Universidad Autónoma de Madrid

Inteligencia Artificial

práctica 1

Nombres	Daniel Tijerina González
	Suemy Inagaki Pinheiro Fagundes
Grupo	2363

Madrid, 27 de Febrero de 2020

1 Ceros de una función

1.1 Problema 1-A

1.1.1 Analise de los Requisitos

- la funcion f debe ser continua y derivable
- Se debe utilizar el método *Newton-Raphson*
- Se debe partir de una semilla x_0
- el algoritmo debe terminar cuando la estimacion x_{n+1} está suficientemente cerca de x* o cuando el algoritmo no converge, alcanzando un numero máximo de iteraciones max-inter.

1.1.2 Bateria de Ejemplos

```
(newton #'sin #'cos 50 2.0) ;ejemplo propuesto en el ejercicio

;ejemplo con error numerico. El valor de (cos pi/2) es 0 (en teoria) pero
;es evaluado como diferente de zero
(newton #'sin #'cos 50 (/ pi 2))
```

1.1.3 Pseudocódigo

```
Inicio Funcion (f, df-dx, max-iter, x0, tol-abs)
    Si tol-abs es null:
        tol-abs <- 0.0001
    Caso contrario:
        aux <- (x0 - f(x0)/df-dx(x0))
        Si el modulo de (aux - x0) es menos de tol-abs:
            aux
        Si max-iter es igual a 0:
            nil
        Caso contrario:
            llamada recursiva:
                funcion(f, df-dx, (max-iter - 1), aux, tol-abs)
Fin Funcion
```

1.1.4 Definición de la Función

```
(defun newton (f df-dx max-iter x0 &optional (tol-abs 0.0001))
  "Zero of a function using the Newton-Raphson method


    INPUT:  f:        function whose zero we wish to find
            df-dx:    derivative of f
            max-iter: maximum number of iterations
```

```
            x0:       initial estimation of the zero (seed)
            tol-abs:  tolerance for conv6trfvergence


    OUTPUT: estimation of the zero of f, NIL if not converged"
  (let ((aux (- x0 (/ (funcall f x0) (funcall df-dx x0)))))
    (cond ((< (abs (- aux x0)) tol-abs) aux)
          ((eq max-iter 0) nil)
          (t (newton f df-dx (- max-iter 1) aux tol-abs)))))
```

1.1.5 Tests

```
>> (newton #'sin #'cos 50 2.0)
3.1415927

>> (newton #'sin #'cos 50 (/ pi 2))
-1.6331239353195368d16 ;era para retornar NIL, pero hay un error numerico
```

1.1.6 Comentarios

Hay un error numerico presentado al evaluar el cosino de $\pi/2$ El resultado de la evaluacion debería ser 0, pero es un número muy pequeño, cerca de 10^{-17}.

1.2 Problema 1-B

1.2.1 Analise de los Requisitos

- la funcion f debe ser continua y derivable
- Se debe utilizar el método *Newton-Raphson*
- Se debe partir de una lista de semillas
- el algoritmo debe terminar cuando la estimacion x_{n+1} está suficientemente cerca de x* o cuando el algoritmo no converge, alcanzando un numero máximo de iteraciones max-inter.

1.2.2 Bateria de Ejemplos

```
;ejemplo propuesto en el enunciado
(newton-all #'sin #'cos 50 (mapcar #'eval '((/ pi 2) 1.0 2.0 4.0 6.0)))

;ejemplo donde seeds es una lista vacia:
(newton-all #'sin #'cos 50 '())

;ejemplo con la condicion de parada = solamente dos elementos en la lista
(newton-all #'sin #'cos 50 (mapcar #'eval '((/ pi 2) 1.0 2.0)))
```

1.2.3 Pseudocódigo

```
Inicio Funcion (f df-dx max-iter seeds tol-abs)
    Si tol-abs es null:
        tol-abs <- 0.0001
        aux <- llama newton donde los parametros son:
            parametro1 <- f
            parametro2 <- df-dx
            parametro3 <- max-iter
            parametro4 <- primer elemento de seeds
            parametro5 <- tol-abs
    Si seed es vacio:
        nil
    Si seeds solo tiene un elemento:
       retorna una lista con aux
    Caso contrario:
        crea una lista donde el primer elemento es aux
        y el segundo elemento es la llamada recursiva:
            parametro1 <- f
            parametro2 <- df-dx
            parametro3 <- max-iter
            parametro4 <- seeds sin el primer elemento
            parametro5 <- tol-abs
Fin Funcion
```

1.2.4 Código de la Función

```
(defun newton-all (f df-dx max-iter seeds &optional (tol-abs 0.0001))
  "Zeros of a function using the Newton-Raphson method


    INPUT:  f:        function whose zero we wish to find
            df-dx:    derivative of f
            max-iter: maximum number of iterations
            seeds:    list of initial estimations of the zeros
            tol-abs:  tolerance for convergence


    OUTPUT: list of estimations of the zeros of f"
    (cond ((null seeds) nil)
      ((null (cdr seeds)) (cons (newton f df-dx max-iter (car seeds) tol-abs) '()))
      (t (cons (newton f df-dx max-iter (car seeds) tol-abs) (newton-all f
      df-dx max-iter (cdr seeds) tol-abs)))))
```

1.2.5 Tests

```
;ejemplo propuesto en el enunciado
>> (newton-all #'sin #'cos 50 (mapcar #'eval '((/ pi 2) 1.0 2.0 4.0 6.0)))
(-1.6331239353195368d16 0.0 3.1415927 3.1415927 6.2831855)
;ejemplo donde seeds es una lista vacia:
>> (newton-all #'sin #'cos 50 '())
NIL
;ejemplo con la condicion de parada = solamente dos elementos en la lista
>> (newton-all #'sin #'cos 50 (mapcar #'eval '((/ pi 2) 1.0 2.0)))
(-1.6331239353195368d16 0.0 3.1415927)
```

1.2.6 Comentarios

La funcion apresenta error numerico como consecuencia de la funcion newton

2 Combinación de Listas

2.1 Problema 2-A

2.1.1 Analise de los Requisitos

- La funcion debe combinar el elemento recibido con cada elemento de la lista recibida.
- Cuando la lista recibida for vacia, la funcion debe retornar NIL.
- El elemento recibido no puede ser nulo, pues el numero de parametros estará incorrecto.
- Todas las combinaciones debem resultar en parejas.
- Cada pareja debe estar en una lista.
- Todas las listas de parejas debem estar dentro de una unica lista para ser retornada.

2.1.2 Bateria de Ejemplos

```
(combine-elt-lst '1 '())  ;lista vacia
(combine-elt-lst '1 '(A B C))  ;lista no vacia
(combine-elt-lst 'a '(1 2 3))  ;ejemplo propuesto
```

2.1.3 Pseudocódigo

```
Inicio Funcion (elt lst)
    Para cada elemento de lst:
        arg <- elemento de lst
        cria una lista (elt arg)

    Retorna una lista con todas as listas criadas
Fim funcion
```

2.1.4 Código de la Función

```
(defun combine-elt-lst (elt lst)
  "Combines an element with all the elements of a list


    INPUT:  elt: element
            lst: list


    OUTPUT: list of pairs, such that
            the first element of the pair is elt.
            the second element is an element from lst"

  (mapcar #'(lambda (arg) (list elt arg)) lst))
```

2.1.5 Tests

```
>> (combine-elt-lst '1 '())  ;lista vacia
NIL

>> (combine-elt-lst '1 '(A B C))  ;lista no vacia
((1 A) (1 B) (1 C))

>> (combine-elt-lst 'a '(1 2 3))  ;ejemplo propuesto
((A 1) (A 2) (A 3))
```

2.1.6 Comentarios

La funcion combina un elemento con una lista. Si la lista es null, retorna nil

2.2 Problema 2-B

2.2.1 Analise de los Requisitos

- La funcion debe combinar cada elemento en la lista 1 con cada elemento de la lista 2
- Si una o las dos listas son vacias, la funcion retorna NIL
- Todas las combinaciones debem resultar en parejas.
- Cada pareja debe estar en una lista.
- Todas las listas de parejas debem estar dentro de una unica lista para ser retornada.

2.2.2 Bateria de Ejemplos

```
(combine-lst-lst '(1) '(a))    ;1 elemento en cada lista

;una de las listas vacia (análogo para la segunda lista vacia)
(combine-lst-lst '() '(a b c))

(combine-lst-lst '(1 2) '(a b))  ;mas de 1 elemento en las listas
```

2.2.3 Pseudocódigo

```
Inicio Funcion (lst1 lst2)
    Si lst1 no es null:
        Para cada elemento de lst2:
            arg <- elemento de lst2
            elt <- primer elemento de lst1
            cria una lista (elt arg)
        llamada recursiva de la funcion:
            parametro_1 <- lst1 sin el primer elemento
            parametro_2 <- lst2
    Retorna todas las listas criadas dentro de una unica lista
Fim funcion
```

2.2.4 Codigo de la Función

```
(defun combine-lst-lst (lst1 lst2)
  "Producto Cartesiano entre dos listas

    INPUT:  lst1: list 1
            lst2: list 2


    OUTPUT: Lista de productos cartesianos
            entre la lista 1 y la lista 2"

    (unless (null lst1)
        (append
            (combine-elt-lst (car lst1) lst2)
            (combine-lst-lst (cdr lst1) lst2))))
```

2.2.5 Tests

```
>> (combine-lst-lst '(1) '(a))    ;1 elemento en cada lista
((1 A))
>> (combine-lst-lst '() '(a b c)) ;una de las listas vacia
NIL
```

```
>> (combine-lst-lst '(1 2) '(a b))  ;mas de 1 elemento en las dos listas
((1 A) (1 B) (2 A) (2 B))
```

2.2.6 Comentarios

la funcion es recursiva y la condicion de parada es cuando lst1 = null

2.3 Problema 2-C

2.3.1 Analise de los Requisitos

- La funcion debe combinar cada elemento en cada lista con los elementos de todas las otras listas.
- Si la lista es vacia, debe retornar NIL.
- Si la lista tiene solamente un elemento, debe retornar la propria lista.
- Todas las combinaciones debem resultar en una sublista de tamaño N, donde N es el tamaño de la lista original.
- Todas las sublistas de tamaño N debem estar dentro de una unica lista para ser retornada.

2.3.2 Bateria de Ejemplos

anade-elemento

```
(anade-elemento 'a '((1 2))) ;una unica lista en la lista
(anade-elemento 'a '((1) (2))) ;mas de una lista en la lista
(anade-elemento 'a '()) ;intentar anadir a una lista vacia
```

combine-lsts-lst

```
(combine-lsts-lst '(1) '((a))) ;cada lista con un solo elemento

;lista 1 con un elemento y lista 2 con mas de un elemento
(combine-lsts-lst '(1) '((a) (b)))

;lista 1 con mas de un elemento y lista 2 con un solo elemento
(combine-lsts-lst '(1 2) '((a)))

;las dos listas con mas de un elemento
(combine-lsts-lst '(1 2) '((a) (b)))

;condicion de parada, lst1 = null
(combine-lsts-lst '() '((a)))
```

combine-list-of-lsts

```
;lisya vacia
(combine-list-of-lsts '())

;lista con una lista
(combine-list-of-lsts '((a)))

;lista con dos listas (condicion de parada)
(combine-list-of-lsts '((a b c) (1 2 3 4 5)))

;lista con mas de dos listas (recursion)
(combine-list-of-lsts '((a 2) (b 1) (c 3)))
```

2.3.3 Pseudocódigo

anade-elemento

```
Inicio Funcion (elt lst)
    Para cada sublista en lst:
        arg <- una sublista de lst
        cria una lista donde el primer elemento es elt
        y los otros son los elementos de arg.
    Retorna todas las listas criadas dentro de una unica lista
Fim funcion
```

combine-lsts-lst

```
Inicio Funcion (lst1 lst2)
    Si lst1 no es null:
        Para cada elemento de lst2:
            arg <- elemento de lst2
            elt <- primer elemento de lst1
            cria una lista (elt arg)
        llamada recursiva de la funcion:
            parametro_1 <- lst1 sin el primer elemento
            parametro_2 <- lst2
    Retorna todas las listas criadas dentro de una unica lista
Fim funcion
```

combine-list-of-lsts

```
Inicio Funcion (lolsts)
    si lolsts tiene solamente un elemento:
        retorna lolsts
    si lolsts tiene solamente dos elementos:
        hace la combinacion de esas dos listas
    En todos los otros casos:
        combina la primera lista de lolsts con la llamada recursiva:
            parametro_1 <- lolsts sin el primer elemento
```

```
    Retorna todas las listas criadas dentro de una unica lista
Fim funcion
```

2.3.4 Código de las Funciones Auxiliares

```
(defun anade-elemento (elt lista)

    "Anade un elemento a cadaj lista

    INPUT:  elt: element
            lista: list of lists

    OUTPUT: lista de listas donde en cada lista el elt se agrega
    al principio"

(mapcar #'(lambda (arg) (cons elt arg)) lista))

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

(defun combine-lsts-lst (lst1 lst2)

    "Produto cartesiano de uma lista de listas com outra lista

    INPUT:  lst1: lista de atomos
            lst2: lista de listas

    OUTPUT: Lista com o produto cartesiano"

(unless (null lst1)
    (append
        (anade-elemento (car lst1) lst2)
        (combine-lsts-lst (cdr lst1) lst2))))
```

2.3.5 Código de la Funcion Principal

```
(defun combine-list-of-lsts (lolsts)

  "Combinations of N elements, each of wich

   INPUT:  lstolsts: list of N sublists (list1 ... listN)

   OUTPUT: list of sublists of N elements, such that in each sublist the
   first element is from list1 the second element is from list 2 ... the
   Nth element is from list N"

(cond
```

```
    ((null (cdr lolsts)) lolsts)
    ((null (cddr lolsts)) (combine-lst-lst (car lolsts) (cadr lolsts)))
    (t (combine-lsts-lst (car lolsts)(combine-list-of-lsts (cdr lolsts))))))
```

2.3.6 Tests

anade-elemento

```
>> (anade-elemento 'a '((1 2))) ;una unica lista en la lista
((A 1 2))
>> (anade-elemento 'a '((1) (2))) ;mas de una lista en la lista
((A 1) (A 2))
>> (anade-elemento 'a '()) ;intentar anadir a una lista vacia
NIL
```

combine-lsts-lst

```
;cada lista con un solo elemento
>> (combine-lsts-lst '(1) '((a)))
((1 A))
;lista 1 con un elemento y lista 2 con mas de un elemento
>> (combine-lsts-lst '(1) '((a) (b)))
((1 A) (1 B))
;lista 1 con mas de un elemento y lista 2 con un solo elemento
>> (combine-lsts-lst '(1 2) '((a)))
((1 A) (2 A))
;las dos listas con mas de un elemento
>> (combine-lsts-lst '(1 2) '((a) (b)))
((1 A) (1 B) (2 A) (2 B))
;condicion de parada, lst1 = null
>> (combine-lsts-lst '() '((a)))
NIL
```

combine-list-of-lsts

```
;lista con una lista
>> (combine-list-of-lsts '((a)))
((A))

;lista con dos listas (condicion de parada)
>> (combine-list-of-lsts '((a b c) (1 2 3 4 5)))
((A 1) (A 2) (A 3) (A 4) (A 5) (B 1) (B 2) (B 3) (B 4) (B 5)
(C 1) (C 2) (C 3) (C 4) (C 5))

;lista con mas de dos listas (recursion)
>> (combine-list-of-lsts '((a 2) (b 1) (c 3)))
((A B C) (A B 3) (A 1 C) (A 1 3) (2 B C) (2 B 3) (2 1 C) (2 1 3))
```

2.3.7 Comentários

La funcion anade-elemento fue criada para auxiliar la funcion "combine-lsts-lst". Ella añade un elemento al principio de la lista. Utilizaremos esta funcion al implementar la funcion combine-lsts-lst.

La funcion combine-lsts-lst combina una lista de listas con una lista de átomos. Es una funcion auxiliar a la funcion combine-list-of-lsts y tiene una implementacion recursiva:

- caso base: lst1 = null
- recursion: combine-lsts-lst (cdr lst1) lst2

La funcion combine-list-of-lsts combina los elementos de las N sublistas que pertenecen a lolsts. Tiene una implementacion recursiva:

- caso base: cuando la lista lolsts tiene solamente 2 sublistas
- recursion: pasar para la funcion combine-lsts-lst la primera sublista de lolsts y la llamada a la funcion combine-list-of-lsts con (cdr lolsts).

Tuvimos que tratar a parte el caso de cuando la lista lolsts tiene solamente un elemento, por eso utilizamos "cond".

Así, el cond evalua los tres posibles casos:

1. el caso en que la lista de entrada solo tiene una lista
2. el caso en que la lista de entrada tiene dos listas (caso base de la recursion)
3. el caso en que la lista de entrada tiene mas de dos listas
   recursion, (combine-list-of-lsts (cdr lolsts))

Caso las dos primeras opciones se evaluen como NIL, obligatoriamente tiene que entrar en el tercer caso, enconces la expresion booleana del tercer caso es t que es siempre verdadero.

3 Producto escalar y distancia euclídea

3.1 Producto Escalar

3.1.1 Analise de los Requisitos

- La funcion debe recibir dos vectores
- Los vectores deben tener el mismo tamaño
- La funcion retorna un numero.
- Si los dos vectores son vacios, la funcion retorna NIL.

3.1.2 Bateria de Ejemplos

```
;dos listas con un solo elemento cada
(scalar-product '(1) '(2))

;produto escalar entre una lista vacia y una no vacia
(scalar-product '() '(1))

;produto escalar entre dos listas con mas de un elemento
(scalar-product '(1 1 3) '(3 4 6))
```

3.1.3 Pseudocódigo

```
Inicio Funcion
    x <- parametro_1
    y <- parametro_2

    soma <- 0

    Para cada elemento de X y de Y:
        calcular el respectivo producto entre ellos.
        somar el producto encontrado con soma

    Retornar soma
Fim funcion
```

3.1.4 Definicion de la Funcion Auxiliar

```
(defun soma (lst)
"Calcula la suma de los elementos de una lista

   INPUT:  lst: list

   OUTPUT: suma de los elementos de la lista

   NOTES: implementacion recursiva"

(if (null lst) 0 (+ (car lst) (soma (cdr lst)))))
```

3.1.5 Definicion de la Funcion Principal

```
(defun scalar-product (x y)
  "Calculates the scalar product of two vectors

   INPUT:  x: vector, represented as a list
           y: vector, represented as a list
```

```
   OUTPUT: scalar product between x and y


   NOTES:
        * Implemented with mapcar"

  (unless (or (null x) (null y))
    (soma (mapcar #'(lambda (arg1 arg2) (* arg1 arg2)) x y))))
```

3.2 Norma Euclídea

3.2.1 Analise de los Requisitos

- La funcion debe recibir un vector
- La funcion retorna un numero.
- Si el vector es vacio, la funcion retorna NIL.

3.2.2 Bateria de Ejemplos

```
(euclidean-norm '()) ;lista vazia
(euclidean-norm '(1)) ;lista con un elemento
(euclidean-norm '(1 1 1)) ;lista con mas de un elemento
```

3.2.3 Pseudocódigo

```
Inicio Funcion
    x <- parametro_1

    soma <- 0

    Para cada elemento de X:
        elt <- elemento de x
        calcular el valor de elt^2.
        somar el producto encontrado con soma

    Retornar srqt de soma
Fim funcion
```

3.2.4 Definicion de la Funcion

```
(defun euclidean-norm (x)
  "Calculates the euclidean (l2) norm of a vector

    INPUT:  x: vector, represented as a list
```

```
    OUTPUT: euclidean norm of x"

  (unless (null x)
    (sqrt (soma (mapcar #'(lambda (a) (* a a)) x)))))
```

3.3 Distancia Euclídea

3.3.1 Analise de los Requisitos

- La funcion recibe dos vectores de mismo tamaño
- Si uno o los dos vectores son vacios, la funcion retorna NIL
- Caso contrario la funcion retorna un numero.

3.3.2 Bateria de Ejemplos

```
(euclidean-distance '(1) '(0)) ;dos listas con un elemento
(euclidean-distance '(1 1 1) '(0 1 3)) ;dos listas con mas de un elemento
(euclidean-distance '() '(1 2 3)) ;intentar hacer con una lista vacia
```

3.3.3 Pseudocódigo

```
Inicio Funcion
    x <- parametro_1
    y <- parametro_2

    Para cada elemento de X y Y:
        calcular la diferencia entre los respectivos elementos
        añadir este valor a un vector
    calcular la norma del vector resultante

    Retornar la norma encontrada
Fim funcion
```

3.3.4 Definicion de la Funcion

```
(defun euclidean-distance (x y)
  "Calculates the euclidean (l2) distance between two vectors

    INPUT: x: vector, represented as a list
           y: vector, represented as a list


    OUTPUT: euclidean distance between x and y"

    (euclidean-norm (mapcar #'- x y)))
```

3.4 Tests

soma

```
>> (soma '()) ;lista vazia ;caso base
0
>> (soma '(1)) ;lista con solamente un elemento
1
>> (soma '(1 2 3) ;lista con mas de un elemento
6
```

scalar-product

```
>> (scalar-product '(1) '(2)) ;dos listas con un solo elemento cada
2
;produto escalar entre una lista vacia y una no vacia
>> (scalar-product '() '(1))
0

;produto escalar entre dos listas con mas de un elemento
>> (scalar-product '(1 1 3) '(3 4 6))
25
```

euclidean-norm

```
>> (euclidean-norm '())  ;lista vazia
0.0
>> (euclidean-norm '(1)) ;lista con un elemento
1.0
>> (euclidean-norm '(1 1 1)) ;lista con mas de un elemento
1.7320508
```

euclidean-distance

```
>> (euclidean-distance '(1) '(0)) ;dos listas con un elemento
1.0
;dos listas con mas de un elemento
>> (euclidean-distance '(1 1 1) '(0 1 3))
2.236068
```

4 Similitud Coseno y Distancia Angular

4.1 Cosine Similarity

4.1.1 Analise de los Requisitos

- los vectores deben tener el mismo tamaño
- la norma de los dos vectores no debem ser 0
- la funcion retorna un numero entre 0 y 1

4.1.2 Bateria de Ejemplos

```
; cuando o la norma de x o la norma de y es 0, nos es posible evaluar
; el cosine-similarity de estes vectores
(cosine-similarity '(1 1 1) '(0 0 0))

; vectores son identicos.
; en este caso ocurre un error numerico que interfere en el resultado
; de funciones que dependen de esta.
(cosine-similarity '(1 1) '(1 1))

; los vectores son identicos
(cosine-similarity '(1 1 1) '(1 1 1))
```

4.1.3 Pseudocódigo

```
Inicio Funcion
    x <- parametro_1
    y <- parametro_2

    calcular la norma de X
    calcular la norma de Y
    Si la norma de X y la norma de Y no son NIL:
        calcular el producto escalar entre Y y Y
        calcular el producto entre las normas de X y Y
        retornar la division los dos resultados
Fim funcion
```

4.1.4 Código de la Funcion

```
(defun cosine-similarity (x y)
  "Calculates the cosine similarity between two vectors


    INPUT:  x: vector, representad as a list
            y: vector, representad as a list


    OUTPUT: cosine similarity between x and y


    NOTES:
       * Evaluates to NIL (not defined)
         if at least one of the vectors has zero norm.
       * The two vectors are assumed to have the same length"
```

```
    (unless (or (= 0 (euclidean-norm x)) (= 0 (euclidean-norm y)))
        (/ (scalar-product x y) (* (euclidean-norm x) (euclidean-norm y)))))
```

4.1.5 Tests

```
; cuando o la norma de x o la norma de y es 0, nos es posible evaluar
; el cosine-similarity de estes vectores
>> (cosine-similarity '(1 1 1) '(0 0 0))
NIL

; vectores son identicos.
; en este caso ocurre un error numerico que interfere en el resultado
; de funciones abajo.
>> (cosine-similarity '(1 1) '(1 1))
1.0000001

; los vectores son identicos
>> (cosine-similarity '(1 1 1) '(1 1 1))
1.0
```

4.2 Distancia Angular

4.2.1 Analise de los Requisitos

- Los dos vectores deben tener el mismo tamaño
- Los dos vectores no debem tener la norma 0 o NIL
- La funcion retorna un numero entre 0 y 1

4.2.2 Bateria de Ejemplos

```
(angular-distance '(1) '(1))

; los vectores forman 90º
(angular-distance '(1 0) '(0 1))

;;; los vectores son identicos (deberia ser 0). En este caso ha tenido
;;;         un error numerico pues cuando evaluamos (cosino-similarity
;;; '(1 1) '(1 1)) es retornado el valor 1.000001, un poco más que 1.
;;; Como el dominio de la funcion acos es [-1,1] y este valor esta
;;; fuera del dominio, hay este error en la evaluacion de
;;; angular-distance de estes vectores.

(angular-distance '(1 1) '(1 1))
```

4.2.3 Pseudocódigo

```
Inicio Funcion
    x <- parametro_1
    y <- parametro_2

    calcular la similitud de cosino entre X y Y
    si la similitud de cosino entre X y Y no es null:
        a <- acos del cosino obtido
        retornar la division de a por Pi
Fim funcion
```

4.2.4 Código de la Funcion

```
(defun angular-distance (x y)
  "Calculates the angular distance between two vectors


   INPUT:  x: vector, representad as a list
           y: vector, representad as a list


   OUTPUT: cosine similarity between x and y


   NOTES:
      * Evaluates to NIL (not well defined)
        if at least one of the vectors has zero norm.
      * The two vectors are assumed to have the same length"

  (unless (null (cosine-similarity x y))
      (/ (acos (cosine-similarity x y)) pi)))
```

4.2.5 Tests

```
>> (angular-distance '(1) '(1))
0.0

; los vectores forman 90º
>> (angular-distance '(1 0) '(0 1))
0.5

;;; los vectores son identicos (deberia ser 0). En este caso ha tenido
;;;         un error numerico pues cuando evaluamos (cosino-similarity
;;; '(1 1) '(1 1)) es retornado el valor 1.000001, un poco más que 1.
;;; Como el dominio de la funcion acos es [-1,1] y este valor esta
;;; fuera del dominio, hay este error en la evaluacion de
```

```
;;; angular-distance de estes vectores.

>> (angular-distance '(1 1) '(1 1))
#C(0.0d0 1.5542473058510512d-4)
```

4.2.6 Comentários

esa funcion retorna NIL cuando cosine-similariy retorna NIL. Teste necesario para la funcion nearest neighbor.

5 Categorización de Textos

5.1 Analise de los Requisitos

- La funcion debe recibir una funcion de similitud
- La funcion debe calcular la similitud de acuerdo con la funcion recibida
- La funcion debe recibir un valor opcional entre 0 y 1 (threshold).
- Si el parametro threshold no es pasado, el valor es definido como 0.
- La funcion debe retornar solamente los vectores con similitud superior a la similitud dada (o a la definida como padron, caso no la haya recibido en el parametro)
- La funcion debe utilizar "remove-if"y "sort"
- Los vectores resulutantes debem aparecer en orden do más grande para el más pequeño.
- Caso no sea posible calcular la similitud de un vector, él debe ser ignorado.

5.2 Bateria de Ejemplos

```
(calcula-sim-dis '((1 1 1)) '(1 1 1) #'cosine-similarity)
(calcula-sim-dis '((1 1 1) (0 0 0)) '(1 1 1) #'cosine-similarity)
(calcula-sim-dis '((1 0 1) (1 1 1) (1 0 0)) '(1 1 2) #'cosine-similarity)
(calcula-sim-dis '((1 1 1)) '(1 1 1) #'angular-distance)
(calcula-sim-dis '((1 1 1) (0 0 0)) '(1 1 1) #'angular-distance)
(calcula-sim-dis '((1 0 1) (1 1 1) (1 0 0)) '(1 1 2) #'angular-distance)

;no hay vector con similitud superior
(select-vectors '((1 1 1)) '(1 0 0) #'cosine-similarity 0.6)

;sin pasar el parametro opcional
(select-vectors '((1 1 1)) '(1 0 0) #'cosine-similarity)

;los vectores aparecen en la ordem pedida
(select-vectors '((1 1 1) (1 0 0) (1 0 1)) '(1 0 0) #'cosine-similarity 0.6)
```

5.3 Pseudocódigo

calcula-sim-dis

```
Inicio Funcion
    lst-vectors <- parametro_1
    test-vector <- parametro_2
    fn <- parametro_3

    si lst-vectors no es null:
        aux1 <- primer elemento de lst-vectors
        construye una lista donde el primer elemento
        es la lista :
            donde el primer elemento
            es aux1 y el segundo elemento es el resultado
            de la funcion fn entre aux1 y test-vector.
        y el segundo elemento es la llamada recursiva donde:
            parametro_1 <- lst-vectors sin el primer elemento
            parametro_2 <- test-vector
            parametro_3 <- fn
    Retorna la lista resultante.
Fim funcion
```

select-vector

```
Inicio Funcion
    lst-vectors <- parametro_1
    test-vector <- parametro_2
    similarity-fn <- parametro_3

    si parametro_4 = NIL:
        threshold <- 0
    caso contrario:
        threshold <- parametro_4

    ls <- calcula-sim-dis:
        parametro_1 <- lst-vectors
        parametro_2 <- test-vector
        parametro_3 <- similarity-fn

    Para cada elemento de ls:
        arg <- elemento de ls
        si el segundo elemento de arg = NIL o
        el segundo elemento de arg < threshold:
            remove el elemento de ls
    ordena ls en orden decrecente del segundo elemento de cada elemento de ls.
    retorna ls
Fim funcion
```

5.4 Codigo de Funciones Auxiliares

```
(defun calcula-sim-dis (lst-vectors test-vector fn)
  "Calcula la similaridad o la distancia entre cada vector de
   la lista de vetores y el vector teste. Es una funcion auxiliar
   para la funcion select-vectors y nearest-neighbor.

   INPUT: lst-vectors:   list of vectors
          test-vector:   test vector, representad as a list
          similarity-fn: reference to a similarity o a distance function

   OUTPUT: lista de parejas donde el primer elemento de cada pareja
           es el vector y el segundo elemento es la similaridad o la
           distancia.

   NOTES: implementacion recursiva"

(unless (null lst-vectors)
    (cons
        (cons
            (car lst-vectors)
            (list (funcall fn test-vector (car lst-vectors))))
        (calcula-sim-dis (cdr lst-vectors) test-vector fn))))
```

5.5 Código de la Funcion

```
(defun select-vectors
    (lst-vectors test-vector similarity-fn &optional (threshold 0))
    "Selects from a list the vectors whose similarity to a
     test vector is above a specified threshold.
     The resulting list is ordered according to this similarity.

     INPUT:  lst-vectors:   list of vectors
             test-vector:   test vector, representad as a list
             similarity-fn: reference to a similarity function
             threshold:     similarity threshold (default 0)

     OUTPUT: list of pairs. Each pair is a list with
             a vector and a similarity score.
             The vectors are such that their similarity to the
             test vector is above the specified threshold.
             The list is ordered from larger to smaller
             values of the similarity score

     NOTES:
        * Uses remove-if and sort"
```

```
(sort
    (copy-list
        (remove-if
            #'(lambda (x) (or (null (second x)) (< (second x) threshold)))
            (calcula-sim-dis
                lst-vectors test-vector similarity-fn)))
    #'> :key #'second))
```

5.6 Tests

```
>> (calcula-sim-dis '((1 1 1)) '(1 1 1) #'cosine-similarity)
(((1 1 1) 1.0))

>> (calcula-sim-dis '((1 1 1) (0 0 0)) '(1 1 1) #'cosine-similarity)
(((1 1 1) 1.0) ((0 0 0) NIL))

>> (calcula-sim-dis '((1 0 1) (1 1 1) (1 0 0)) '(1 1 2) #'cosine-similarity)
(((1 0 1) 0.8660254) ((1 1 1) 0.942809) ((1 0 0) 0.40824828))

>> (calcula-sim-dis '((1 1 1)) '(1 1 1) #'angular-distance)
(((1 1 1) 0.0d0))

>> (calcula-sim-dis '((1 1 1) (0 0 0)) '(1 1 1) #'angular-distance)
(((1 1 1) 0.0d0) ((0 0 0) NIL))

>> (calcula-sim-dis '((1 0 1) (1 1 1) (1 0 0)) '(1 1 2) #'angular-distance)
(((1 0 1) 0.1666666713045892d0) ((1 1 1) 0.10817350043503084d0)
((1 0 0) 0.36613976572112517d0))

;no hay vector con similaridad superior
>> (select-vectors '((1 1 1)) '(1 0 0) #'cosine-similarity 0.6)
NIL

;sin pasar el parametro opcional
>> (select-vectors '((1 1 1)) '(1 0 0) #'cosine-similarity)
(((1 1 1) 0.57735026))

;los vectores aparecen en la ordem pedida
>> (select-vectors '((1 1 1) (1 0 0) (1 0 1)) '(1 0 0) #'cosine-similarity 0.6)
(((1 0 0) 1.0) ((1 0 1) 0.70710677))
```

5.7 Comentários

En la funcion calcula-sim-dis todavia no es hecha la separacion entre vectores validos o no para calcular la similaridad o la distancia, entonces cuando pasamos el vector '(0 0 0), con norma = 0, la funcion retorna la similaridad o la distancia de este vector como NIL.

En la funcion Lambda de select-vectors tuvimos que poner (or (null (second x)) (< (second x) threshold)) pues cuando no era posible calcular el cosine-similarity entre el vector de lst-vectors y test-vector, ocurria un error en la comparacion (< (second x) threshold)), ya que no es posible comparar NIL con un NUMBER.

6 Vecino Más Próximo

6.1 Analise de los Requisitos

- La funcion debe ignorar lo vectores que no sean posible calcular la distancia.
- La implementacion de la funcion es recursiva
- la funcion debe retornar el vector cuya distancia hasta el vector de teste es la menor.
- Caso no haya un vector que sea posible calcular la distancia, la funcion retorna NIL
- La funcion utiliza la funcion calcula-sim-dis como auxiliar.

6.2 Bateria de Ejemplos

```
;con solo un vector en la lista
(nearest-neighbor '((1 1 1)) '(10 10 10) #'angular-distance)

(nearest-neighbor '((1 1 1) (1 0 0) (2 3 4)) '(1 0 0) #'angular-distance)

; no es posible calcular la distancia angular entre un vetor
; nulo  y otro vector
(nearest-neighbor '((0 0 0)) '(1 0 0) #'angular-distance)

; el vector cuja distancia angular no es posible calcular (vector nulo)
; fue ignorado
(nearest-neighbor '((1 0 0) (0 0 0)) '(1 0 0) #'angular-distance)
```

6.3 Pseudocódigo

```
Inicio Funcion
    lst-vectors <- parametro_1
    test-vector <- parametro_2
    distance-fn <- parametro_3

    ls <- calcula-sim-dis:
        parametro_1 <- lst-vectors
        parametro_2 <- test-vector
        parametro_3 <- distance-fn

    Para cada elemento de ls:
```

```
        arg <- elemento de ls
        si el segundo elemento de arg = NIL
            remove el elemento de ls
    ordena ls en orden crecente del segundo elemento de cada elemento de ls.
    retorna el primer elemento de ls ordenado.
Fim funcion
```

6.4 Código de la Funcion

```
(defun nearest-neighbor (lst-vectors test-vector distance-fn)
  "Selects from a list the vector that is closest to the
   reference vector according to the specified distance function

   INPUT:  lst-vectors:   list of vectors
           ref-vector:    reference vector, represented as a list
           distance-fn:   reference to a distance function

   OUTPUT: List formed by two elements:
           (1) the vector that is closest to the reference vector
               according to the specified distance function
           (2) The corresponding distance value.


   NOTES:
      * The implementation is recursive
      * It ignores the vectors in lst-vectors for which the
        distance value cannot be computed."

(car
    (sort (copy-list
        (remove-if #'(lambda (x) (null (second x)))
            (calcula-similaridad-distancia
                lst-vectors test-vector distance-fn)))
    #'< :key #'second)))
```

6.5 Tests

```
;con solo un vector en la lista
>> (nearest-neighbor '((1 1 1)) '(10 10 10) #'angular-distance)
((1 1 1) 1.0990189218125593d-4)

>> (nearest-neighbor '((1 1 1) (1 0 0) (2 3 4)) '(1 0 0) #'angular-distance)
((1 0 0) 0.0d0)

; no es posible calcular la distancia angular entre un vetor nulo
; y otro vector
```

```
>> (nearest-neighbor '((0 0 0)) '(1 0 0) #'angular-distance)
NIL

; el vector cuja distancia angular no es posible calcular (vector nulo)
; fue ignorado
>> (nearest-neighbor '((1 0 0) (0 0 0)) '(1 0 0) #'angular-distance)
((1 0 0) 0.0d0)
```

7 Problema 7

7.1 Analise de Requisitos

- La base de conocimiento se representa como una lista de sublistas
- El primer elemento de lista es una lista de símbolos correspondientes a los literales positivos que aparecen en el antecedente de una regla o NIL en caso de se trate de un hecho.
- El segundo es el símbolo del literal positivo que aparece en el consecuente de la regla.
-

7.2 Bateria de Ejemplos

```
;;; EJEMPLOS
(backward-chaining 'Q  '((NIL A) (NIL B) ((P) Q) ((L M) P) ((B L) M) ((A P) L)
((A B) L)))
(backward-chaining 'Q '((NIL A) (NIL B) ((A B) C) ((A B) D) ((C D) Q)))
(backward-chaining 'Q '((NIL A) (NIL B) ((A B) C) ((A B) D) ((C E) Q)))
(backward-chaining 'Q '((NIL A) (NIL B) ((A E) C) ((A B) D) ((C D) Q)))
; caso base
(backward-chaining nil '((NIL A) ))
```

7.3 Pseudocodigo

```
Inicio function backward_chaining_aux(Goal lst-rules pending-goals)
    Si Goals = Nil:
        retorna True
    Caso contrario
        si goal está en pending-goal:
            retorna False
        Caso Contrario
            Lista-aux = Todas las listas de lst-rules que tienem Goal
            como segundo elemento

            Sea L[i] una lista que pertenence a Lista-aux. caso haga
```

```
                alguna L[i] tal que todos los elementos dentro de la
                primeira lista de L[i] puedan ser escritos a partir de la
                base de conocimiento, o sea, llamando recursivamente la
                funcion, la funcion retorna True
    Fin Funcion

    Inicio funcion backward_chaining ( Goal lst-rules)
        Backward_chaining_aux(goal lst-rules pending Nil)
    Fin Funcion Backward_chaining
```

7.4 Codigo de la Funcion

```
(defun busca-elemento (elt lst)
  "Funcion que retorna una lista con todas las reglas o hechos
  que posuen ELT como segundo busca-elemento

  INPUT: elt: un elemento
         lst: base de conocimiento

  OUTPUT: una lista con todas las reglas o hechos
  que posuen ELT como segundo busca-elemento"
  (if (null lst) NIL
      (if (eql elt (second (first lst)))
      (cons (first lst)
            (busca-elemento elt (rest lst)))
            (busca-elemento elt (rest lst))
      )
  )
)

(defun aux-some (lst f goal lst-rules pending-goals g)
  "Funcion auxiliar que evalua si hay alguna ocurrencia de T al
  aplicar la funcion f
  INPUT: lst: lista
         f: funcion auxiliar
         goal: elemento que esta buscando
         lst-rules: base de conocimiento
         pending-goals: casos pendientes de las
           otras llamadas recursivas.
         g: funcion backward-chaining-aux
  OUTPUT T o NIL"
  (if (null lst)
      nil
      (or (funcall f goal (first (first lst)) lst-rules pending-goals g)
          (aux-some (rest lst) f goal lst-rules pending-goals g))))
```

```
(defun auxiliar (goal lst lst-rules pending-goals-past f2)
  "Funcion auxiliar donde para cada elemento de lst aplica
   la funcion f2 y retorna T caso sea verdadera para todos
   los elementos de lst
    INPUT: goal: elemento
           lst: una lista con al maximo 2 elementos
           lst-rules: base de conocimiento
           pending-goals-past: casos pendientes de las
           otras llamadas recursivas.
           f2: funcion backward-chaining-aux
    OUTPUT: T o F
  "
  (and
    (funcall f2 (first lst)
                (remove-if #'(lambda (x) (and (equal lst (first x))
                    (eql goal (second x)))) lst-rules)
                (cons goal pending-goals-past)
                #'busca-elemento #'aux-some)
    (funcall f2 (second lst)
                (remove-if #'(lambda (x) (and (equal lst (first x))
                    (eql goal (second x)))) lst-rules)
                (cons goal pending-goals-past)
                #'busca-elemento
                #'aux-some)))

    (defun backward-chaining-aux (goal lst-rules pending-goals
    busca-elemento aux-some)
    "Funcion que retorna T o F si el elemento goal es consecuencia logica
    de la base de conocimiento lst-rules sin utlizar ninguno elemento que
    pertenece al pending-goals.
    INPUT: goal: elemento procurado
           lst-rules: base de conocimiento
           pending-goals: casos pendientes de las
           otras llamadas recursivas.
           busca-elemento: funcion auxiliar busca-elemento
           aux-some: funcion auxiliar aux-some
    OUTPUT: T o F"

    (if (null goal)
        t
        (if (member goal pending-goals)
            nil
            (funcall aux-some
                     (funcall busca-elemento goal lst-rules)
                     #'auxiliar
                     goal
```

```
                    lst-rules
                    pending-goals
                    #'backward-chaining-aux))))

(defun backward-chaining (goal lst-rules)
  "Backward-chaining algorithm for propositional logic

   INPUT: goal:      symbol that represents the goal
          lst-rules: list of pairs of the form
                     (<antecedent>  <consequent>)
                     where <antecedent> is a list of symbols
                     and  <consequent> is a symbol


   OUTPUT: T (goal derived) or NIL (goal cannot be derived)


   NOTES:
        * Implemented with some, every"


  (backward-chaining-aux goal lst-rules NIL #'busca-elemento #'aux-some)  )
```

7.5 Tests

```
;;; EJEMPLOS
>> (backward-chaining 'Q  '((NIL A) (NIL B) ((P) Q) ((L M) P) ((B L) M)
((A P) L) ((A B) L)))
    0: (BACKWARD-CHAINING Q ((NIL A) (NIL B) ((P) Q) ((L M) P) ((B L) M) ((A P) L)
    ((A B) L)))
    0: BACKWARD-CHAINING returned T
T

>> (backward-chaining 'Q '((NIL A) (NIL B) ((A B) C) ((A B) D) ((C D) Q)))
    0: (BACKWARD-CHAINING Q ((NIL A) (NIL B) ((A B) C) ((A B) D) ((C D) Q)))
    0: BACKWARD-CHAINING returned T
T

>> (backward-chaining 'Q '((NIL A) (NIL B) ((A B) C) ((A B) D) ((C E) Q)))
  0: (BACKWARD-CHAINING Q ((NIL A) (NIL B) ((A B) C) ((A B) D) ((C E) Q)))
  0: BACKWARD-CHAINING returned NIL
NIL

>> (backward-chaining 'Q '((NIL A) (NIL B) ((A E) C) ((A B) D) ((C D) Q)))
  0: (BACKWARD-CHAINING Q ((NIL A) (NIL B) ((A E) C) ((A B) D) ((C D) Q)))
  0: BACKWARD-CHAINING returned NIL
```

```
NIL

; caso base
>> (backward-chaining nil '((NIL A) ))
  0: (BACKWARD-CHAINING NIL ((NIL A)))
  0: BACKWARD-CHAINING returned T
T
```

8 Búsqueda en Anchura

8.1 Problema 8 - A

Analiza con detalle la siguiente implementación del algoritmo BFS:

Funcion BFS

El código BFS realiza la búsqueda en anchura en un grafo definido como una lista de adjacencias. Para esto, debe recibir como parámetros:

- end: el node que quieres procurar.
- queue: una lista donde cada elemento es una sublista con el node inicial y sus vecinos. Para la llamada inicial de la funcion es pasado una lista con una sublista que contiene el nudo por donde debe empezar.
- net: la lista de adjacencias.

Si queue es vacio, debe retornar NIL.

Caso la fila no sea vacia, llamamos de path la primera lista de queue y llamamos de node el primer elemento de path. O sea, node es el nudo que vay a ser analizado.

Lo seguiente paso es analizar si node es el nudo procurado.

- Si es el procurado, retorna el path reverso. Fin de la Funcion. condicion de parada. Nudo encontrado.
- Si no es el procurado, enconces es hecha la llamada recuriva.

Los nuevos parametros de la llamada recursiva son:

- parametro 1: end *(es el nudo que seguimos buscando)*
- parametro 2: es una nueva lista donde la continuacion de los caminos pasando por los vecinos del node es añadido al rest de queue.

  (Mirar la análize de la funcion new-path)
- parametro 3: net *(es la lista de adjacencias del grafo. No cambia.)*

Funcion New-Path

Es una funcion hecha para añadir los vecinos del nudo en la fila para que sean analizados después.

8.2 Problema 8 - B

Ejemplo: ((A B E) (B A C D) (C B) (D B) (E A)) Grafo Especial

```
llamada: end <- D
         queue <- ((A))
         net <- ((A B E) (B A C D) (C B) (D B) (E A))
Paso 1: queue <- ((A))
        path <- (A)
        node <- A
Paso 2: queue <- ((B A) (E A))
        path <- (B A)
        node <- B
Paso 3: queue <- ((E A) (A B A) (C B A) (D B A))
        path <- (E A)
        node <- E
Paso 4: queue <- ((A B A) (C B A) (D B A) (A E A))
        path <- (A B A)
        node <- A
Paso 5: queue <- ((C B A) (D B A) (A E A) (B A B A) (E A B A))
        path <- (C B A)
        node <- C
Paso 6: queue <- ((D B A) (A E A) (B A B A) (E A B A) (B C B A))
        path <- (D B A)
        node <- D
        Como el node D es igual al end, retorna (A B D)
```

Ejemplo: ((A B E) (B C D) (C) (D) (E)) Caso típico distinto

```
llamada: end <- D
         queue <- ((A))
         net <- ((A B E) (B C D) (C) (D) (E))
Paso 1: queue <- ((A))
        path <- (A)
        node <- A
Paso 2: queue <- ((B A) (E A))
        path <- (B A)
        node <- B
Paso 3: queue <- ((E A) (C B A) (D B A))
        path <- (E A)
        node <- E
Paso 4: queue <- ((C B A) (D B A))
        path <- (C B A)
        node <- C
Paso 5: queue <- ((D B A))
        path <- (D B A)
```

```
          node <- D
      Como el node D es igual al end, retorna (A B D)
```

Ejemplo: ((a d) (b d f) (c e) (d f) (e b f) (f)) Caso Típico

```
llamada: end <- F
          queue <- ((A))
          net <- ((a d) (b d f) (c e) (d f) (e b f) (f))
 Paso 1: queue <- ((A))
         path <- (A)
         node <- A
 Paso 2: queue <- ((D A))
         path <- (D A)
         node <- D
 Paso 3: queue <- ((F D A))
         path <- (F D A)
         node <- F
         Como el node F es igual al end, retorna (A D F)
```

8.3 Problema 8 - C

Ilustra el funcionamiento del código especificando la secuencia de llamadas a las que da lugar la evaluación:

```
  0: (BFS D ((C)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
    1: (BFS D ((E C)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
      2: (BFS D ((B E C) (F E C)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
        3: (BFS D ((F E C) (D B E C) (F B E C)) ((A D) (B D F) (C E) (D F)
        (E B F) (F)))
          4: (BFS D ((D B E C) (F B E C)) ((A D) (B D F) (C E) (D F) (E B F)
          (F)))
          4: BFS returned (C E B D)
        3: BFS returned (C E B D)
      2: BFS returned (C E B D)
    1: BFS returned (C E B D)
  0: BFS returned (C E B D)
(C E B D)
```

8.4 Problema 8 - D

1. end: es el nudo que quiero buscar
2. queue: es la fila con los nudos y sus vecinos.
3. net: es la lista de adjacencia que representa el grafo

4. path: path de tamaño N es una lista que describe un camino empezando por el nodo de origen y que tiene comprimento N. El origen se vá siempre al final de path.

5. node: es el nodo que vay a ser analizado. Siempre es el primer elemento de path.

6. new-paths: es la funcion que cria la lista con los caminos que si pueden seguir a partir del nodo origen hasta los vecinos del node.

7. El path nunca diminui a cada iteracion, siendo posible aumentar o mantener su tamaño. El algoritmo testa todas las posibilidades de camiños a partir del path, entonces caso el nodo no sea encontrado, podemos asegurar que todavia no hay un camino de tamaño menor o igual al tamaño do path + 1.

   Por ejemplo, si en alguna iteración el path está con tamaño 5, como el path empezó con el tamaño 1, podemos asegurar que no hay ningún camino entre los nós que tenga tamaño entre 1 y 5.

   Luego, cuando el algoritmo encuentra un camino, él es el más pequeño por construccion.

   Así, para encontrar el camiño más pequeño posible entre dos nodos, basta llamar la funcion BFS donde end es el nodo destino y start es el nodo de origen, siendo necesario hacer solamente una adaptación en el tipo de parámetro: (list (list start)).

   El parámetro queue es del tipo lista de lista, como star es un átomo, necesitamos de (list (list start)) para que no tenga error.

8.5 Problema 8 - E

```
Inicio Funcion (end, queue, net)
    Si queue es vacia:
        retorna NIL
    Caso contrario:
        path <- primer elemento de queue
        node <- primer elemento de path
        Si el node for igual a end:
            retorna el reverso de path.
        Caso contrario:
            retira el primer elemento de queue (ya fue visitado)

            para cada vecino del node analizado, crea una lista con el vecino
            y el path.

            q1 <- crea una lista con los resultados de las operaciones arriba.

            llamada recursiva:
                parametro1 <- end
                parametro2 <- q1
                parametro3 <- net
Fin Funcion
```

8.6 Problema 8 - F

Comenta el algoritmo BFS, de forma que se ilustre cómo se ha implementado el pseudocódigo propuesto en el apartado anterior en forma de código.

- Para verificar si una lista es vacia, es utilizada la funcion NULL: (if (null queue))
- Es utilizado LET * para guardar en path el valor de (first queue) y en node el valor de (first path): (let* ((path (first queue) (node (first path))
- Para evaluar si node es igual a end es utilizado eql: (eql node end)
- Para reverter la lista path: (reverse path)
- Para remover el primer elemento de queue: (rest queue)
- Para evitar confusiones, es mejor crear una nueva funcion que realize actualizacion de la lista con path. Debe ser creado una lista con cada vecino del node y path.
- Para encontrar los vecinos: (assoc node net).
- Para obtener la lista solamente con los vecinos: (rest (assoc node net))
- Para crear una lista con path para cada vecino: (mapcar #'(lambda (n) (cons n path))
- Esta funcion retorna una lista con sublistas donde cada sublista es un camino posible entre el nodo de origen y el vecino de node.
- Ahora para añadir esta lista arriba a la lista restante de queue, se utiliza append: (append (rest queue) (new-paths path node net))

```
Inicio Funcion (end, queue, net)
        Si queue es vacia:
                retorna NIL
        Caso contrario:
                path <- primer elemento de queue
                node <- primer elemento de path
                Si el node for igual a end:
                        retorna el reverso de path.
                Caso contrario:
                        retira el primer elemento de queue (ya fue
                        visitado)
                        para cada vecino del node analizado, crea una lista
                         con el vecino y el path.
                        q1 <- crea una lista con los resultados de las
                        operaciones arriba.
                        llamada recursiva:
                        parametro1 <- end
                        parametro2 <- q1
                        parametro3 <- net
Fin Funcion
```

```
(defun bfs (end queue net)
        (if (null queue)
                NIL
        (let* ((path (first queue))
                        (node (first path)))
                (if (eql node end)
                        (reverse path)
                (bfs end
                        (append (rest queue)
                        (new-paths path node net))
                        net))))
```

Figura 1: Correspondencias entre Pseudocodigo y Código Lisp

8.7 Problema 8 - G

```
    end = 'c
    queue = '((f))
    net = '((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))

>> (BFS 'C '((F)) '((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)))
(F B A C)
```

8.8 Problema 8 - H

Ejemplo de lista de adjacencia con problema: ((A B) (B A C) (C B D) (D C A))

```
(defun bfs-improved (end queue net)
         (if (null queue)
                 NIL
                 (let* ((path (first queue))
                             (node (first path)))
                 (if (and (= 2 (count node (rest path)))
                     (not (eql node (first (reverse path)))))
                         NIL
                 (if (eql node end)
                         (reverse path)
                 (bfs-improved end
                         (append (rest queue)
                                 (new-paths path node net))
                         net))))))

;caso típico
>> (bfs-improved 'E '((A)) '((A B) (B E C) (C D) (D B) (E)))
(A B E)
; caso que en bfs entrava en un loop infinito
>> (bfs-improved 'A '((D)) '((A B) (B E C) (C D F) (D B) (E) (F)))
NIL
```