MEMORIA
PRÁCTICA 1

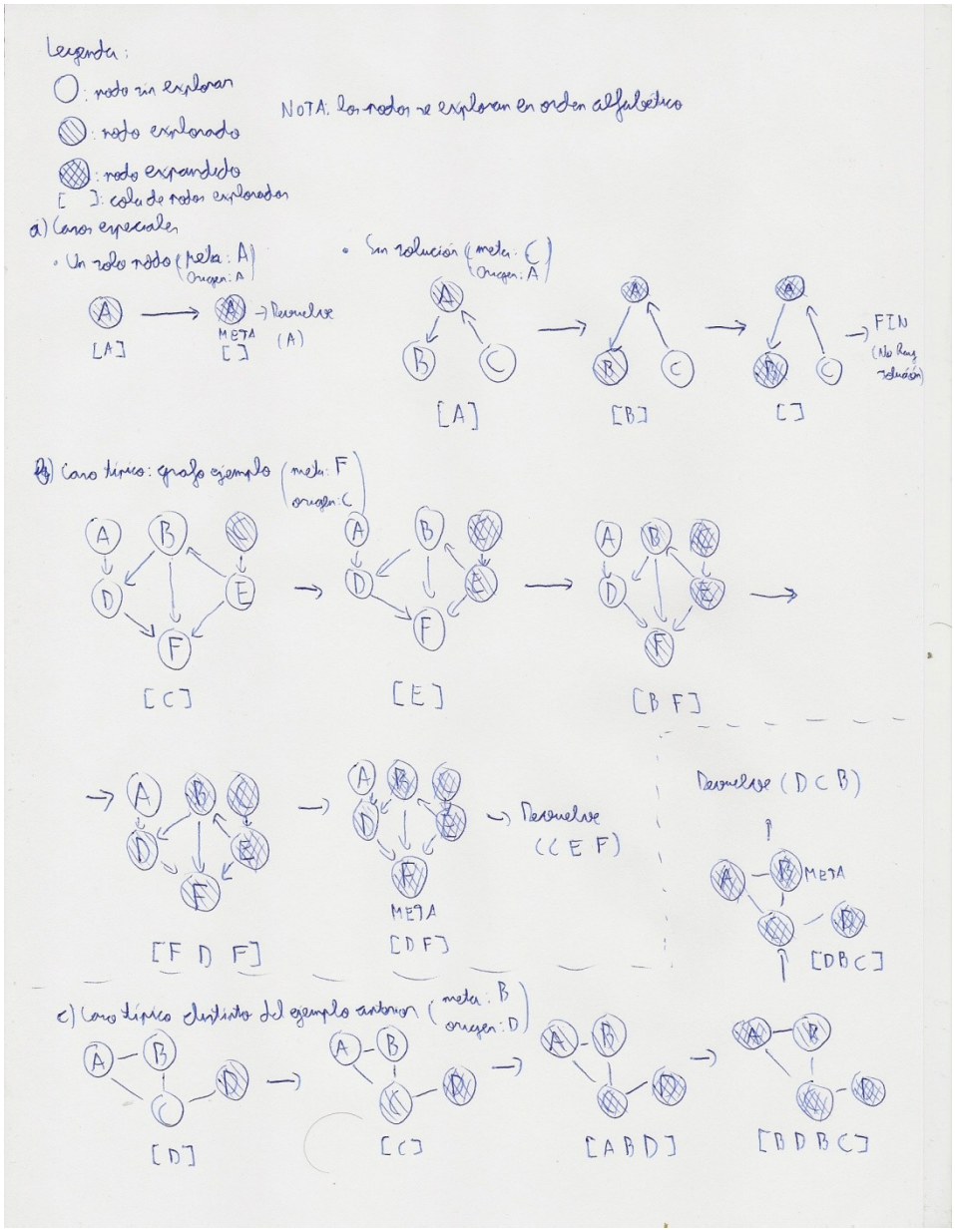
Jose Manuel de Frutos Porras

Daniel Redondo Castilla

Ejercicio 1

(i) En la búsqueda en anchura se ha de mantener una lista de nodos no explorados. Cada vez que se analiza un nodo, se comprueba si este es el nodo objetivo. En caso de no serlo, se añaden los nodos hijos al final de la lista de nodos no explorados. Entonces se toma el primer nodo de la lista y se continúa la búsqueda desde este.

Como siempre se ponen los nodos de mayor profundidad en el árbol al final de la cola de no explorados, el algoritmo se asegura la búsqueda sucesiva por niveles de profundidad.



(ii) La búsqueda en profundidad es óptima si el peso de cada arista es el mismo. Como en este algoritmo no estamos considerando el peso de las aristas (como si todas las aristas tuvieran el mismo peso) entonces podemos concluir que el algoritmo es óptimo.

(iii)
```
bfs( nodo_objetivo, cola_No_Explorados, arbol)

          Si vacia(cola_No_Explorados) devolver nada

          nodo_Explorar<= extraer(cola_No_Explorados)

          Si igual(nodo_Explorar, nodo_objetivo)

               devolver camino_hasta(nodo_explorar)

          si_no bfs( nodo_objetivo,
insertar(cola_No_Explorados, nodos_hijos(nodo_Explorar)), arbol)
```

(iv)
```
(defun bfs (end queue net)
     (if (null queue);Caso base cola de nodos no explorados va-
cia devolvemos nil
     nil
     (let ((path (car queue)))
          (let ((node (car path)))
               (if (eql node end) ; Si el primer nodo de la
cola de nodos no explorados es nodo objetivo finalizamos
                    (reverse path)
                    (bfs end ;En caso de no serlo añadimos los
nodos hijos a la cola de nodos no explorados
                         (append (cdr queue)
                              (new-paths path node net))
                         net))))))

(defun new-paths (path node net)
     (mapcar #'(lambda (n)
                    (cons n path))
          (cdr (assoc node net))))
```

(v) En el ejemplo propuesto:

```
0[1]: (SHORTEST-PATH A F ((A D) (B D F) (C E) (D F) (E B F) (F)))

  1[1]: (BFS F ((A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))

    2[1]: (NEW-PATHS (A) A ((A D) (B D F) (C E) (D F) (E B F) (F)))

    2[1]: returned ((D A))

    2[1]: (BFS F ((D A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))

      3[1]: (NEW-PATHS (D A) D ((A D) (B D F) (C E) (D F) (E B F) (F)))

      3[1]: returned ((F D A))
```

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

Para los casos especiales:

```
0[1]: (SHORTEST-PATH A A ((A)))
   1[1]: (BFS A ((A)) ((A)))
   1[1]: returned (A)
 0[1]: returned (A)
(A)


0[1]: (SHORTEST-PATH A C ((A B) (C A) (B)))
   1[1]: (BFS C ((A)) ((A B) (C A) (B)))
     2[1]: (NEW-PATHS (A) A ((A B) (C A) (B)))
     2[1]: returned ((B A))
     2[1]: (BFS C ((B A)) ((A B) (C A) (B)))
       3[1]: (NEW-PATHS (B A) B ((A B) (C A) (B)))
       3[1]: returned NIL
       3[1]: (BFS C NIL ((A B) (C A) (B)))
       3[1]: returned NIL
     2[1]: returned NIL
   1[1]: returned NIL
 0[1]: returned NIL
NIL
```

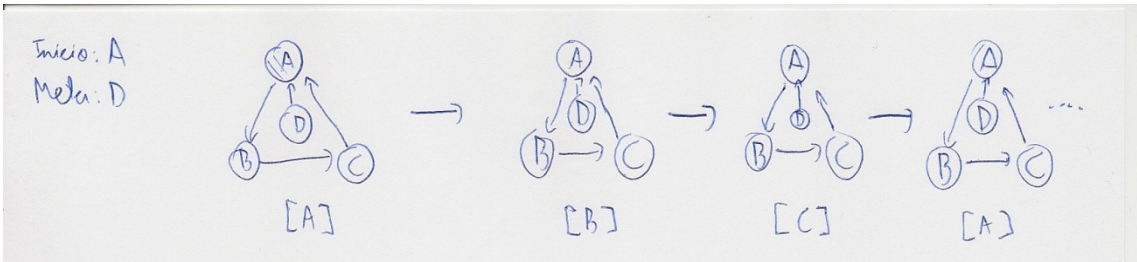
(vi)

a. `(setf 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)))`
`(print (shortest-path 'f 'c  net ))`

b. El resultado es (f b a c), debido a que los nodos adyacentes están introducidos en orden alfabético en el ejemplo.

(vii)

a.



b.

```
PSEUDOCODIGO
      Entrada: nodo_inicial, nodo_objetivo, grafo
      Salida: camino mas corto entre nodo_inicial y nodo_objetivo

      insetar nodo_inicial en cola_generados
      iterar
            if cola_generados es vacia -> return fallo
            mirar nodo de cola_generados
            if igual(nodo, nodo_objetivo ->return camino de
nodo_inicial a nodo_objetivo
            else
                  si nodo no esta en lista_repetidos
                        expandir nodo
                        introducir nodo en lista_repetidos
                        insertar sucesores de nodo en
cola_generados

CODIGO

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; bfs-improved
;;;  bfs que evita recursion infinita en grafos ciclicos para
;;;  problemas sin solucion
;;;  INPUT
;;;    end: nodo objetivo
;;;    queue: lista de listas de nodos por explorar
;;;    net: grafo a explorar
;;;    explored: lista vacia (aqui se introduciran los nodos
;;;                explorados)
;;;  OUTPUT
;;;   camino mas corto entre el nodo introducido en queue y end
```

```
(defun bfs-improved (end queue net explored)
  (if (null queue)
      nil
    (let ((path (car queue)))
      (let ((node (car path)))
        (if (eql node end)
            (reverse path)
          (if (null (member node explored))
              (bfs-improved end
                            (append (cdr queue)
                                    (new-paths path node net))
                            net
                            (cons node explored))
            (bfs-improved end
                          (cdr queue)
                          net
                          explored)))))))
(defun new-paths (path node net)
  (mapcar #'(lambda(n)
              (cons n path))
    (cdr (assoc node net))))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; shortest-path-improved
;;;  encuentra el camino mas corto entre dos nodos, evitando el
;;;  problema de recursion infinita
;;;  INPUT
;;;    end: nodo objetivo
;;;    queue: lista nodo inicial
;;;    net: grafo a explorar
;;;    explored: lista vacia (aqui se introduciran los nodos
;;;              explorados)
;;;  OUTPUT
;;;   camino mas corto entre el nodo introducido en queue y end


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

EJEMPLOS

```
(shortest-path-improved 'A 'A '((A)) nil) ;-> un solo nodo
(shortest-path-improved 'A 'C '((A B) (C A) (B)) nil) ;-> sin
solucion
(shortest-path-improved 'a 'f '((a d) (b d f) (c e) (d f) (e b
f) (f)) nil) ;-> caso ejemplo del enunciado
(shortest-path-improved 'A 'D '((A B) (B C) (C A) (D A)) nil)
;-> caso cíclico sin solución
```

Ejercicio 2

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Distancia euclidea
;;;   Entrada: vector x expresado en coordenadas cartesianas
;;;                vector y expresado en coordenadas cartesianas
;;;
;;; Salida: distancia entre el vector x y el vector y.
;;;
;;;   PSEUDOCODIGO
;;;         Distancia euclidea(x y)
;;;               vectorAux <- restar el vector x componente a
componente al vector y
;;;               vectorAux <- elevar al cuadrado cada componente del
vector vectorAux.
;;;               aux <- sumar dos a dos cada componente de vectorAux
;;;               devolver raiz(aux)
;;;
;;;
;;;



;;;
;;;   euclidean-mapcar(x y)
;;;   calcula la distancia euclidea de dos vectores.
;;;
;;; INPUT: x vector expresado en coordenadas cartesianas
;;; OUTPUT: y vector
;;;
(defun sqr(x)(* x x))

(defun euclidean-mapcar( x y)
  (sqrt (reduce #'+
                (mapcar #'sqr
                  (mapcar #'- x y)))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   EJEMPLOS
;;;   ;Casos Tipicos
;;;         (print (euclidean-mapcar '( 3 5 35) '( 1 8  6))) ;->29.223
;;;         (print (euclidean-mapcar '( 0 0 0 0 ) '( 1 2  2 4))) ; -> 5
;;;
;;;   ;Casos especiales
;;;         (print (euclidean-mapcar '( 3 5 ) '( 1 2  2 4))) ; -> 3.60
;;;         (print (euclidean-mapcar nil '(1 2  3 4)) ); -> 0
;;;         (print (euclidean-mapcar '( 1 2  3 4) nil) ); -> 0
;;;         (print (euclidean-mapcar nil nil) ); -> 0


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   euclidean-rec(x y)
;;;   calcula la distancia euclidea de dos vectores mediante
recuersiva.
```

```
;;;
;;; INPUT: x vector expresado mediante sus n coordenadas.
;;; OUTPUT: y vector
;;;

(defun op1(x y) (sqr (- x y)))

(defun cuadrado-dist(x y)
  (if (or (null x) (null y)) ;Caso base: Alguna de las dos listas ya
no tiene mas elementos.
    0; Sumamos todos los resutados de aplicar op1 sobre cada par de coordenadas.
    (+
     (op1 (first x) (first y)) ; Realizamos op1 sobre la componente primera del
vector
     (cuadrado-dist (rest x) (rest y)))))


(defun euclidean-rec( x y)
  (sqrt (cuadrado-dist x y) ) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   EJEMPLOS
;;;   ;Casos Tipicos
;;;         (print (euclidean-rec '( 3 5 35) '( 1 8  6))) ;->29.223
;;;         (print (euclidean-rec '( 0 0 0 0 ) '( 1 2  2 4))) ; -> 5
;;;
;;;   ;Casos especiales
;;;         (print (euclidean-rec '( 3 5 ) '( 1 2  2 4))) ; -> 3.60
;;;         (print (euclidean-rec nil '(1 2  3 4)) ); -> 0
;;;         (print (euclidean-rec '( 1 2  3 4) nil) ); -> 0
;;;         (print (euclidean-rec nil nil) ); -> 0



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Desviacion absoluta
;;;   Entrada: vector x expresado en coordenadas cartesianas
;;;               vector y expresado en coordenadas cartesianas
;;;
;;; Salida: distancia entre el vector x y el vector y.
;;;
;;;   PSEUDOCODIGO
;;;         Desviacion absoluta(x y)
;;;               vectorAux <- restar el vector x componente a
componente al vector y
;;;               vectorAux <- valor absoluto de cada componente del
vector vectorAux.
;;;               aux <- sumar dos a dos cada componente de vectorAux
;;;               devolver aux
;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   AD-mapcar(x y)
;;;   calcula la desviacion absoluta de dos vectores.
;;;
;;; INPUT: x vector expresado mediante sus n coordenadas.
;;; OUTPUT: y vector

(defun      AD-mapcar(x y)
```

```
  (reduce #'+ (mapcar #'abs (mapcar #'- x y))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   EJEMPLOS
;;;   ;Casos Tipicos
;;;         (print (AD-mapcar '( 3 5 35) '( 1 8  6))) ;-> 34
;;;         (print (AD-mapcar '( 0 0 0 0 ) '( 1 2  2 4))) ; -> 9
;;;
;;;   ;Casos especiales
;;;         (print (AD-mapcar '( 3 5 ) '( 1 2  2 4))) ; -> 5
;;;         (print (AD-mapcar nil '(1 2  3 4)) ); -> 0
;;;         (print (AD-mapcar '( 1 2  3 4) nil) ); -> 0
;;;         (print (AD-mapcar nil nil) ); -> 0


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   AD-rec(x y)
;;;   calcula la desviacion absoluta de dos vectores.
;;;
;;; INPUT: x vector expresado mediante sus n coordenadas.
;;; OUTPUT: y vector
;;;

(defun op2(x y) (abs (- x y)))

(defun AD-rec(x y)
  (if (or (null x) (null y))  ;Caso base: Alguna de las dos listas ya no tiene mas elementos.
      0 ; Sumamos todos los resutados de aplicar op1 sobre cada par de coordenadas.
    (+ (op2 (first x) (first y)) (AD-rec (rest x) (rest y)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   EJEMPLOS
;;;   ;Casos Tipicos
;;;         (print (AD-rec '( 3 5 35) '( 1 8  6))) ;-> 34
;;;         (print (AD-rec '( 0 0 0 0 ) '( 1 2  2 4))) ; -> 9
;;;
;;;   ;Casos especiales
;;;         (print (AD-rec '( 3 5 ) '( 1 2  2 4))) ; -> 5
;;;         (print (AD-rec nil '(1 2  3 4)) ); -> 0
;;;         (print (AD-rec '( 1 2  3 4) nil) ); -> 0
;;;         (print (AD-rec nil nil) ); -> 0
```

Ejercicio 3

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;
;;; Ejercicio combine-elt-lst
;;; Entrada: elt (elemento)
;;;               lst (lista de elementos)
;;;
;;;   Salida: lista de todas las combinaciones del elemento
;;;
;;;   PSEUDOCODIGO:
```

```
;;;         combine-elt-lst( elt, lst)
;;;               si lst vacia
;;;                     devolver lista-vacia
;;;               combinacion <- combinar elt con primer elemento lst
;;;               anadir a lista-combinaciones combinacion y combinar-
elt-let(elt, resto de lst)
;;;               devolver lista-combinaciones
;;;
;;;
;;;   combine-elt-lst: combina un elemento dado con todos los de una
lista.
;;;
;;;   ENTRADA: elt [Elemento]
;;;               lst [Lista]
;;;
;;;   SALIDA: Lista de pares formados por el elemento elt y un
elemento de la lista.
;;;


(defun combine-elt-lst(elt lst)
  (unless (null lst)
    (cons (list elt (car lst)) ( combine-elt-lst elt (cdr lst)) )))

;;; ;EVALUACION:
;;;
;;;   ;CASOS ESPECIALES:
;;;               (combine-elt-lst nil nil); -> NIL
;;;               (combine-elt-lst nil '(1 2 3)); -> ((NIL 1) (NIL 2)
(NIL 3))
;;;         (combine-elt-lst '(a) nil); ->((A NIL))
;;;
;;;         ;CASOS TIPICOS:
;;;               (combine-elt-lst 'a '(1 2 3) ); -> ((A 1) (A 2) (A
3))
;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;
;;;   Ejercicio combine list-of-lsts
;;;
;;;   Entrada: lst1 (lista de elementos)
;;;               lst2 (lista de elementos)
;;;
;;;   Salida: producto cartesiano de las dos listas
;;;
;;;   PSEUDOCODIGO:
;;;         combine-lst-lst( lst1, lst2)
;;;               Si lst1 o lst2 es la lista vacia
;;;               devolver lista-vacia
;;;         combinaciones <-combina primer elemento lst1 con lst2
;;;               anadir a la lista-resultados combinaciones y
combinar-elt-let(elt, resto de lst1)
;;;
;;;   combine-lst-lst(lst1 lst2)
;;;         Realiza el producto cartesiano de dos listas.
;;;
;;;   ENTRADA: lst1 [Lista1]
;;;               lst2 [Lista2]
;;; SALIDA: Lista de pares formados por un elemento de lst1 y un
elemento de lst2
```

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



;;; ;EVALUACION:
;;;
;;;   ;CASOS ESPECIALES:
;;;         ;;Argumentos nulos
;;;         (combine-lst-lst nil nil); -> NIL
;;;
;;;         (combine-lst-lst nil '(1 2 3))
;;;    -> ((NIL 1) (NIL 2) (NIL 3))
;;;
;;;         (combine-lst-lst '(a) nil); ->((A NIL))
;;;
;;;         (combine-lst-lst '(a) '(1 2 3))
;;; ->((A 1) (A 2) (A 3)) ;Un unico elemento con lista (caso del ej. anterior)
;;;
;;;         ;CASOS TIPICOS:
;;;               (combine-lst-lst '(a b c) '(1 2 3))
;;; -> ((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3))
;;;
;;;               (combine-lst-lst '(a b c d) '(1 2 3))
;;;-> ((A 1) (A 2) (A 3) (B 1) (B 2) (B 3) (C 1) (C 2) (C 3) (D 1) ...) ;No muestra
todo el resultado
```

COMENTARIO: se deben pasar listas como argumentos, en caso contrario no funcionara

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;
;;;   Ejercicio combine-lst-lsts
;;;   Entrada: elt [elemento]
;;;                lstolsts [lista de listas de elementos]
;;;
;;;   Salida: lista de listas de elementos con el elt como nuevo
elemento.
;;;
;;;   PSEUDOCODIGO:
;;;         combine-elt-lstolsts(elt lstolsts)
;;;               Si lstolsts es la lista vacia devolver lista-vacia
;;;         devolver Anade(Anade elt a la lista first(lstolsts),
combine-elt-lstolsts(elt lstolsts))
;;;
;;;   INPUT: elt [elemento]
;;;                lstolsts [Lista de listas]
;;;
;;; OUTPUT: Lista de n-tuplas formadas por un elemento de cada lista
;;;
;;; Combina un elemento con cada las listas de elementos que
constituyen lstolsts
(defun combine-elt-lstolsts( elt lstolsts )
  (unless (null lstolsts)    ; Caso base
    ( cons (cons elt (first lstolsts)) (combine-elt-lstolsts elt (rest
lstolsts ))))) ; Anadimos el elemento a la primera lista de elementos
de lstolsts y hacemos recursion sobre la lista de listas

;;; ;EVALUACION:
```

```
;;;
;;;   ;CASOS TIPICOS:
;;;     (combine-elt-lstolsts 'a '((1 2 3) (+ -))); -> ((A 1 2 3) (A + -))
;;;
;;;   ;CASOS ESPECIALES:
;;;     (combine-elt-lstolsts 'a nil);-> nil
;;;   (combine-elt-lstolsts nil '((a b c) (+ -)));->((NIL A B C) (NIL + -))
;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;
;;;
;;;   Entrada: lst [ lista de elemento]
;;;                lstolsts [lista de listas de elementos]
;;;
;;;   Salida: combinacion de las listas de elementos que constituyen
lstolsts con los elementos de lst.
;;;
;;;   PSEUDOCODIGO:
;;;         combine-lst-lsts(lst lstolsts)
;;;               Si lstolsts es la lista vacia devolver lista-vacia
;;;         devolver anade( combine-elt-lstolsts(first(lst) lstolsts),
combine-lst-lsts(rest(elt) lstolsts))
;;;
;;;   INPUT: lst  [lista de elementos]
;;;               lstolsts [Lista de listas]
;;;
;;; OUTPUT: Lista de n-tuplas formadas por un elemento de cada lista

;;; combina los elementos de lst con las listas de elementos que
constituyen lstolsts.

(defun combine-lst-lsts(lst lstolsts)
  (unless (null lst)
    (append (combine-elt-lstolsts (first lst) lstolsts) (combine-lst-
lsts (rest lst) lstolsts))))

;;; ;EVALUACION
;;; ;CASOS TIPICOS
;;;         (combine-lst-lsts '(a b c) '((1 2 3) (+ -))); -> ((A 1 2 3)
(A + -) (B 1 2 3) (B + -) (C 1 2 3) (C + -)
;;;   ;CASO ESPECIALES
;;;         (combine-lst-lsts '(a b c) '((1 2 3) (+ -))); ->  nil
;;;         (combine-lst-lsts '(a b c) '((1 2 3) (+ -))); -> nil
;;;         (combine-lst-lsts '(a b c) '((1 2 3) (+ -)));-> nil


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;
;;;
;;;   Entrada: lstolsts [lista de listas de elementos]
;;;
;;;   Salida: producto cartesiano de las listas que constituyen
lstolsts
;;;
;;;   PSEUDOCODIGO:
;;;         combine-lst-of-lsts(lst lstolsts)
;;;               Si lstolsts esta compuesto de dos listas combinamos
una lista con otra.
```

```
;;;         Combina la primera lista con combine-lst-of-
lsts(rest(lstolsts)) ; A1 X ...X An = A1 X (A2 X (...XAn)...)
;;;
;;;   INPUT: lst  [lista de elementos]
;;;               lstolsts [Lista de listas]
;;;
;;; OUTPUT: Lista de n-tuplas formadas por un elemento de cada lista
(defun combine-lst-of-lsts(lstolsts)
  (if (null (rest (rest lstolsts))) ; Si solo hay dos listas en
lstolsts hacemos la combinacion de ambas
      (combine-lst-lst  (first lstolsts) (first(rest lstolsts )) )
    (combine-lst-lsts (first lstolsts) (combine-lst-of-lsts(rest
lstolsts))))) ; A1 X ...X An = A1 X (A2 X (...XAn)...)


;;; ;EVALUACION:
;;;
;;;   ;CASOS TIPICOS:
;;;         (setf res (combine-lst-of-lsts '((1 2 3) (+ -) (a b c))) );
;;;         (length res); -> 18 imprimimos la longitud ya que es poco
viable imprimir toda la lista resultado.
;;;         (combine-lst-of-lsts '((1 2 3) (+ -) ));  -> ((1 +) (1 -)
(2 +) (2 -) (3 +) (3 -))

;;;   ;CASOS ESPECIALES:
;;;         (combine-lst-of-lsts nil); -> NIL
;;;         (combine-lst-of-lsts '((1 2 3) )); -> nil
;;;
```

Ejercicio 4

Utilizamos estas funciones auxiliares definidas en el enunciado más otras definidas por nosotros:

```
(defconstant +bicond+ '<=>)
(defconstant +cond+ '=>)
(defconstant +and+ '^)
(defconstant +or+ 'v)
(defconstant +not+ '¬)

(defun truth-value-p (x)
  (or (eql x T) (eql x NIL)))
(defun unary-connector-p (x)
  (eql x +not+))
(defun binary-connector-p (x)
  (or (eql x +bicond+)
      (eql x +cond+)))
(defun n-ary-connector-p (x)
  (or (eql x +and+)
      (eql x +or+)))
(defun connector-p (x)
  (or (unary-connector-p x)
      (binary-connector-p x)
      (n-ary-connector-p x)))

(defun bicond-p (x)
  (eql x +bicond+))

(defun or-p (x)
  (eql x +or+))
```

```
(defun and-p (x)
  (eql x +and+))

(defun not-p (x)
  (eql x +not+))

(defun cond-p(x)
  (eql x +cond+))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   extrae_simbolos
;;;   INPUT: expresion de FBFs en cualquier formato
;;;   OUTPUT: lista de simbolos atomicos sin repeticiones
;;;
;;;   PSEUDOCODIGO:
;;;         extrae_simbolos(expr)
;;;         if(es_vacia(expr)) -> devolver lista_vacia
;;;         if (primer_elemento es atomico) then
;;;               if (es conector o valor-de-verdad) -> devolver
extrae_simbolos (resto_elementos)
;;;               else union(primer elemento y
extrae_simbolos(resto_elementos))
;;;         else
;;;               union(extrae_simbolos(primer_elemento) y
extrae_simbolos(resto_elementos))
;;;


(defun extrae-simbolos(expr)
  (unless (null expr)
    (if (atom (first expr))
        (if (or (connector-p(first expr)) (truth-value-p (first
expr)))
            (extrae-simbolos (rest expr))
          (union (list(first expr)) (extrae-simbolos (rest expr))))
      (union (extrae-simbolos(first expr)) (extrae-simbolos(rest
expr))))))


;;; EVALUACION:
;;;
;;;   ;CASOS ESPECIALES
;;;         (extrae-simbolos nil )              ;lista vacia
;;;         (extrae-simbolos '((¬ p)))          ;solo una fbf
;;;         (extrae-simbolos '((P) (Q) (R) nil t))   ;no debe extraer
valores de verdad
;;;
;;;   ;CASOS TIPICOS
;;;         (extrae-simbolos '((=> (^ P I) L) (=> (¬ P) (¬ L)) (v P)
(L))) ;caso ejemplo del enunciado
;;;         (extrae-simbolos '(=> (^ (v P Q) ) L <=> A))
      ;caso ejemplo con notacion infija y prefija


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;   genera-lista-interpretaciones
;;;   Entrada: lista (lista de atomos)
;;;
```

```
;;;   Salida: lista de todas las 2^n interpretaciones.
;;;
;;;   PSEUDOCODIGO:
;;;         genera-lista-interpretaciones(lst):
;;;               Si lst es vacia devolver lista_vacia
;;;               interAtomoP<-combinar primer_elemento lst con (V F)
;;;               devolver combinar interAtomoP con genera-lista-
interpretaciones(resto lst)
;;;
;;;
;;;
;;;   Entrada: lst [lista de n simbolos atomicos]
;;;                       (ATOMO1 ATOMO2 ... ATOMOn)
;;;   Salida: Lista de las 2^N interpretaciones.
;;;               ( (<Interpretacion>(ATOMO1 valorDeVerdad) (ATOMO2
valorDeVerdad) ... (ATOMOn valorDeVerdad))... )
;;;
;;;
;;;
;;;
(defun genera-lista-interpretaciones(lst)
  (if (null lst)
      '(nil)
    (combine-lst-lsts (combine-elt-lst (first lst) '(T nil)) (genera-
lista-interpretaciones (rest lst)))))

;;; Es importante tener en cuenta que esta funcion en el fondo hace el
producto cartesiano de las las posibles interpretaciones de un atomo
con la de los otros atomos.
;;; El elemento neutro del producto cartesiano no es el conjunto
vacio. A X ø =  ø. Si no que es el conjunto formado por el conjunto
vacio.  A X {ø} =  {ø}. Esto explica la eleccion del caso base

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   ;EVALUACION:
;;;     ;CASOS TIPICOS
;;;         (genera-lista-interpretaciones '(P Q R)) ;->(((P T) (Q T)
(R T)) ((P T) (Q T) (R NIL)) ((P T) (Q NIL) (R T)) ((P T) (Q NIL) (R
NIL)) ((P NIL) (Q T) (R T)) ((P NIL) (Q T) (R NIL)) ((P NIL) (Q NIL)
(R T)) ((P NIL) (Q NIL) (R NIL)))
;;;
;;;     ;CASOS ESPECIALES
;;;         (genera-lista-interpretaciones nil) ; -> (nil)
;;;         (genera-lista-interpretaciones '(A)); -> (((A T)) ((A
NIL)))
;;;
;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; interpretacion-modelo-p
;;;
;;; Es una funcion compleja luego hemos separado mucho el codigo
menudas subrutinas.
;;;
;;; ENTRADA: lstexp [lista de fbf que constituyen la base de
conocimiento]
```

```
;;;         inter [interpretacion]
;;;
;;; SALIDA: T si la interpretacion es modelo de la base de
conocimiento
;;;         nil en caso contrario.
;;;
;;; PSEUDOCODIGO:
;;; interpretacion-modelo-p(lstexp inter)
;;;     Si es vacia lstexp entonces devolver nil
;;;     Si tiene valor de verdad (first lstexp) para la interpretacion
inter
;;;     y interpretacion-modelo-p( (rest lstexp) inter) entonces
;;;     devolver T
;;;     Si no devolver nil
;;;
;;;


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; funcion evaluar(exp inter)
;;; Necesitamos definir otra funcion que evalue una fimple fbf y
devuelva su valor de verdad, para esa interpretacion.
;;;
;;; ENTRADA: exp [fbf]
;;;         inter [interpretacion]
;;;
;;; SALIDA: T si la fbf tiene valor de verdad
;;;         nil en caso contrario.
;;;
;;; PSEUDOCODIGO:
;;; Suponemos notacion prefija
;;; evaluar(exp inter)
;;;     Si first(exp) es un bicondicional entonces devolver valor-de-
verdad-bicondicional( rest(exp) inter) ; Los argumentos del operador
bicondicional estaran acontinuacion en la lista y su valor de verdad
en la lista inter
;;;     Si first(exp) es un condicional entonces devolver valor-de-
verdad-condicional( (rest(exp) inter)
;;;     Si first(exp) es un or entonces devolver valor-de-verdad-
or( rest(exp) inter)
;;;     Si first(exp) es un and entonces devolver valor-de-verdad-and(
rest(exp) inter)
;;;     Si first(exp) es un not entonces devolver valor-de-verdad-not(
rest(exp) inter)
;;;     Si first(exp) es atomo entonces devolver valor-de-verdad-
atomo( exp inter)
;;;
;;;
;;;
;;; funciones:
;;; or-valor-de-verdad(exp inter), and-valor-de-verdad(exp inter),
bicond-valor-de-verdad(exp inter), cond-valor-de-verdad(exp inter) ,
not-valor-de-verdad(exp inter)
;;;
;;; Necesitamos ahora funciones que nos devuelvan el valor de verdad
de las fbf mas elementales (=>, <=> , V , ^ , ¬).
;;; En definitiva el programa tomará un fbf compleja compuesta de
muchas simples e irá separando esta en sus fbf simples constitutivas y
analizando su valor de verdad.
;;;
```

```
;;; PSEUDOCODIGO:
;;; or-valor-de-verdad(exp inter)
;;;      Si exp es vacio devolver nil ; elemento neutro del or
;;;      or(evaluar(first(exp) inter) or-valor-de-verdad(rest(exp)
inter))
;Evaluamos el primer argumto del or y luego recursivamente sus n arguementos hasta que
no haya mas.

;;;
;;; and-valor-de-verdad(exp inter)
;;;     Si exp es vacio devolver T ; elemento neutro del and
;;;      and(evaluar(first(exp) inter) and-valor-de-verdad(rest(exp)
inter))
;;;
;;; ;El bicondicional y el condicional son operadores binarios luego ya sabemos que
solo tienen dos arguementos

;;; bicond-valor-de-verdad(exp inter)
;;;   or(and(evaluar(first(exp) inter) evaluar(second (exp) inter))
and(not(evaluar(first(exp) inter))) not(evaluar(second(exp) inter)))
    ;A<=>B = (A^B) v ( not A ^ not B) . Expresion equivalente.
;;;
;;; cond-valor-de-verdad(exp inter)
;;;   or(not(evaluar(first (exp) inter)) evaluar(second(exp) inter))
    ; A -> B = not A V B
;;;
;;; ;El not es un operador unario luego sabemos que solo tiene un argumento.
;;; not-valor-de-verdad(exp inter)
;;;   not(evaluar(exp) inter)
;;;

(defun bicond-valor-de-verdad(exp inter)
  (or (and (evaluar (first exp) inter) (evaluar (second exp) inter))
(and (not (evaluar (first exp) inter) ) (not (evaluar (second exp)
inter))))) ; A<=>B = (A^B) v ( not A ^ not B)

(defun or-valor-de-verdad(exp inter)
  (unless (null exp) ; Elemento neutro del or es NIL
    (or (evaluar (first exp) inter) (or-valor-de-verdad (rest exp)
inter)))) ; or de todos la fbf que componen la expresion or. Operador n-ario

(defun and-valor-de-verdad(exp inter)
  (if (null exp)
      T                 ;T elemento neutro del and
    (and (evaluar (first exp) inter) (and-valor-de-verdad (rest exp)
inter)))) ; and de todos la fbf que componen la expresion or. Operador n-ario

(defun not-valor-de-verdad(exp inter)
  (not (evaluar exp inter)))

(defun cond-valor-de-verdad(exp inter)
  (or (not (evaluar (first exp) inter)) (evaluar (second exp)
inter) )) ; A -> B = not A V B

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Funcion get-inter-atomo
;;; Funcion que nos permite obtener el valor de verdad de un atomo.
;;; Si es un valor de verdad lo devuelve. Si es un atomo lo busca en
la lista inter y devuelve su valor de verdad.
;;;
;;; INPUT:
;;;    atomo: atomo para el que se busca el valor de verdad
```

```
;;;    inter: interpretacion que estamos usando para la fbf

(defun get-inter-atomo(atomo  inter)
  (if (truth-value-p atomo)
      atomo
    (second (assoc atomo inter))))
;; Busqueda en el diccionario inter el par (atomo valor de verdad) y devuelve el
valor ;;  de verdad asociado al atomo

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;   ; EVALUACIONES
;;;         (get-inter-atomo 'A '((A t) (B nil) (C t))); ->T
;;;         (get-inter-atomo 'B '((A t) (B nil) (C t))); ->nil

;;;   ; CASOS ESPECIALES
;;;          (get-inter-atomo 'H '((A t) (B nil) (C t))); ->nil ;
buscamos un atomo que no esta en las interpretaciones
;;;
;;;         ;;Argumentos nulos
;;;         (get-inter-atomo nil nil) ;-> nil
;;;         (get-inter-atomo A' nil) ;-> nil
;;;         (get-inter-atomo nil '((A t) (B nil) (C t))) ;-> nil


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; funcion evaluar
;;; INPUT: exp [fbf]
;;;         inter [interpretacion]
;;;
;;; OUTPUT: T si la fbf tiene valor de verdad
;;;         nil en caso contrario.
;;;
(defun evaluar(exp inter)
  (unless (or (null exp) (null inter))
; Entradas
    (cond ((and (not (listp exp)) (not (connector-p exp))) (get-inter-
atomo exp inter))    ; Caso en el que evaluamos un atomo
          ((not (connector-p (first exp))) (evaluar (first exp)
inter))                          ; Caso en el que una fbf esta anidada
de mas
          ((bicond-p (first exp)) (bicond-valor-de-verdad (rest exp)
inter))
          ((or-p (first exp)) (or-valor-de-verdad (rest exp) inter))
          ((and-p (first exp)) (and-valor-de-verdad (rest exp) inter))
          ((cond-p (first exp)) (cond-valor-de-verdad (rest exp)
inter))
          ((not-p (first exp)) (not-valor-de-verdad (rest exp)
inter)))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;
;;;
;;; ; EVALUACIONES
;;;         ;CASOS TIPICOS
;;;               (evaluar '(V (¬ A) B) '((A nil) (B nil))); -> T
;;;               (evaluar '(V (^ A C) B) '((A T) (B nil)(C T))); -> T
;;;               (evaluar '(V (^ A C) (<=> B D)) '((A T) (B nil)(C
nil)(D T))); -> nil
```

```
;;;               (evaluar '(V (^ A C) (<=> B D)) '((A T) (B nil)(C
nil)(D nil))); -> T
;;;               (evaluar '(V (^ A C) (=> B D)) '((A T) (B nil)(C nil)
(D T))); -> T
;;;               (evaluar '(V (^ A C) (=> (¬ B) D)) '((A T) (B nil)(C
nil)(D nil))); -> nil
;;;
;;;     ;CASOS ESPECIALES
;;;          (evaluar nil '((A nil) (B nil))) ;-> nil ; sin expresion
;;;          (evaluar '(V (^ A C) (<=> B D)) nil) ;-> nil ; sin
interpretaciones

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO interpretacion-modelo-p
;;;
;;; ;INPUT: lstexp [lista de fbf que constituyen la base de
conocimiento]
;;;         inter [interpretacion]
;;;
;;; ;OUTPUT: T si la interpretacion es modelo de la base de
conocimiento
;;;         nil en caso contrario.
;;;
(defun interpretacion-modelo-p(lstexp inter)
  (if (null lstexp)
      T
; T elemento neutro del and
    (and (evaluar (first lstexp) inter) (interpretacion-modelo-p (rest
lstexp) inter ))))  ; Vemos si cada fbf de la base de conocimiento es
Verdadera para esa interpretacion

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;
;;;
;;; ; EVALUACION:
;;;     ;CASOS TIPICOS:
;;;          (interpretacion-modelo-p '((<=> A (¬ H)) (<=> P (^ A H))
(<=> H P)) '((A NIL) (P NIL) (H T))); -> nil
;;;          (interpretacion-modelo-p '((<=> A (¬ H)) (<=> P (^ A H))
(<=> H P)) '((A T) (P NIL) (H NIL))); -> T
;;;
;;;     ;CASOS ESPECIALES
;;;        (interpretacion-modelo-p '(((<=> ((A)) (¬ H))) (<=> P ((^ A
H))) (<=> H P) t) '((A NIL) (P NIL) (H T))); ->nil ; Caso 1 con mas
parentesis.Balanceo correcto
;;;        (interpretacion-modelo-p '((<=> ((A)) (¬ H)) ((<=> P (^ A
H))) (<=> H P) nil) '((A T) (P NIL) (H NIL))); -> T ; Caso 2 con mas
parentesis
;;;        (interpretacion-modelo-p '((<=> A (¬ H)) (<=> P (^ A H))
(<=> H P)) '((A T) (P NIL) (H NIL))); -> nil ; Una kb con F no puede
tener modelos
;;;        (interpretacion-modelo-p nil '((A NIL) (P NIL) (H T))) ; ->
T pues trivialemente todos las fbf de la base de conocimiento tienen
el valor d verdad para esta interpretacion
;;;        (interpretacion-modelo-p '(((<=> ((A)) (¬ H))) (<=> P ((^ A
H))) (<=> H P)) nil)  ;-> nil pues no hay interpretaciones



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

```
;;;
;;;   cuales-son-modelos(kb inter)
;;;         de una base de conocimientos dada kb y una serie de
interpretaciones devuelve cuales son modelo
;;;
;;;   Entrada: kb lista de fbf que forman la base de conocimiento
;;;               inter lista de interpretaciones
;;;
;;;   Salida: lista con las interpretaciones modelo
;;;
;;;   PSEUDOCODIGO:
;;;         cuales-son-modelos(kb inter)
;;;               Si inter es lista vacia devolver nil
;;;               Si first(inter) es modelo de kb entonces
anadir(interModelo,first(inter))
;;;               cuales-son-modelos(rest(kb) inter)
;;;
;;;
;;; INPUT: kb lista de fbf que forman la base de conocimiento
;;;               inter lista de interpretaciones
;;;
;;;   OUTPUT: lista con las interpretaciones modelo
;;;

(defun cuales-son-modelos(kb inter)
  (cond ((or (null inter) (null kb)) nil) ;Caso base no quedan mas
interpretaciones que analizar o el argumento es nulo
        ((interpretacion-modelo-p kb (first inter)) ; Si la
interpretacion es modelo la anadimos a nuetra lista de
interpretaciones modelo
         (cons (first inter) (cuales-son-modelos kb (rest inter))))
        ((cuales-son-modelos kb (rest inter)))))     ;Si no lo es no
la anadimos a la lista de interpretaciones modelo


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;
;;;
;;;   encuentra-modelos(kb inter)
;;;         de una base de conocimientos dada kb devuelve sus
interpretaciones modelo
;;;
;;;   Entrada: kb lista de fbf que forman la base de conocimiento
;;;
;;;   Salida: lista con las interpretaciones modelo
;;;
;;;   PSEUDOCODIGO:
;;;         encuentra-modelos(kb)
;;;               cuales-son-modelos(kb (todas-las-posibles-
interpretaciones de los atomos que contituyen kb))
;;;
;;; INPUT: kb lista de fbf que forman la base de conocimiento
;;;
;;;   OUTPUT: lista con las interpretaciones modelo
;;;

(defun encuentra-modelos(kb)
  (cuales-son-modelos kb (genera-lista-interpretaciones (extrae-
simbolos kb)))) ;Generamos todas las interpretaciones posibles de la
FBF y vemos cuales son modelos.
```

```
;;; EVALUACION
;;;         CASOS TIPICOS
;;;               (encuentra-modelos '((=> A (¬ H)) (<=> P (^ A H)) (=>
H P))); -> (((A T) (P NIL) (H NIL)) ((A NIL) (P NIL) (H NIL)))
;;;               (encuentra-modelos '((=> (^ P I) L) (=> (¬ P) (¬ L))
(¬ P) L)); -> NIL
;;;
;;;         CASOS ESPECIALES
;;;               (encuentra-modelos nil) ; ->nil ; argumento vacio
;;;               (encuentra-modelos '(nil (v A (¬ A)))); ->nil ; una
kb con nil no tiene modelos
;;;               (encuentra-modelos '((v A (¬ A)))) ;-> (((A T)) ((A
NIL))) ;tautologia


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO SAT-p
;;
;; RECIBE : base de conocimiento (KB)
;; EVALÚA A : T si existe al menos un modelo para kb
;; NIL en caso contrario
;;
;;

(defun SAT-p (kb)
  (if (truth-value-p kb)
      kb
    (not (null (encuentra-modelos kb)))))

;;;; EJEMPLOS:
;;;  CASOS TIPICOS
;;;      (SAT-p '((<=> A (¬ H)) (<=> P (^ A H)) (<=> H P))) ;;; T
;;;      (SAT-p '((=> (^ P I) L) (=> (¬ P) (¬ L)) (¬ P) L)) ;;; NIL
;;;  CASOS ESPECIALES
;;;      (SAT-p 't)-> T ;el valor de verdad verdadero es satisfactible
;;;      (SAT-p 'nil); -> NIL ; Argumento vacio
;;;      (SAT-p '((<=> A (¬ H)) (<=> P (^ A H)) (<=> H P) nil)); ->nil
Una base de conocimiento con falso nunca tiene modelos


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO consecuencia-logica-p
;;
;; RECIBE : base de conocimiento (KB), fórmula bien formada (FBF)
;; EVALÚA A : T si FBF es consecuencia lógica de KB
;; NIL en caso contrario
;;

(defun consecuencia-logica-p (kb fbf)
  (if (null kb)
      t
    ;;Utilizando la reduccion al absurdo
    (not (sat-p (cons (list '¬ fbf) kb)))))

;;; EJEMPLOS:
;;; CASOS TIPICOS:
;;;    (consecuencia-logica-p '((<=> A (¬ H)) (<=> P (^ A H)) (<=> H
P)) 'A) ;-> T
;;;    (consecuencia-logica-p '((<=> A (¬ H)) (<=> P (^ A H)) (<=> H
P)) '(¬ A)) ;-> nil
```

```
;;; CASOS ESPECIALES:
;;;    (consecuencia-logica-p '((<=> A (¬ H)) (<=> P (^ A H)) (<=> H
P)) t) ;-> T ;verdadero es consecuencia logica de cualquier kb
;;;    (consecuencia-logica-p '((<=> A (¬ H)) (<=> P (^ A H)) (<=> H
P)) nil) ; -> NIL ; Argumento nulo
;;;    (consecuencia-logica-p '((=> (^ P I) L) (=> (¬ P) (¬ L)) (¬ P)
L) nil) ; -> T ;De una base insatisfactible se sigue cualquier cosa
;;;    (consecuencia-logica-p 'nil 'A); -> NIL ; De lo falso se sigue
cualquier cosa
```