (result, list(x)))

**list: transforma en una lista

**append: añade una lista a otra

Expand-truth-tree-aux

Entrada: lst, fbf

Result: Res (lista de listas de atomos)

Proceso:

Cuando (fbf =! null):

Si (fbf no es un literal):

Si (fbf[1] = not): res = expand-truth-tree-aux(eliminar-not (fbf))

Si (fbf[1] = ⇔):res = expand-truth-tree-aux (eliminar-bicondicional(fbf))

Si(fbf[1] = ➔):res = expand-truth-tree-aux(eliminar-codicional(fbf))

Si (fbf[1] es un conector n-ario): res =créate-childs (lst,fbf)

Si(fbf[1] es un literal): res =and-tree (list(fbf[1]) , expand-truth-tree-aux(rest(fbf)))

si no: res = and_tree (expand-truth-tree-aux(fbf[1]), expand-truth-tree-aux(rest(fbf)))

si no:

si (lst = null & fbf es un literal): res = and-tree(fbf[1], lst)

si no: res = lst

Eliminar-not

Entrada: x (fbf)

Salida: x (sin negaciones extra, solo se aceptan !a, siendo a un atomo)

Proceso:

Si(x = null OR x es un literal): res = x

Si no:

Connector = x[1]

Si (conector = not):

Aux = fbf[2]

Si (aux = not): res = eliminar-not (aux[2])

Si no:

Para cada i >= 1 y i < dimensión(fbf)

Z[i-1] = not fbf[i]

Res = Eliminar-not (cons (intercambio fbf[1]) , z)

Si no:

Res = cons(connector, eliminar-not(rest(fbf)))

***cons tiene el mismo significado que la función en Lisp que tiene el mismo nombre

Eliminar-parentesis

Entrada: atomos(lista), fbf

Resultado: res (lista sin paréntesis extra)

Proceso:

Si(fbf = null): res = null

Si (fbf es un literal): res = cons(fbf, atomos)

Si(fbf[1] es un literal): res = append(list(fbf[1]) , eliminar-parentesis(átomos, rest(fbf)))

Si no: res = append(fbf[1], eliminar-parentesis(átomos, rest(fbf)))

Intercambio

Entrada: connector

Salida: Res (and o or)

Proceso:

Si (connector = or): res = and

Si(connector = and): res = or

Si no: res = connector

Valor-verdad

Entrada: x (lista de valores de verdad)

Salida: res(Valor verdad)

Proceso:

Para cada i <= dimensión(x):

If(x[i] = nil):

res = nil

Return

Res = True

Evaluar

Entrada: lst (lista de atomos)

Salida: Res (lista de valores de verdad)

Proceso:

Si (lst = null OR lst es un literal): res = lst

Si no: res[i] = evaluar-aux(lst[i], lst)

Evaluar-aux

Entrada: elt(literal), lst(lista)

Salida: True o False

Proceso:

Para i <= dimensión(lst):

Si (elt = not lst[i]): return False

Return True

SAT

Entrada: ramas (lista de valores de verdad)

Salida: True o False

Proceso:

Para i <=dimensión(ramas):

Si(ramas[i] = True):

Return True

Return False

Truth-tree

Entrada: fbf

Salida: True or false

Proceso:

Si (fbf = null): return False

Si no:

Si alguna de las ramas del árbol es True: return true

Si no: **return False**

Código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; expand-bicond
;;; Recibe una expresion con bicondicional y la transforma en una con
;;; and's y or's
;;; determinar si es SAT o UNSAT
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar.
;;; OUTPUT : fbf - Formula bien formada (FBF) sin bicondicional
;;;
(defun expand-bicond (fbf)
 (list +or+ (list +and+ (second fbf) (third fbf)) (list +and+ (list +not+ (second fbf) +not+ (third
fbf)))))

 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; expand-cond
;;; Recibe una expresion con condicional y la transforma en una con
;;; and's y or's
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar.
;;; OUTPUT : fbf - Formula bien formada (FBF) sin condicional
;;;
(defun expand-cond (fbf)
  (list +or+ (list +not+ (second fbf)) (third fbf)))


 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; expand
;;; Recibe una expresion con condicional o bicondicional
;;; y la transforma en una con and's y or's
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) con condicionales
;;; OUTPUT : fbf - Formula bien formada (FBF) sin condicionales
;;;

(defun expand(fbf)
		(if (equal T (bicond-connector-p (first fbf)))
						(list +and+ (expand-bicond fbf))
		(list +and+ (expand-cond fbf))))

;;;
;;; create-childs
;;; va haciendo llamadas recursivas a expand-truth-tree-aux
;;; segun si es un and o un or
;;;
;;; INPUT: lst lista literales hasta ahira
;;;						fbf - formula bien formada
;;; OUTPUT: lista no contradictoria de literales
;;;
(defun create-childs (lst fbf)
		(if (equal +or+ (first fbf))
				(mapcar #'(lambda(x) (expand-truth-tree-aux lst x)) (cdr fbf))

		(and_tree (expand-truth-tree-aux lst (cdr fbf)) lst)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; and_tree
;;; Organiza la lista para cuando haya un and
;;;
;;; INPUT  : result : resultado de la funcion expand-truth-tree-aux (de los
;;;                                                         operadores del and)
;;;                         lst: lista de atomos
;;; OUTPUT : lista que debe ser evaluable su valor de verdad
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun and_tree (result lst)
		(if (null lst) result
		(mapcar #'(lambda(x)(append result (LIST x))) lst)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; expand-truth-tree-aux
;;; Se encarga de ir ramificando la fbf de manera adecuada para finalmente
;;; poder comprobar si es verdad (en una funcion exterior)
;;;
;;; INPUT  : lst: lista de atomos
;;;                                    fbf: formula bine formada
;;; OUTPUT : lista de listas de atomos.
;;;                                    se puede ver como un arbol, en el que cada rama es una lista
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun expand-truth-tree-aux (lst fbf)
		(when (not (null fbf))

		(if (and (not (literal-p fbf)) (not (null fbf)))
			(cond ((unary-connector-p (first fbf)) (expand-truth-tree-aux lst (eliminar-
not fbf)))
				( (binary-connector-p (first fbf)) (expand-truth-tree-aux lst
(expand fbf)))
				( (n-ary-connector-p (first fbf))(create-childs lst fbf))
				( (literal-p (first fbf)) (and_tree (list (first fbf)) (expand-truth-tree-
aux lst (cdr fbf))))
				(T (and_tree (expand-truth-tree-aux lst (first fbf)) (expand-truth-
tree-aux lst (cdr fbf)) lst)))
			(if (and (null lst)(literal-p fbf)) (and_tree (list fbf) lst)
				lst))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; eliminar-not
;;; Reduce la expresion a un literal o a un literal negado
;;;
;;; INPUT  : x - expresion de un literal negado multiples veces
;;; OUTPUT : x - literal negado o literal positivo
;;;
(defun eliminar-not (x)
		(if (or (null x) (literal-p x)) x
		;;si no eliminamos...
		(let ((connector (first x)))
		;;Comprobamos si es una negacion
		(if (unary-connector-p connector)
			(let ((aux (second x)))
			;;si es un not volvemos a reducir
			(if (unary-connector-p (first aux)) (eliminar-not (second aux))
			;;si no modificamos la operacion
				(eliminar-not (cons (intercambio (first aux))
```

```
                                                                (mapcar #'(lambda(x) (list +not+
x)) (cdr aux)))))
                                )
                (cons connector (mapcar #'eliminar-not (cdr x)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; eliminar-parentesis
;;; Elimina los parentesis necesarios de una lista para solo obtener atomos
;;;
;;; INPUT  : atomos : lista de atomos
;;;                                   fbf: lista de atomos resultante de expand-truth-tree-aux
;;;                           lst: lista de atomos
;;; OUTPUT : lista de atomos que ya puede ser evaluable por las funciones creadas
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminar-parentesis (atomos fbf)
                (cond
                                ((null fbf) nil)
                                ((equal T (literal-p fbf)) (cons fbf atomos))
                                ((equal T (literal-p (first fbf))) (append (list (first fbf)) (eliminar-parentesis
atomos (cdr fbf))))
                                (T (append (first fbf) (eliminar-parentesis atomos (cdr fbf))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; intercambio
;;; Intercambia un and por un or
;;;
;;; INPUT  : connector: conector que debe ser intercambiado
;;;                           lst: lista de atomos
;;; OUTPUT : un or (cuando conector = and) o un and (conector = or)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun intercambio (connector)
                (cond
                                ((equal +or+ connector) +and+)
                                ((equal +and+ connector) +or+)
                                (t connector)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; valor-verdad
;;; Devuelve T o NIL segun el valor-verdad de esa lista
;;;
;;; INPUT  : x: lista de valores de verdad (T o F)
;;; OUTPUT : T si son todos T, NIL, si no
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun valor-verdad (x)
                (cond       ((null x) T)
                                        ((equal NIL (first x)) nil)
                                        ((valor-verdad (cdr x)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; evaluar
;;; Evalua si hay contradicciones en una lista de atomos
;;;
```

```
;;; INPUT  : lst: lista de atomos
;;;
;;; OUTPUT : lista con tantos NIL como atomos con contradiccion haya
;;;                                                 si un atomo no tiene contradccion le corresponde un T
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun evaluar (lst)
		(if (or (null lst) (literal-p lst)) lst
				(mapcar #'(lambda(x) (evaluar-aux (eliminar-not x) (cdr lst))) lst)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; evaluar-aux
;;; Mira las contradicciones de un literal con otra lista de literales
;;;
;;; INPUT  : elt: listeral que miramos si tiene contradicciones
;;;                                   lst: lista de atomos
;;; OUTPUT : T si no hay contradiccion, NIL, si la hay
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun evaluar-aux (elt lst)
		(if (or (null lst) (literal-p lst))
				(not (equal elt (eliminar-not (list +not+ lst))))
		(and (evaluar-aux elt (first lst)) (evaluar-aux elt (cdr lst)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; SAT
;;; Mira si un arbol es SAT
;;;
;;; INPUT  : ramas: lista de listas de atomos
;;; OUTPUT : T si alguna de las ramas no tiene contradicciones, Nil si no
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun SAT (ramas)
		(cond ((null ramas) nil)
				((equal T (first ramas)) T)
				(T (SAT (cdr ramas)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; truth-tree
;;; Recibe una expresion y construye su arbol de verdad para
;;; determinar si es SAT o UNSAT
;;;
;;; INPUT  : fbf - Formula bien formada (FBF) a analizar.
;;; OUTPUT : T   - FBF es SAT
;;;          N   - FBF es UNSAT
;;;
(defun truth-tree (fbf)
		(if (null fbf) NIL
		(SAT (mapcar #'(lambda(x) (valor-verdad (evaluar (eliminar-parentesis NIL
x))))(expand-truth-tree-aux NIL fbf)))))
```

No hemos añadido ni el código ni el pseudocódigo de las funciones proporcionadas en el enunciado, ya que no hemos modificado nada.

Ejemplos de ejecución:

>>CG-USER(35): (truth-tree '())

NIL

>>-USER(36): (truth-tree '(^ (v A B)))

T

>>CG-USER(37): (truth-tree '(^ (! B) B))

NIL

>>CG-USER(38): (truth-tree '(<=> (=> (^ P Q) R) (=> P (v (! Q) R))))

T

Hemos añadido algún ejemplo más:

>>CG-USER(9): (truth-tree '(v (v (! a) a) (v b c)))

T

>>(truth-tree '(v (a v b) ((! a) b)))

T

>>CG-USER(21): (truth-tree '(^ (=> (!a) a) (v b c) (<=> a (! b))))

T

>>CG-USER(22): (truth-tree '(^ (=> (!a) a) (v b c) (<=> a (! a))))

NIL

Pregunta 1:

Con (^ *A* (v *B C*)) obtenemos True, sin embargo, con (^ *A* (¬ *A*) (v *B C*)) debemos de obtener NIL, y eso es lo que obtenemos.

Pregunta 2:

(^ *A* (v *B C*) (¬ *A*)) de nuevo debemos obtener NIL, y es lo que obtenemos

Pregunta 3:

Devuelve una lista de átomos, que corresponde a una rama del árbol en la que no hay contradicciones.

En este ejercicio hemos decidido crear tantas funciones como necesitáramos e intentando hacer una correcta descomposición funcional.

El modelo que hemos seguido para la función *expand-truth-tree-aux* es uno parecido al explicado en clase.

Explicación de expand-tree-aux:

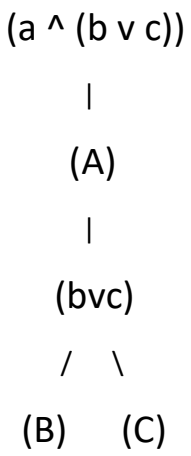
Partimos de tener la expresión sin condicionales/bicondicionales/not + fbf

Podemos ver la fbf como un nodo, si es una conjunción '^' cada elemento de la fbf es

Formará parte del mismo nodo.

Si es una fbf disyuntiva 'v' sacamos tantos hijos como elementos tenga la fbf.

Ejemplo:



Y esta función devolverá una lista de listas con cada una de las ramas(solo los átomos), en este caso:

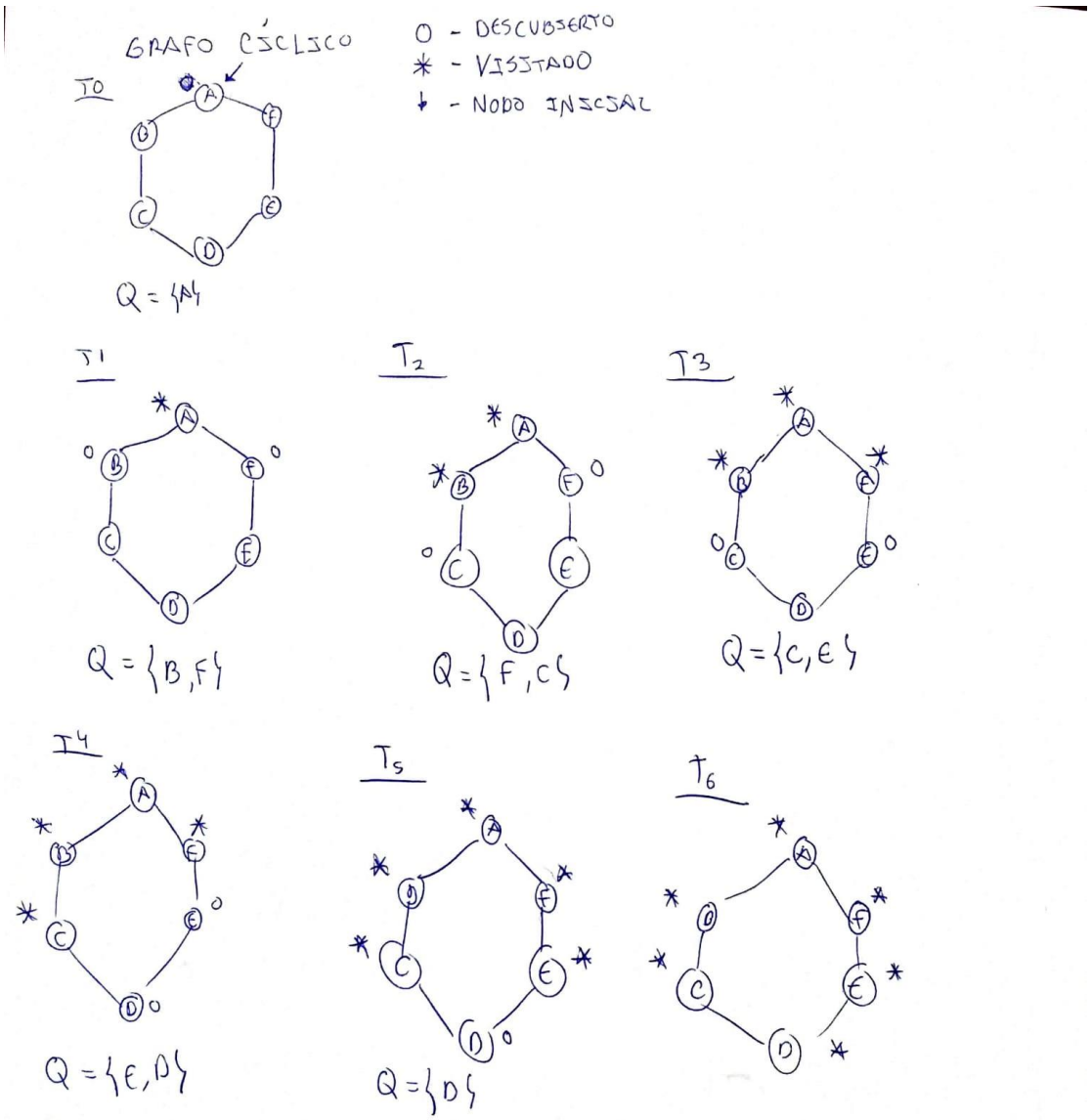
((A B) (A C))

Con esa lista llamaremos (desde truth-tree) a las funciones encargadas de evaluar su valor de verdad. En este caso devolverá que es SAT.

Para que sea SAT solo es necesario que **una** de las listas sea SAT, ya que habrá tantas listas como ramas, y estas son creadas cuando hay un or.

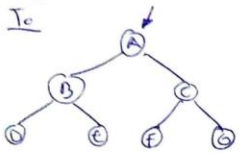
EJERCICIO 5.

5.1



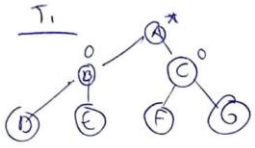
ARBOL BINARIO

T_0



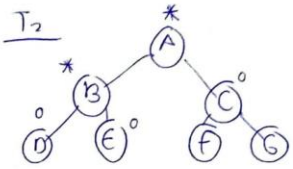
$Q = \{A\}$

T_1



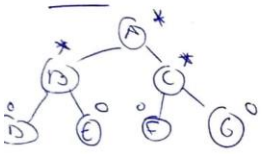
$Q = \{B, C\}$

T_2



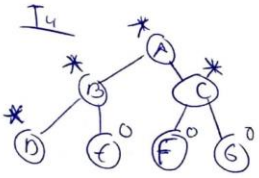
$Q = \{C, D, E\}$

T_3



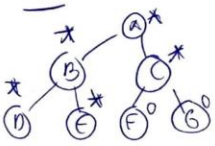
$Q = \{D, E, F, G\}$

T_4



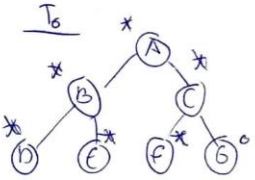
$Q = \{E, F, G\}$

T_5



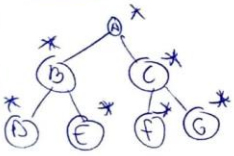
$Q = \{F, G\}$

T_6



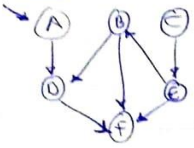
$Q = \{G\}$

T_7



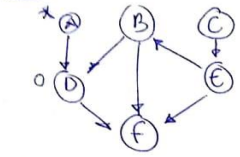
ESEMPIO DESDE A

T_0



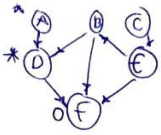
$Q = \{A\}$

T_1



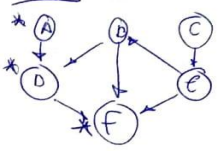
$Q = \{D\}$

T_2



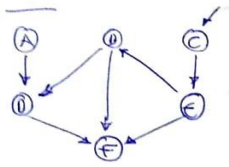
$Q = \{F\}$

T_3



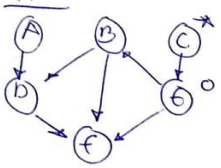
ESEMPIO DESDE C

T_0



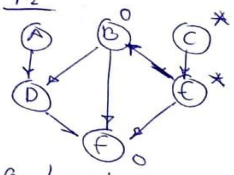
$Q = \{C\}$

T_1



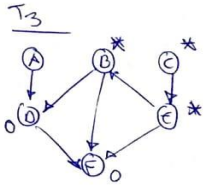
$Q = \{E\}$

T_2



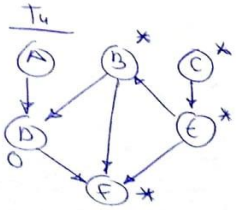
$Q = \{B, F\}$

T_3



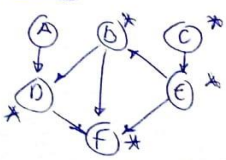
$Q = \{F, D\}$

T_4

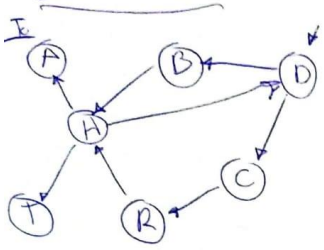


$Q = \{D\}$

T_5

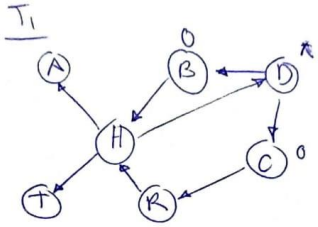


CASO TÍPICO



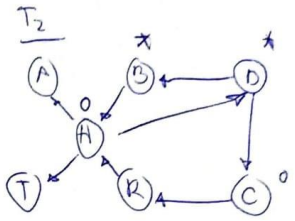
$Q = \{D\}$

T_1



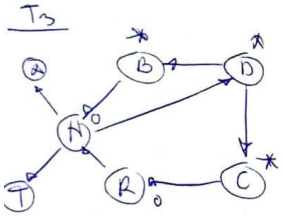
$Q = \{B, C\}$

T_2



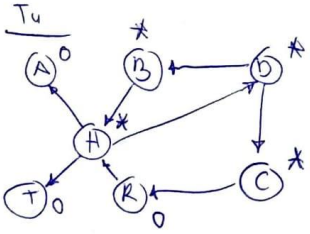
$Q = \{C, H\}$

T_3



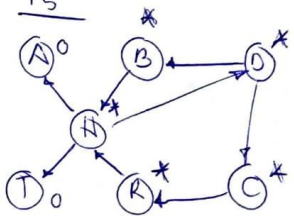
$Q = \{H, R\}$

T_4



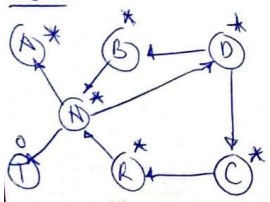
$Q = \{R, A, T\}$

T_5



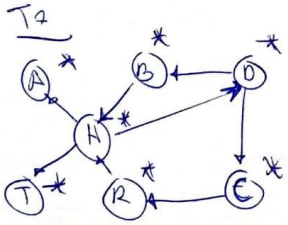
$Q = \{A, T\}$

T_6



$Q = \{T\}$

T_7



5.2

Pseudocódigo:

Si los Nodos están de blanco es que todavía no han sido descubiertos.

Si están de gris es que han sido descubiertos, pero no visitados.

Si están de negro es que han sido visitados.

Entrada: Grafo g, nodo inicio, nodo fin.

Salida: Éxito o Fracaso en la búsqueda.

Proceso:

```
Se pintan todos los nodos de g de blanco
Encolar inicio
Mientras la cola no sea vacía:
    Decolar el primer nodo de la cola
    Si el nodo es igual al actual:
        Devolver Éxito.
    Para cada vecino “v” del nodo actual:
        Si el “v” es blanco:
            Encolar v
    Pintar el nodo actual de negro
    Devolver Fracaso.paths
```

5.4

```lisp
;;; Breadth-first-search in graphs
;;;
(defun bfs (end queue net)
  (if (null queue) '()
    ;Como queue es una lista de listas, escogemos path = Una lista con el nodo
    ;inicial (1a iteracion) y escogemos node = primer nodo del path.
    (let* ((path (first queue))
        (node (first path)))
      (if (eql node end)
          ;Como escogemos como nodo actual el primer nodo de la cola, la
          ;recursion hace que el path en este punto este invertido, de ahi
          ;la funcion reverse.
          (reverse path)
        (bfs end
            ;Pasamos como nueva cola la cola anterior+la generada por new-paths
            (append (rest queue)
                    (new-paths path node net))
            net)))))

(defun new-paths (path node net)
  ;Para cada vecino de node añade el vecino al principio del path
  (mapcar #'(lambda (n)
              (cons n path))
            ;Crea una lista con los vecinos de node
            (rest (assoc node net))))
;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.5

Lo que hace la función shortest-path es llamar a la función bfs pero con la cola inicial de esta forma ‘((start)). La función bfs visita los nodos que estén a una distancia x del actual antes que los que estén a una distancia x+1. Por lo tanto, al iniciar la cola con el nodo de inicio y parar la búsqueda cuando se llega al destino, shortest-path devuelve el camino más corto desde start hasta end.

5.6

```
[12]> ( shortest-path 'a 'f '(( a d ) ( b d f ) ( c e ) ( d f ) ( e b f ) ( f )))
1. Trace: (SHORTEST-PATH 'A 'F '((A D) (B D F) (C E) (D F) (E B F) (F)))
2. Trace: (BFS 'F '((A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
3. Trace: (NEW-PATHS '(A) 'A '((A D) (B D F) (C E) (D F) (E B F) (F)))
3. Trace: NEW-PATHS ==> ((D A))
3. Trace: (BFS 'F '((D A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
4. Trace: (NEW-PATHS '(D A) 'D '((A D) (B D F) (C E) (D F) (E B F) (F)))
4. Trace: NEW-PATHS ==> ((F D A))
4. Trace: (BFS 'F '((F D A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
4. Trace: BFS ==> (A D F)
3. Trace: BFS ==> (A D F)
2. Trace: BFS ==> (A D F)
1. Trace: SHORTEST-PATH ==> (A D F)
(A D F)
```

5.7

```
[20]> (shortest-path 'b 'g
             '((a b c d e) (b a d e f) (c a g) (d a b g h)
             (e a b g h) (f b h) (g c d e h) (h d e f g)))
(B D G)
```

5.8

Al tener un grafo del tipo ‘((a b) (b a) (c)) (un grafo con a y b vecinos y c aislado). Si llamamos a la función shortest-path desde ‘a hasta ‘c esta entraría en un bucle infinito intentando llegar a c a través de ‘a o a través de b’ alternativamente y sin éxito, mientras el path se va llenando de ‘a y ‘b de la misma manera.

Para solucionar esto añadimos un control a la función bfs de manera que en cada llamada compruebe si el numero de veces que aparece en el path el nodo actual. Si este aparece más de una vez, es decir, estamos pasando dos veces por el mismo nodo y por lo tanto dando vueltas hasta el infinito, asique devolvemos nil (el nodo final no se puede alcanzar). En otro caso, la función continua ejecutándose como antes.

Codigo:

```
(defun bfs-improved (end queue net)
  (if (null queue) '()
     (let* ((path (first queue))
            (node (first path)))
        (if (eql node end)
              (reverse path)
          (if (> (count node path) 1) '()
            (bfs-improved end
                          (append (rest queue)
                                  (new-paths path node net))
                          net))))))

(defun shortest-path-improved (start end net)
  (bfs-improved end ( list ( list start )) net))
```