# Memoria Práctica 1

## Ejercicio 1:

Apartado 1:

Para este ejercicio hemos creado una función auxiliar que calcula el producto escalar de dos vectores:

*(defun producto-escalar-rec (x y)*

*(if (or (null x) (null y))*

*0*

*(+ (\* (first x) (first y)) (producto-escalar-rec (rest x) (rest y))))*

*)*

Y luego una función que haciendo uso de esa función calcule la distancia coseno entre dos vectores de manera recursiva:

*(defun cosine-distance-rec (x y)*

*(if (= 0 (\* (producto-escalar-rec x x) (producto-escalar-rec y y)))*

*nil*

*(cosine-distance (producto-escalar-rec x y) (\* (sqrt (producto-escalar-rec x x)) (sqrt (producto-escalar-rec y y)))))*

*)*

Haciendo uso de una formula que aplica la fórmula dada llamada cosine distance:

*(defun cosine-distance (x y)*

*(- 1 (/ x y))*

*)*

Luego haciendo uso de las funciones mapcar hemos reducido la función de la distancia coseno:

*(defun cosine-distance-mapcar (x y)*

*(if (= 0 (\* (apply #'+ (mapcar #'\* x x)) (apply #'+ (mapcar #'\* y y))))*

*nil*

*(cosine-distance (apply #'+ (mapcar #'\* x y)) (\* (sqrt (apply #'+ (mapcar #'\* x x))) (sqrt (apply #'+ (mapcar #'\* y y))))))*

*)*

Los resultados a las pruebas pedidas en el ejercicio 1.1 son:

1. (cosine-distance ’(1 2) ’(1 2 3)) → 0.40238577

2. (cosine-distance nil ’(1 2 3)) → 0

3. (cosine-distance ’() ’()) → 0

4. (cosine-distance ’(0 0) ’(0 0)) → 0

### Apartado 2:

En este apartado se pide crear una función que ordene los vectores de la lista según el valor de confianza con el vector de referencia, siempre y cuando superen el valor de confianza dado como argumento de entrada:

Para ello hemos creado dos funciones auxiliares: la primera para insertar en orden en una lista:

*(defun insert-ordered-cosine-distance (reference vector lst-of-vectors)*

*(cond ((null vector)*

*lst-of-vectors)*

*((null lst-of-vectors)*

*(cons vector lst-of-vectors))*

*((< (cosine-distance-mapcar reference vector) (cosine-distance-mapcar reference (first lst-of-vectors)))*

*(cons vector lst-of-vectors))*

*(t*

*(cons (first lst-of-vectors) (insert-ordered-cosine-distance reference vector (rest lst-of-vectors)))))*

*)*

Esta función recibe una referencia (el vector respecto del cual se ordena la lista) un vector y la lista para insertarlo, si el vector es nulo devuelve la lista como tal, si lo nulo es la lista devuelve el vector como único elemento de la lista, en caso de que el vector sea más cercano que la lista se inserta el primero y en caso contrario se pasa al siguiente por recursión.

Y la siguiente como parte recursiva de la función principal:

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

*(cond ((null lst-of-vectors)*

*nil)*

*(t*

*(insert-ordered-cosine-distance vector (first lst-of-vectors) (order-vectors-cosine-distance-rec vector (rest lst-of-vectors) confidence-level))))*

*)*

Esta función envía a la anterior de manera recursiva todos los elementos de la lista para que los vaya ordenando.

Por lo que al final la función principal se dedicaría solamente a llamar a la recursiva y ésta a la de insertar en orden:

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

*(order-vectors-cosine-distance-rec vector*

*(remove-if (lambda (v) (< (- 1 confidence-level) (cosine-distance-mapcar vector v))) lst-of-vectors)*

*confidence-level)*

*)*

Esta función una vez recibe la lista ordenada por distancia elimina los que no cumplen el nivel de confianza.

La salida a los ejercicios puestos es:

(order-vectors-cosine-distance ’(1 2 3) ’()) → NIL

(order-vectors-cosine-distance ’() ’((4 3 2) (1 2 3))) → ((1 2 3) (4 3 2))

### Apartado 3:

En este ejercicio se pide que dada una lista de textos y de categorías se le asigne a cada texto la categoría más cercana según la función de distancia asignada, para ello hemos creado dos funciones, la primera que dado un texto y una lista de categorías saque la que le corresponde al texto y la otra que dada una lista de textos y otra de categorías llame repetidamente a la primera para que saque las categorías.

1. **Primera función:**

*(defun get-category (categories text distance-measure minimum)*

*(cond ((null categories)*

*(list (first (first minimum)) (second minimum)))*

*(t*

*(if (< (funcall distance-measure text (first categories)) (second minimum))*

*(get-category (rest categories) text distance-measure (list (first categories) (funcall distance-measure text (first categories))))*

*(get-category (rest categories) text distance-measure minimum))))*

*)*

1. **Segunda función:**

*(defun get-vectors-category (categories texts distance-measure)*

*(cond ((null texts)*

*nil)*

*(t*

*(cons (get-category (rest categories) (first texts) distance-measure (list (first categories) (funcall distance-measure (first texts) (first categories)))) (get-vectors-category categories (rest texts) distance-measure))))*

*)*

### Apartado 4:

(get-vectors-category ’(()) ’(()) #’cosine-distance-mapcar) → **(NIL 0)**

(get-vectors-category ’((1 4 2) (2 1 2)) ’((1 1 2 3)) #’cosine-distance-mapcar) → (**2 0.39753592)**

(get-vectors-category ’(()) ’((1 1 2 3) (2 4 5 6)) #’cosine-distance-mapcar) **→ ((NIL 0) (NIL 0))**

# Ejercicio 2:

### Apartado 1:

*(defun newton (f df max-iter x0 &optional (tol 0.001))*

*(cond ((= max-iter 0)*

*nil)*

*((< (abs (funcall f x0)) tol)*

*x0)*

*(t*

*(newton f df (- max-iter 1) (- x0 (/ (funcall f x0) (funcall df x0))) tol)))*

*)*

### Apartado 2:

*(defun one-root-newton (f df max-iter semillas &optional (tol 0.001))*

*(cond ((null semillas)*

*nil)*

*((null (newton f df max-iter (first semillas) tol))*

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

*(t*

*(newton f df max-iter (first semillas) tol)))*

*)*

### Apartado 3:

1. **Función 1:**

*(defun all-roots-newton (f df max-iter semillas &optional (tol 0.001))*

*(cond ((null semillas)*

*nil)*

*(t*

*(cons (newton f df max-iter (first semillas) tol) (all-roots-newton f df max-iter (rest semillas) tol))))*

*)*

1. **Función 2:**

*(defun list-not-nil-roots-newton (f df max-iter semillas &optional ( tol 0.001))*

*(mapcan #'(lambda (x) (if (null x) nil (list x))) (all-roots-newton f df max-iter semillas tol))*

*)*

## Ejercicio 3:

### Apartado 1:

*(defun combine-elt-lst (elt lst)*

*(cond ((or (null lst) (null elt))*

*nil)*

*(t*

*(mapcar #'(lambda (x) (list elt x)) lst)))*

*)*

La salida para los casos propuestos es:

1. (combine-elt-lst ’a nil) **→** **NIL**

2. (combine-elt-lst nil nil) **→** **NIL**

3. (combine-elt-lst nil ’(a b)) **→ NIL**

### Apartado 2:

*(defun combine-lst-lst (lst1 lst2)*

*(mapcan #'(lambda (x) (combine-elt-lst x lst2)) lst1)*

*)*

La salida para los casos propuestos es:

1. (combine-lst-lst nil nil) **→** **NIL**

2. (combine-lst-lst ’(a b c) nil) **→** **NIL**

3. (combine-lst-lst nil ’(a b c)) **→** **NIL**

### Apartado 3:

1. **Función 1:**

*(defun combine-list-of-lsts (lstolsts)*

*(if (null lstolsts)*

*(list nil)*

*(combine-lst-lst-cons (first lstolsts) (combine-list-of-lsts (rest lstolsts))))*

*)*

1. **Función 2:**

*(defun combine-elt-lst-cons (elt lst)*

*(cond ((or (null lst) (null elt))*

*nil)*

*(t*

*(mapcar #'(lambda (x) (cons elt x)) lst)))*

*)*

1. **Función 3:**

*(defun combine-lst-lst-cons (lst1 lst2)*

*(mapcan #'(lambda (x) (combine-elt-lst-cons x lst2)) lst1)*

*)*

La salida para los casos propuestos es:

1. (combine-list-of-lsts ’(() (+ -) (1 2 3 4))) **→** **NIL**

2. (combine-list-of-lsts ’((a b c) () (1 2 3 4))) **→NIL**

3. (combine-list-of-lsts ’((a b c) (1 2 3 4) ())) **→** **NIL**

4. (combine-list-of-lsts ’((1 2 3 4))) **→** **((1) (2) (3) (4))**

5. (combine-list-of-lsts ’(nil)) **→** **NIL**

6. (combine-list-of-lsts nil) **→** (**NIL)**

## Ejercicio 4:

### Apartado 1:

Para este apartado hemos implementado los siguientes grupos de funciones:

1. **Grupo 1:** Funciones para evaluar expresiones que comienzan con una negación.

#### **EVALUAR-NEG-AND**

**PSEUDOCODIGO:**

**Entrada:** fbf (Fórmula bien formada)

**Salida:** fbf (Fórmula bien formada negada)

**Procesamiento:**

Si el resto es null

Devuelve lista que contiene el primer elemento con un not (!) delante

En otro caso crea una lista con un not y el primer elemento y se llama

a si misma con el resto de la lista.

**CÓDIGO:**

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; evaluar-neg-and

;;; Recibe una expresion y niega sus terminos

;;;

;;; INPUT : fbf - Formula bien formada (FBF) a analizar cuyo primer

;;; conector es ! ^

;;; OUTPUT : fbf - Lista con los argumentos atomicos negados

;;;

*(defun* ***evaluar-neg-and*** *(fbf)*

*(if (null (rest fbf))*

*(list (list +not+ (first fbf)))*

*(append (list (list +not+ (first fbf))) (evaluar-neg-and (rest fbf)))))*

**COMENTARIOS:**

Niega cada uno de los términos de la fbf que se le pasa como argumento.

#### **EVALUAR-NEG-OR**

**PSEUDOCODIGO:**

**Entrada:** fbf (Fórmula bien formada)

**Salida:** fbf (Fórmula bien formada negada)

**Procesamiento:**

Si el resto es null

Devuelve lista que contiene el primer elemento con un not (!) delante

En otro caso crea una lista con un not y el primer elemento y se llama

a si misma con el resto de la lista.

**CÓDIGO:**

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; evaluar-neg-or

;;; Recibe una expresion y niega sus términos

;;;

;;; INPUT : fbf - Formula bien formada (FBF) a analizar cuyo primer

;;; conector es ! v

;;; OUTPUT : fbf - Lista con los argumentos atomicos negados

;;;

*(defun* ***evaluar-neg-or*** *(fbf)*

*(if (null (rest fbf))*

*(list (list +not+ (first fbf)))*

*(append (list (list +not+ (first fbf))) (evaluar-neg-or (rest fbf)))))*

**COMENTARIOS:**

Niega cada uno de los términos de la fbf que se le pasa como argumento.

#### **EVALUAR-N-ARY-NEG**

**PSEUDOCODIGO:**

**Entrada:** fbf (Fórmula bien formada)

**Salida:** fbf (Fórmula bien formada negada)

**Procesamiento:**

Si el primer elemento de fbf es V (or)

Devuelve lista que contiene un ^ (and) al principio y lo que devuelve negar-neg-or del resto

Si el primer elemento de fbf es ^ (and)

Devuelve lista que contiene un V (or) al principio y lo que devuelve negar-neg-and del resto

**CÓDIGO:**

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; evaluar-n-ary-neg

;;; Recibe una expresion y evalua a que función

;;; mandar el resto de la fbf dependiendo del

;;; conector que preceda

;;;

;;; INPUT : fbf - Formula bien formada (FBF) a analizar

;;; OUTPUT : fbf - Lista con los argumentos atomicos

;;;

*(defun* ***evaluar-n-ary-neg*** *(fbf)*

*(cond ((eql (first fbf) +or+)*

*(append (list +and+) (evaluar-neg-or (rest fbf))))*

*((eql (first fbf) +and+)*

*(append (list +or+) (evaluar-neg-and (rest fbf)))))*

*nil)*

**COMENTARIOS:**

Dependiendo el conector que tenga manda el resto de fbf a negar-or o negar-and y lo que devuelven estas funciones lo añaden a una lista con el conector opuesto al que tenía la fbf original.

#### **EVALUAR-NEG-COND**

**PSEUDOCODIGO:**

**Entrada:** fbf (Fórmula bien formada)

**Salida:** fbf (Fórmula bien formada negada)

**Procesamiento:**

Si es null

Devuelve nil.

En otro caso aplica la fórmula para negar una condicional.

**CÓDIGO:**

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; evaluar-neg-cond

;;; Recibe una expresion y la niega sabiendo que el conector que la

;; precedia era ! =>

;;;

;;; INPUT : fbf - Formula bien formada (FBF) a analizar cuyo primer

;;; conector es ! =>

;;; OUTPUT : fbf - Lista con la fbf negada

;;;

*(defun* ***evaluar-neg-cond*** *(fbf)*

*(if (null fbf)*

*nil*

*(list +and+ (list +not+ (second fbf)) (first fbf))))*

**COMENTARIOS:**

Aplica la fórmula para negar una expresión condicional.

#### **EVALUAR-NEG-BICOND**

**PSEUDOCODIGO:**

**Entrada:** fbf (Fórmula bien formada)

**Salida:** fbf (Fórmula bien formada negada)

**Procesamiento:**

Aplica la fórmula para negar una bicondicional.

**CÓDIGO:**

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; evaluar-neg-bicond

;;; Recibe una expresion y la niega sabiendo que el conector que la

;; precedia era ! <=>

;;;

;;; INPUT : fbf - Formula bien formada (FBF) a analizar cuyo primer

;;; conector es ! <=>

;;; OUTPUT : fbf - Lista con la fbf negada

;;;

*(defun* ***evaluar-neg-bicond*** *(fbf)*

*(list +or+ (evaluar-neg-cond fbf) (evaluar-neg-cond (list (second fbf) (first fbf)))))*

**COMENTARIOS:**

Aplica la fórmula para negar una expresión bicondicional.

#### **EVALUAR-NOT**

**PSEUDOCODIGO:**

**Entrada:** fbf (Fórmula bien formada)

**Salida:** fbf (Fórmula bien formada negada)

**Procesamiento:**

Si el primer elemento es not (!)

Devuelve el resto de la fbf (doble negación)

Si el primer elemento es un conector n-ario

Manda toda la fbf a la función evaluar-n-ary-neg

Si el primer elemento es <=>

Manda el resto de la fbf a evaluar-neg-bicond

Si el primer elemento es =>

Manda el resto de la fbf a evaluar-neg-cond

En otro caso devuelve la lista con un not (!) delante

**CÓDIGO:**

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; evaluar-not

;;; Recibe una expresion y la evalua

;;;

;;; INPUT : fbf - Formula bien formada (FBF) a analizar cuyo primer

;;; conector es ! y evalua a que funcion la tiene que

;;; mandar dependiendo del siguiente conector.

;;; OUTPUT : list - Lista con la fbf negada

;;;

*(defun* ***evaluar-not*** *(fbf)*

*(cond ((unary-connector-p (first fbf))*

*(rest fbf))*

*((n-ary-connector-p (first fbf))*

*(evaluar-n-ary-neg fbf))*

*((bicond-connector-p (first fbf))*

*(evaluar-neg-bicond (rest fbf)))*

*((cond-connector-p (first fbf))*

*(evaluar-neg-cond (rest fbf)))*

*(t*

*(list +not+ (first fbf)))))*

**COMENTARIOS:**

Recibe una fbf la cual va a mandar a una función de negación dependiendo del primer elemento que tenga.

1. **Grupo 2:** Funciones que convierten las expresiones que comienzan con un conector binario en una expression con conectores n-arios.

#### **EVALUAR-COND**

**PSEUDOCODIGO:**

**Entrada:** fbf (Fórmula bien formada)

**Salida:** fbf (Fórmula bien formada con conectores n-arios)

**Procesamiento:**

Si es null lo que recibe

Devuelve nil

En otro caso devuelve lo que resulta de aplicar la fórmula de conversión de una expresión condicional a una expresión con conectores n-arios

**CÓDIGO:**

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; evaluar-cond

;;; Recibe una expresion y la evalua

;;;

;;; INPUT : fbf - Formula bien formada (FBF) a analizar cuyo primer

;;; conector es =>

;;; OUTPUT : fbf - Lista con la expresion con solo conectores n-arios

;;; NIL - En caso de que los elementos sean vacios o NIL

;;;

*(defun* ***evaluar-cond*** *(fbf)*

*(if (null fbf)*

*nil)*

*(list +or+ (list +not+ (first fbf)) (second fbf)))*

**COMENTARIOS:**

Aplica la fórmula para convertir una expresión condicional en una que solo contenga ands, ors y negaciones.

#### **EVALUAR-BICOND**

**PSEUDOCODIGO:**

**Entrada:** fbf (Fórmula bien formada)

**Salida:** fbf (Fórmula bien formada con conectores n-arios)

**Procesamiento:**

Devuelve lo que resulta de aplicar la fórmula de conversión de una expresión condicional a una expresión con conectores n-arios

**CÓDIGO:**

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; evaluar-cond

;;; Recibe una expresion y la evalua

;;;

;;; INPUT : fbf - Formula bien formada (FBF) a analizar cuyo primer

;;; conector es <=>

;;; OUTPUT : fbf - Lista con la expresion con solo conectores n-arios

;;;

*(defun* ***evaluar-bicond*** *(fbf)*

*(list +and+ (evaluar-cond fbf) (evaluar-cond (list (second fbf) (first fbf)))))*

**COMENTARIOS:**

Aplica la fórmula para convertir una expresión bicondicional en una que solo contenga ands, ors y negaciones.

#### **EVALUAR**

**PSEUDOCODIGO:**

**Entrada:** fbf (Fórmula bien formada)

**Salida:** fbf (Fórmula bien formada con conectores n-arios)

**Procesamiento:**

Si la fbf es un literal

Devuelve la fbf tal cual le llega

Si el primer elemento es not (!)

Manda a evaluar lo que le devuelva la función evaluar-not de lo que esta negado

Si el primer elemento es <=>

Manda a evaluar lo que devuelva el mandar el resto de la fbf a evaluar-bicond

Si el primer elemento es =>

Manda a evaluar lo que devuelva el mandar el resto de la fbf a evaluar-cond

Si el primer elemento es un conector n-ario o un literal

Devuelve una lista con ese conector/literal y lo que devuelva evaluar el resto de las expresiones.

En otro caso devuelve la fbf tal cual.

**CÓDIGO:**

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; evaluar

;;; Recibe una expresion y la evalua

;;;

;;; INPUT : fbf - Formula bien formada (FBF) a analizar

;;; OUTPUT : fbf - Lista la formula bien formada con solo conectores

;;; n-arios.

;;;

*(defun* ***evaluar*** *(fbf)*

*(cond ((literal-p fbf)*

*fbf)*

*((unary-connector-p (first fbf)) ;; !(expresion)*

*(evaluar (evaluar-not (second fbf))))*

*((bicond-connector-p (first fbf)) ;; Bicond*

*(evaluar (evaluar-bicond (rest fbf))))*

*((cond-connector-p (first fbf)) ;; Cond*

*(evaluar (evaluar-cond (rest fbf))))*

*((or (literal-p (first fbf)) (n-ary-connector-p (first fbf)))*

*(cons (first fbf) (mapcar #'(lambda(x) (evaluar x)) (rest fbf))))*

*(t*

*fbf))*

*)*

**COMENTARIOS:**

Va evaluando los elementos de una fórmula bien formada y a través de mandarlos a las diferentes funciones descritas anteriormente consigue devolver una fbf que únicamente contiene conectores n-arios y negaciones.

#### **Ilustramos el funcionamiento de las funciones anteriores con algunos ejemplos:**

## Ejercicio 5:

5.2 Pseudocódigo del bfs:

*BFS (G, s) //Where G is the graph and s is the source node*

*let Q be queue*

*Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked*

*mark s as visited.*

*while ( Q is not empty)*

*//Removing that vertex from queue,whose neighbour will be visited now*

*v = Q.dequeue( )*

*//processing all the neighbours of v*

*for all neighbours w of v in Graph G*

*if w is not visited*

*Q.enqueue( w ) //Stores w in Q to further visit its neighbor*

*mark w as visited.*