Inteligencia artificial

Prática 1

UAM 2018

Óscar Gómez Borzdynski, José Ignacio Gómez García

Ejercicio 1.1:

- **PSEUDOCÓDIGO**

Entrada: Dos vectores (x e y).

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

Función general:

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

División entre:

Producto escalar (x y)

Raíz de multiplicación de:

Producto escalar (x x)

Producto escalar (y y)

Función producto escalar recursivo (a b):

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

Comprobar condiciones de error

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

Función producto escalar por mapcar (a b):

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

Suma todos los elementos

- **CÓDIGO**

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

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

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

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-rec (x y)
;;; Calcula la similitud coseno de un vector de forma recursiva
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT similitud coseno entre x e y
;;;
(defun sc-rec (x y)
 (if (or (null x) (null y)) ;; Comprobamos que no esten vacios para evitar divisiones entre 0.
    0
  (opera-con-error #'/
            (pe-rec x y)
            (sqrt (* (pe-rec x x)
                  (pe-rec y y))))))

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

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-mapcar (x y)
;;; Calcula la similitud coseno de un vector usando mapcar
;;;
;;; INPUT: x: vector, representado como una lista
```

```
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: similitud coseno entre x e y
;;;
(defun sc-mapcar (x y)
  (if (or (null x) (null y)) ;; Comprobamos que no esten vacios para evitar divisiones entre 0
      0
    (/ (pe-mapcar x y)
       (sqrt (* (pe-mapcar x x)
                (pe-mapcar y y))))))

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

- **COMENTARIOS**

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

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

Ejercicio 1.2:

- **PSEUDOCÓDIGO**

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

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

Función:

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

Ordenar los vectores resultantes por confianza

- **CÓDIGO**

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


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf (x vs conf)
;;; Devuelve aquellos vectores similares a una categoria
;;;
;;; INPUT: x: vector, representado como una lista
;;; vs: vector de vectores, representado como una lista de listas
;;; conf: Nivel de confianza
;;;
;;; OUTPUT: Vectores cuya similitud es superior al nivel de confianza, ordenados
;;;
(defun sc-conf (x vs conf)
  (mapcar #'second ;; Nos quedamos solo con el segundo
    (sort ;; De la lista ordenada de mayor a menor por el coseno
     (filtrar-vectores x vs conf) #'> :key #'first)))

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

- **COMENTARIOS**

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

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

Ejercicio 1.3:

- **PSEUDOCÓDIGO**

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

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

Función:

Para cada vector:

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

Obtener la tupla con mayor confianza

- **CÓDIGO**

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

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

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

;; Ejercicio 1.4
```

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

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

- **COMENTARIOS**

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

Aprovechamos para explicar también el ejercicio 1.4:

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

Tiempo de CPU de mapcar: 0.015625

Tiempo de CPU de recursivo: 0.109375

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

Ejercicio 2.1:

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

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

- **PSEUDOCÓDIGO**

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

- **CÓDIGO**

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

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


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

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

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

;; Use cases

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

- **COMENTARIOS**

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

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

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

Ejercicio 2.2:

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

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

- **PSEUDOCÓDIGO**

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

- **CÓDIGO**

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

;; Use cases

(allroot #'(lambda(x) (sin (* 6.28 x))) '(0.25 0.75 1.25 1.75 2.25) 0.0001)
;; -> (0.50027466 1.0005188 1.5007629 2.001007)

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

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

- **COMENTARIOS**

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

Ejercicio 2.3

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

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

- **PSEUDOCÓDIGO**

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

- ***CÓDIGO***

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

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

;; Use cases

(allind #'(lambda (x) (sin (* 6.28 x))) 0.1 2.25 1 0.0001) ;; -> NIL

(allind #'(lambda (x) (sin (* 6.28 x))) 0.1 2.25 2 0.0001)
;; -> (0.50027084 1.0005027 1.5007347 2.0010324)

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

- ***COMENTARIOS***

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

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

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

Ejercicio 3.1:

- **PSEUDOCÓDIGO**

Entrada: Un elemento y una lista

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

Función:

Para todos los elementos de la lista

Combinar dicho elemento con el elemento suministrado

Combinación:

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

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

Si son dos listas, concatenar ambas.

- **CÓDIGO**

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

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

- **COMENTARIOS**

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

Ejercicio 3.2:

- **PSEUDOCÓDIGO**

Entrada: Dos listas

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

Función:

Para cada elemento de una de las listas:

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

- **CÓDIGO**

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

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

- **CÓMENTARIOS**

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

Ejercicio 3.3:

- **PSEUDOCÓDIGO**

Entrada: Una lista de listas

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

Función:

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

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

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

- **CÓDIGO**

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

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

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

- **COMENTARIOS**

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

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

Ejercicio 4.1.1:

- **PSEUDOCÓDIGO**

Entrada: Una expresión

Salida: T si dicha expresión es un literal positivo, NIL en caso contrario

Función:

Para que x sea un literal positivo:

Tiene que ser un átomo de Lisp

No puede ser un conector

No puede ser un elemento de verdad de Lisp (T o NIL)

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.1
;; Predicado para determinar si una expresion en LISP
;; es un literal positivo
;;
;; RECIBE   : expresion
;; EVALUA A : T si la expresion es un literal positivo,
;;          NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun positive-literal-p (x)
  (and (atom x)
       (not (or (connector-p x)
                (truth-value-p x)))))

;; EJEMPLOS:
(positive-literal-p 'p)
;; evalua a T
(positive-literal-p T)
(positive-literal-p NIL)
(positive-literal-p '~)
(positive-literal-p '=>)
(positive-literal-p '(p))
(positive-literal-p '(~ p))
(positive-literal-p '(~ (v p q)))
;; evaluan a NIL
```

Ejercicio 4.1.2:

- **PSEUDOCÓDIGO**

Entrada: Una expresión

Salida: T si dicha expresión es un literal negativo, NIL en caso contrario

Función:

Para que x sea un literal negativo:

Tiene que ser una lista de Lisp

El primer elemento tiene que ser el conector not

El segundo elemento tiene que ser un literal positivo

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.2
;; Predicado para determinar si una expresion
;; es un literal negativo
;;
;; RECIBE   : expresion x
;; EVALUA A : T si la expresion es un literal negativo,
;;          NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun negative-literal-p (x)
  (and (listp x)
       (eql (first x) +not+)
       (positive-literal-p (second x))))

;; EJEMPLOS:
(negative-literal-p '(~ p))       ; T
(negative-literal-p NIL)          ; NIL
(negative-literal-p '~)           ; NIL
(negative-literal-p '=>)          ; NIL
(negative-literal-p '(p))         ; NIL
(negative-literal-p '((~ p)))     ; NIL
(negative-literal-p '(~ T))       ; NIL
(negative-literal-p '(~ NIL))     ; NIL
(negative-literal-p '(~ =>))      ; NIL
(negative-literal-p 'p)           ; NIL
(negative-literal-p '((~ p)))     ; NIL
(negative-literal-p '(~ (v p q)))  ; NIL
```

Ejercicio 4.1.3:

- **PSEUDOCÓDIGO**

Entrada: Una expresión

Salida: T si dicha expresión es un literal, NIL en caso contrario

Función:

Para que x sea un literal:

X puede ser un literal positivo o un literal negativo

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.3
;; Predicado para determinar si una expresion es un literal
;;
;; RECIBE   : expresion x
;; EVALUA A : T si la expresion es un literal,
;;          NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun literal-p (x)
  (or (positive-literal-p x) (negative-literal-p x)))

;; EJEMPLOS:
(literal-p 'p)
(literal-p '(~ p))
;;; evaluan a T
```

```
(literal-p '(p))
(literal-p '(~ (v p q)))
;;; evaluan a  NIL
```

Ejercicio 4.1.4:

- **PSEUDOCÓDIGO**

Entrada: Una expresión

Salida: T si dicha expresión está en formato infijo, NIL en caso contrario

Función:

Para que x esté en formato infijo:

X puede ser un literal o:

Si el primer elemento es un conector unario, el segundo tiene que ser una expresión en formato infijo.

Si el segundo elemento es un conector binario, el primer y el tercero tiene que ser expresiones en formato infijo.

Si el segundo elemento es un conector n-ario puede:

Estar solo

Primer y tercer elemento expresiones prefijo

Cuarto elemento un conector igual al segundo, entonces la lista a partir del tercero tiene que ser una expresión infijo.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.4
;; Predicado para determinar si una expresion esta en formato infijo
;;
;; RECIBE   : expresion x
;; EVALUA A : T si x esta en formato infijo,
;;          NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun wff-infix-p (x)
  (unless (null x)           ;; NIL no es FBF en formato infijo (por convencion)
    (or (literal-p x)        ;; Un literal es FBF en formato infijo
        (and (listp x)       ;; En caso de que no sea un literal debe ser una lista
             (cond
              ((connector-p (first x)) ;; Caso conector en primera posicion
               (cond
                ((unary-connector-p (first x)) ;; si es conector unario
                 (and (null (third x))       ;; deberia tener la estructura (<conector> FBF)
                      (wff-infix-p (second x))))
                ((n-ary-connector-p (first x)) ;; si es conector n-ario
                 (null (second x)))          ;; debería tener la estructura (<conector>)
                (t NIL)))
              ((binary-connector-p (second x))  ;; Caso conector binario
               (and (null (fourth x))          ;; deberia tener estructura (FBF <conector> FBF)
                    (wff-infix-p (first x))
                    (wff-infix-p (third x))))
              ((n-ary-connector-p (second x))   ;; Caso conector n-ario en segunda posicion
               (if (null (fourth x))           ;; formato (FBF <conector> FBF)
```

```
            (and (wff-infix-p (first x))
                (wff-infix-p (third x)))
           (and (eql (second x) (fourth x)) ;; formaro (FBF <conector> FBF <conector igual> ...)
               (wff-infix-p (first x))
               (wff-infix-p (rest (rest x))))))
         (t NIL))))))


;;
;; EJEMPLOS:
;;
(wff-infix-p 'a)                                                    ; T
(wff-infix-p '(^))                                          ; T  ;; por convencion
(wff-infix-p '(v))                                          ; T  ;; por convencion
(wff-infix-p '(A ^ (v)))                               ; T
(wff-infix-p '( a ^ b ^ (p v q) ^ (~ r) ^ s))          ; T
(wff-infix-p '(A => B))                                     ; T
(wff-infix-p '(A => (B <=> C)))                        ; T
(wff-infix-p '( B => (A ^ C ^ D)))                          ; T
(wff-infix-p '( B => (A ^ C)))                         ; T
(wff-infix-p '( B ^ (A ^ C)))                          ; T
(wff-infix-p '((p v (a => (b ^ (~ c) ^ d))) ^ ((p <=> (~ q)) ^ p ) ^ e))  ; T
(wff-infix-p nil)                                           ; NIL
(wff-infix-p '(a ^))                                        ; NIL
(wff-infix-p '(^ a))                                        ; NIL
(wff-infix-p '(a))                                          ; NIL
(wff-infix-p '((a)))                                   ; NIL
(wff-infix-p '((a) b))                                 ; NIL
(wff-infix-p '(^ a b q (~ r) s))                  ; NIL
(wff-infix-p '( B => A C))                             ; NIL
(wff-infix-p '( => A))                                         ; NIL
(wff-infix-p '(A =>))                                          ; NIL
(wff-infix-p '(A => B <=> C))                     ; NIL
(wff-infix-p '( B => (A ^ C v D)))                     ; NIL
(wff-infix-p '( B ^ C v D ))                           ; NIL
(wff-infix-p '((p v (a => e (b ^ (~ c) ^ d))) ^ ((p <=> (~ q)) ^ p ) ^ e)); NIL
```

Ejercicio 4.1.4:

- **PSEUDOCÓDIGO**

Entrada: Una expresión en formato infijo

Salida: La misma expresión en formato prefijo

Función:

Comprobar casos:

1) Un literal -> ya es prefijo
2) Conector unario -> unir dicho conector con la expresión en prefijo posterior
3) Conector binario -> hacer prefijo, conector, hacer prefijo
4) Conector n-ario
    a. Si el 4 elemento es NILL devolver igual que si fuera binario
    b. En caso de que haya mas elementos continuar la recursión y eliminar el conector con un rest

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.5
;;
```

```
;; Convierte FBF en formato infijo a FBF en formato prefijo
;;
;; RECIBE   : FBF en formato infijo
;; EVALUA A : FBF en formato prefijo
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun infix-to-prefix (wff)
  (when (wff-infix-p wff)
    (if (literal-p wff)
        wff
      (let ((first (first wff))
            (second (second wff)))
        (cond
         ((unary-connector-p first)
          (list first (infix-to-prefix second)))
         ((binary-connector-p second)
          (list second
                (infix-to-prefix first)
                (infix-to-prefix (third wff))))
         ((n-ary-connector-p second)
          (if (null (fourth wff))
              (list second
                    (infix-to-prefix first)
                    (infix-to-prefix (third wff)))
              (cons second
                    (cons (infix-to-prefix first)
                          (rest (infix-to-prefix (rest (rest wff))))))))
         (t NIL))))))

;;
;; EJEMPLOS
;;
(infix-to-prefix nil)      ;; NIL
(infix-to-prefix 'a)       ;; a
(infix-to-prefix '((a)))   ;; NIL
(infix-to-prefix '(a))     ;; NIL
(infix-to-prefix '(((a)))) ;; NIL
(prefix-to-infix (infix-to-prefix '((p v (a => (b ^ (~ c) ^ d))) ^ ((p <=> (~ q)) ^ p) ^ e)) )
;;-> ((P V (A => (B ^ (~ C) ^ D))) ^ ((P <=> (~ Q)) ^ P) ^ E)


(infix-to-prefix '((p v (a => (b ^ (~ c) ^ d))) ^  ((p <=> (~ q)) ^ p) ^ e))
;; (^ (V P (=> A (^ B (~ C) D))) (^ (<=> P (~ Q)) P) E)

(infix-to-prefix '(~ ((~ p) v q v (~ r) v (~ s))))
;; (~ (V (~ P) Q (~ R) (~ S)))


(infix-to-prefix
 (prefix-to-infix
  '(V (~ P) Q (~ R) (~ S))))
;;-> (V (~ P) Q (~ R) (~ S))

(infix-to-prefix
 (prefix-to-infix
  '(~ (V (~ P) Q (~ R) (~ S)))))
;;-> (~ (V (~ P) Q (~ R) (~ S)))


(infix-to-prefix 'a)  ; A
(infix-to-prefix '((p v (a => (b ^ (~ c) ^ d))) ^  ((p <=> (~ q)) ^ p) ^ e))
;; (^ (V P (=> A (^ B (~ C) D))) (^ (<=> P (~ Q)) P) E)

(infix-to-prefix '(~ ((~ p) v q v (~ r) v (~ s))))
```

```
;; (~ (V (~ P) Q (~ R) (~ S)))

(infix-to-prefix  (prefix-to-infix '(^ (v p (=> a (^ b (~ c) d)))))) ; '(v p (=> a (^ b (~ c) d))))
(infix-to-prefix  (prefix-to-infix '(^ (^ (<=> p (~ q)) p ) e))) ; '(^ (^ (<=> p (~ q)) p ) e))
(infix-to-prefix (prefix-to-infix '( v (~ p) q (~ r) (~ s))))  ; '( v (~ p) q (~ r) (~ s)))
;;;

(infix-to-prefix '(p v (a => (b ^ (~ c) ^ d)))) ; (V P (=> A (^ B (~ C) D)))
(infix-to-prefix '(((P <=> (~ Q)) ^ P) ^ E))  ; (^ (^ (<=> P (~ Q)) P) E)
(infix-to-prefix '((~ P) V Q V (~ R) V (~ S))); (V (~ P) Q (~ R) (~ S))
```

Ejercicio 4.1.5:

- **PSEUDOCÓDIGO**

Entrada: Una expresión en formato prefijo

Salida: T si dicha expresión es una cláusula disyuntiva, NIL en caso contrario

Función:

Para que x sea una cláusula disyuntiva en formato prefijo:

X tiene que ser una expresión prefijo y una lista

El primer elemento de X tiene que ser +or+

Puede:

No tener más elementos

El segundo elemento tiene que ser un literal y la lista resultante de eliminar el segundo elemento sea una cláusula.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.6
;; Predicado para determinar si una FBF es una clausula
;;
;; RECIBE   : FBF en formato prefijo
;; EVALUA A : T si FBF es una clausula, NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun clause-p (wff)
  (when (and (wff-prefix-p wff) (listp wff))
   (let ((connector (first wff)))
    (when (eql connector +or+)
     (let ((elements (rest wff)))1
     (cond ((null elements) ;;No hay más elementos, es clausula por convencion
           t)
          ((and (literal-p (first elements)) ;;Comprobamos el primer elemento
              (clause-p (cons connector (rest elements)))) ;;El resto tienen que seguir el mismo esquema
           t)
          (t NIL)))))) ;;Nunca deberia llegar aqui

;;
;; EJEMPLOS:
;;
(clause-p '(v))          ; T
(clause-p '(v p))         ; T
```

```
(clause-p '(v (~ r)))      ; T
(clause-p '(v p q (~ r) s)) ; T
(clause-p NIL)              ; NIL
(clause-p 'p)               ; NIL
(clause-p '(~ p))           ; NIL
(clause-p NIL)              ; NIL
(clause-p '(p))             ; NIL
(clause-p '((~ p)))         ; NIL
(clause-p '(^ a b q (~ r) s))    ; NIL
(clause-p '(v (^ a b) q (~ r) s)) ; NIL
(clause-p '(~ (v p q)))         ; NIL
```

Ejercicio 4.1.7:

- **PSEUDOCÓDIGO**

Entrada: Una expresión en formato prefijo

Salida: T si dicha expresión es una FNC, NIL en caso contrario

Función:

Para que x sea una FNC en formato prefijo:

X tiene que ser una expresión prefijo y una lista

El primer elemento de X tiene que ser +and+

Puede:

No tener más elementos

El segundo elemento tiene que ser una clausula y la lista resultante de eliminar el segundo elemento sea una fnc.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.7
;; Predicado para determinar si una FBF esta en FNC
;;
;; RECIBE   : FFB en formato prefijo
;; EVALUA A : T si FBF esta en FNC con conectores,
;;          NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun cnf-p (wff)
 (when (and (wff-prefix-p wff) (listp wff) (not (null wff)))
   (let ((connector (first wff)))
    (when (eql connector +and+)
      (let ((elements (rest wff)))
      (cond ((null elements) ;;No hay más elementos, es cnf por convencion
            t)
           ((and (clause-p (first elements)) ;;Comprobamos el primer elemento
                 (cnf-p (cons connector (rest elements)))) ;;El resto tienen que seguir el mismo esquema
            t)
           (t NIL)))))))) ;;Nunca deberia llegar aqui
;;
;; EJEMPLOS:
;;
(cnf-p '(^ (v a  b c) (v q r) (v (~ r) s) (v a b))) ; T
```

```
(cnf-p '(^ (v a  b (~ c)) ))                    ; T
(cnf-p '(^ ))                                  ; T
(cnf-p '(^(v )))                               ; T
(cnf-p '(~ p))                                 ; NIL
(cnf-p '(^ a b q (~ r) s))                     ; NIL
(cnf-p '(^ (v a b) q (v (~ r) s) a b))         ; NIL
(cnf-p '(v p q (~ r) s))                       ; NIL
(cnf-p '(^ (v a b) q (v (~ r) s) a b))         ; NIL
(cnf-p '(^ p))                                 ; NIL
(cnf-p '(v ))                                  ; NIL
(cnf-p NIL)                                    ; NIL
(cnf-p '((~ p)))                               ; NIL
(cnf-p '(p))                                   ; NIL
(cnf-p '(^ (p)))                               ; NIL
(cnf-p '((p)))                                 ; NIL
(cnf-p '(^ a b q (r) s))                       ; NIL
(cnf-p '(^ (v a  (v b c)) (v q r) (v (~ r) s) a b)) ; NIL
(cnf-p '(^ (v a (^ b c)) (^ q r) (v (~ r) s) a b))  ; NIL
(cnf-p '(~ (v p q)))                           ; NIL
(cnf-p '(v p q (r) s))                         ; NIL
```

Ejercicio 4.2.1:

- **PSEUDOCÓDIGO**

Entrada: Una expresión en formato prefijo

Salida: Una expresión sin bicondicionales

Función:

Comprobamos que no sea un literal o lista vacia.

Eliminamos la condicional con :

(a ⇔ b) = (^(a=>b)(b=>a))

Si no tiene bicondicional, comprobamos el resto de elementos.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.1: Incluya comentarios en el codigo adjunto
;;
;; Dada una FBF, evalua a una FBF equivalente
;; que no contiene el connector <=>
;;
;; RECIBE   : FBF en formato prefijo
;; EVALUA A : FBF equivalente en formato prefijo
;;          sin connector <=>
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-biconditional (wff)
  (if (or (null wff) (literal-p wff)) ;;En caso de que sea una lista vacía o un literal:
      wff                       ;; no tiene ningun bicondicional, por lo que no hace falta eliminar nada
    (let ((connector (first wff)))
      (if (eq connector +bicond+) ;;Si el conector es una bicondicional:
          (let ((wff1 (eliminate-biconditional (second wff)))  ;;Eliminamos la bicondicional de las dos
                (wff2 (eliminate-biconditional (third  wff)))) ;;FBF de la bicondicional
```

```
          (list +and+ ;; Creamos una lista de la forma (^(w1 -> w2) (w2 -> w1))
               (list +cond+ wff1 wff2)
               (list +cond+ wff2 wff1)))
        (cons connector ;; Si no es una bicondicional tenemos que analizar el resto de FBF
             (mapcar #'eliminate-biconditional (rest wff))))))

;;
;; EJEMPLOS:
;;
(eliminate-biconditional '(<=> p  (v q s p) ))
;;   (^ (=> P (v Q S P)) (=> (v Q S P) P))
(eliminate-biconditional '(<=>  (<=> p  q) (^ s (~ q))))
;;   (^ (=> (^ (=> P Q) (=> Q P)) (^ S (~ Q)))
;;      (=> (^ S (~ Q)) (^ (=> P Q) (=> Q P))))
```

Ejercicio 4.2.2:

- **PSEUDOCÓDIGO**

Entrada: Una expresión en formato prefijo

Salida: Una expresión sin condicionales

Función:

Comprobamos que no sea un literal o lista vacia.

Eliminamos la condicional con :

(a=> b) = (v(~a)(b))

Si no tiene condicional, comprobamos el resto de elementos.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.2
;; Dada una FBF, que contiene conectores => evalua a
;; una FBF equivalente que no contiene el connector =>
;;
;; RECIBE   : wff en formato prefijo sin el connector <=>
;; EVALUA A : wff equivalente en formato prefijo
;;            sin el connector =>
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-conditional (wff)
 (if (or (null wff) (literal-p wff))  ;;En caso de que sea una lista vacía o un literal:
      wff                         ;; no tiene ningun condicional, por lo que no hace falta eliminar nada
    (let ((connector (first wff)))
      (if (eq connector +cond+) ;;Si el conector es una condicional:
         (let ((wff1 (eliminate-conditional (second wff)))  ;;Eliminamos la condicional de las dos
              (wff2 (eliminate-conditional (third  wff)))) ;;FBF de la condicional
           (list +or+ ;; Creamos una lista de la forma (^(w1 -> w2) (w2 -> w1))
                (list +not+ wff1)
                wff2))
        (cons connector ;; Si no es una condicional tenemos que analizar el resto de FBF
             (mapcar #'eliminate-conditional (rest wff))))))

;;
;; EJEMPLOS:
;;
(eliminate-conditional '(=> p q))               ;;; (V (~ P) Q)
(eliminate-conditional '(=> p (v q s p)))           ;;; (V (~ P) (V Q S P))
```

(eliminate-conditional '(=> (=> (~ p) q) (^ s (~ q)))) ;;; (V (~ (V (~ (~ P)) Q)) (^ S (~ Q)))

Ejercicio 4.2.3:

- **PSEUDOCÓDIGO**

Entrada: Una expresión en formato prefijo

Salida: Una expresión donde las únicas negaciones son átomos

Función:

Comprobamos que no sea un literal o lista vacia.

Comprobamos dobles negaciones.

Aplicamos las leyes de De Morgan.

Comprobamos el resto de elementos.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.3
;; Dada una FBF, que no contiene los conectores <=>, =>
;; evalua a una FNF equivalente en la que la negacion
;; aparece unicamente en literales negativos
;;
;; RECIBE   : FBF en formato prefijo sin conector <=>, =>
;; EVALUA A : FBF equivalente en formato prefijo en la que
;;          la negacion  aparece unicamente en literales
;;          negativos.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun reduce-scope-of-negation (wff)
 (if (or (null wff) (literal-p wff))  ;; En caso de que sea una lista vacía o un literal:
    wff                          ;; no tiene negaciones.
  (let ((connector (first wff)))
    (if (eq connector +not+) ;; Si el conector es una negación:
       (let ((connector-2 (first (second wff)))
             (args (rest (second wff))))
         (cond
          ((eq connector-2 +not+) ;; Doble negación
           (reduce-scope-of-negation (first args)))
          ((n-ary-connector-p connector-2) ;;De Morgan
           (cons (exchange-and-or connector-2) ;; Cambiamos signo y creamos una nueva lista
                 (mapcar #'(lambda (x)       ;; con todos los elementos con las negaciones reducidas
                             (reduce-scope-of-negation (list +not+
                                                             x)))
                         args)))
          (t (list +not+
                   (reduce-scope-of-negation (rest wff))))))) ;; Ultimo caso, revisamos argumentos
      (cons connector ;; Si no es una negacion tenemos que analizar el resto de FBF
            (mapcar #'reduce-scope-of-negation (rest wff)))))))

;;
;;  EJEMPLOS:
;;
(reduce-scope-of-negation '(~ (v p (~ q) r)))
;;; (^ (~ P) Q (~ R))
(reduce-scope-of-negation '(~ (^ p (~ q) (v  r s (~ a)))))
;;;  (V (~ P) Q (^ (~ R) (~ S) A))
```

Ejercicio 4.2.4:

- **PSEUDOCÓDIGO**

Entrada: Una expresión en formato prefijo

Salida: Una expresión en fnc

Función:

Comprobamos que no sea una fnc.

En literales, transformamos a fnc con (^(v a)).

Si el conector es ^ tenemos que mirar los elementos aplicados.

Si el conector es v transformamos a fnc:

Cambiamos el conector y procesamos los elementos, creando cláusulas en cada uno de ellos.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.4: Comente el codigo adjunto
;;
;; Dada una FBF, que no contiene los conectores <=>, => en la
;; que la negacion aparece unicamente en literales negativos
;; evalua a una FNC equivalente en FNC con conectores ^, v
;;
;; RECIBE   : FBF en formato prefijo sin conector <=>, =>,
;;          en la que la negacion aparece unicamente
;;          en literales negativos
;; EVALUA A : FBF equivalente en formato prefijo FNC
;;          con conectores ^, v
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Función que combina un elt con todos los elementos de una lista
(defun combine-elt-lst (elt lst)
  (if (null lst)
      (list (list elt))
    (mapcar #'(lambda (x) (cons elt x)) lst))) ;; Crea un cons con los elementos correspondientes

;; Función que cambia entre formas normales
(defun exchange-NF (nf)
  (if (or (null nf) (literal-p nf)) ;; Si es NIL o un solo literal:
      nf                        ;; está en ambas formas normales
    (let ((connector (first nf)))
      (cons (exchange-and-or connector) ;; Cambiar conector (para cambiar de FN)
          (mapcar #'(lambda (x)      ;; Aplicar NF_AUX al resto y combinarlo con el conector
                    (cons connector x))
            (exchange-NF-aux (rest nf)))))))

;; Función auxiliar para exchange-NF-aux
(defun exchange-NF-aux (nf)
  (if (null nf) ;; Si es NIL, no lo procesamos.
      NIL      ;; COMENTARIO: pensamos que esto se podría simplificar con un unless
    (let ((lst (first nf)))
      (mapcan #'(lambda (x)
              (combine-elt-lst  ;; Combinamos cada elemento con la expresión
               x              ;; obtenida de cambiar la FN del resto de elementos
```

```
            (exchange-NF-aux (rest nf))))
      (if (literal-p lst) (list lst) (rest lst))))))

;; Simplicar un FN
(defun simplify (connector lst-wffs )
 (if (literal-p lst-wffs) ;; Si es un literal, no tenemos que simplificarlo
    lst-wffs
  (mapcan #'(lambda (x) ;; A todos los elementos
           (cond
            ((literal-p x) (list x)) ;; Si es un literal, crear una lista con él
            ((equal connector (first x)) ;; Si es un conector:
             (mapcan ;; A todos los elementos aplicados al conector
                #'(lambda (y) (simplify connector (list y))) ;; Simplificamos el elemento
              (rest x))) ;; Por tanto devolvemos la lista sin el conector
            (t (list x))))
    lst-wffs)))

;; Convierte una FBF a FNC
(defun cnf (wff)
 (cond
  ((cnf-p wff) wff) ;; Si ya es cnf, no procesamos
  ((literal-p wff)  ;; Si es un literal lo cambiamos a (^(v a))
   (list +and+ (list +or+ wff)))
  ((let ((connector (first wff)))
    (cond
     ((equal +and+ connector) ;; Si el conector es de tipo and
      (cons +and+ (simplify +and+ (mapcar #'cnf (rest wff))))) ;; Simplificamos el resto de elementos
     ((equal +or+ connector) ;; Si el conector es de tipo or
      (cnf (exchange-NF (cons +or+ (simplify +or+ (rest wff)))))))))) ;; Transformamos a fnc con exchange NF


(cnf 'a)

(cnf '(v (~ a) b c))
(print (cnf '(^ (v (~ a) b c) (~ e) (^ e f (~ g) h) (v m n) (^ r s q) (v u q) (^ x y))))
(print (cnf '(v (^ (~ a) b c) (~ e) (^ e f (~ g) h) (v m n) (^ r s q) (v u q) (^ x y))))
(print (cnf '(^ (v p  (~ q)) a (v k  r  (^ m  n)))))
(print (cnf '(v p  q  (^ r  m)  (^ n  a)  s )))
(exchange-NF '(v p  q  (^ r  m)  (^ n  a)  s ))
(cnf '(^ (v a b (^ y r s) (v k l)) c (~ d) (^ e f (v h i) (^ o p))))
(cnf '(^ (v a b (^ y r s)) c (~ d) (^ e f (v h i) (^ o p))))
(cnf '(^ (^ y r s (^ p q (v c d))) (v a b)))
(print (cnf '(^ (v (~ a) b c) (~ e) r s
           (v e f (~ g) h) k (v m n) d)))
;;
(cnf '(^ (v p (~ q)) (v k r (^ m  n))))
(print  (cnf '(v (v p q) e f (^ r  m) n (^ a (~ b) c) (^ d s))))
(print (cnf '(^ (^ (~ y) (v r (^ s (~ x)) (^ (~ p) m (v c d))) (v (~ a) (~ b))) g)))
;;
;; EJEMPLOS:
;;
(cnf NIL)            ; NIL
(cnf 'a)             ; (^ (V A))
(cnf '(~ a))         ; (^ (V (~ A)))
(cnf '(V (~ P) (~ P))) ; (^ (V (~ P) (~ P)))
(cnf '(V A))         ; (^ (V A))
(cnf '(^ (v p (~ q)) (v k r (^ m  n))))
;;;   (^ (V P (~ Q)) (V K R M) (V K R N))
(print  (cnf '(v (v p q) e f (^ r  m) n (^ a (~ b) c) (^ d s))))
;;; (^ (V P Q E F R N A D)     (V P Q E F R N A S)
;;;    (V P Q E F R N (~ B) D)  (V P Q E F R N (~ B) S)
;;;    (V P Q E F R N C D)     (V P Q E F R N C S)
;;;    (V P Q E F M N A D)     (V P Q E F M N A S)
```

```
;;;   (V P Q E F M N (~ B) D)  (V P Q E F M N (~ B) S)
;;;   (V P Q E F M N C D)     (V P Q E F M N C S))
;;;
(print
 (cnf '(^ (^ (~ y) (v r (^ s (~ x))
                 (^ (~ p) m (v c d)))(v (~ a) (~ b))) g)))
;;;(^ (V (~ Y)) (V R S (~ P)) (V R S M)
;;;  (V R S C D) (V R (~ X) (~ P))
;;;  (V R (~ X) M) (V R (~ X) C D)
;;;  (V (~ A) (~ B)) (V G))
```

Ejercicio 4.2.5:

- **PSEUDOCÓDIGO**

Entrada: Una fnc

Salida: Una lista de listas sin conectores

Función:

Si estamos ante una cnf:

Del segundo elemento en adelante:

Si es un literal: no hacemos nada

Si no lo es: eliminamos los conectores de las subexpresiones.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.5:
;;
;; Dada una FBF en  FNC
;; evalua a lista de listas sin conectores
;; que representa una conjuncion de disyunciones de literales
;;
;; RECIBE   : FBF en FNC con conectores ^, v
;; EVALUA A : FBF en FNC (con conectores ^, v eliminados)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-rec(cnf)
  (mapcar #'(lambda (x)
          (if (literal-p x)
            x
            (eliminate-rec x)))
    (rest cnf)))

(defun eliminate-connectors (cnf)
  (when (cnf-p cnf))
  (eliminate-rec cnf))

(eliminate-connectors 'nil) ;; nil
(eliminate-connectors (cnf '(^ (v p  (~ q))  (v k  r  (^ m  n)))))
(eliminate-connectors
 (cnf '(^ (v (~ a) b c) (~ e) (^ e f (~ g) h) (v m n) (^ r s q) (v u q) (^ x y))))

(eliminate-connectors (cnf '(v p  q  (^ r  m)  (^ n  q)  s )))
(eliminate-connectors (print (cnf '(^ (v p  (~ q)) (~ a) (v k  r  (^ m  n))))))

(eliminate-connectors '(^))
(eliminate-connectors '(^ (v p (~ q)) (v) (v k r)))
```

```
(eliminate-connectors '(^ (v a b)))

;;   EJEMPLOS:
;;

(eliminate-connectors '(^ (v p (~ q)) (v k r)))
;; ((P (~ Q)) (K R))
(eliminate-connectors '(^ (v p (~ q)) (v q (~ a)) (v s e f) (v b)))
;; ((P (~ Q)) (Q (~ A)) (S E F) (B))
```

Ejercicio 4.2.6:

- **PSEUDOCÓDIGO**

Entrada: Una expresión en formato infijo

Salida: Una lista de listas sin conectores

Función:

Transformarla a prefijo

Eliminar bicondicionales y condicionales

Reducir la negación

Convertir a cnf

Eliminar conectores

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.6
;; Dada una FBF en formato infijo
;; evalua a lista de listas sin conectores
;; que representa la FNC equivalente
;;
;; RECIBE   : FBF
;; EVALUA A : FBF en FNC (con conectores ^, v eliminados)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun wff-infix-to-cnf (wff)
  (eliminate-connectors          ;; Eliminamos conectores
   (cnf                          ;; De una cnf
    (reduce-scope-of-negation    ;; A la que hemos reducido las negaciones
     (eliminate-conditional      ;; Eliminado las condicionales
      (eliminate-biconditional   ;; Y bicondicionales
       (infix-to-prefix wff)))))) ;; De la fbf en formato prefijo

;;
;; EJEMPLOS:
;;
(wff-infix-to-cnf 'a)
(wff-infix-to-cnf '(~ a))
(wff-infix-to-cnf  '( (~ p) v q v (~ r) v (~ s)))
(wff-infix-to-cnf  '((p v (a => (b ^ (~ c) ^ d))) ^ ((p <=> (~ q)) ^ p) ^ e))
;; ((P (~ A) B) (P (~ A) (~ C)) (P (~ A) D) ((~ P) (~ Q)) (Q P) (P) (E))
```

Ejercicio 4.3.1:

- **PSEUDOCÓDIGO**

Entrada: Una clausula

Salida: La misma clausula sin entradas repetidas

Función:

Eliminamos todos los elementos repetidos

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.1
;; eliminacion de literales repetidos una clausula
;;
;; RECIBE   : K - clausula (lista de literales, disyuncion implicita)
;; EVALUA A : clausula equivalente sin literales repetidos
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-repeated-literals (k)
  (remove-duplicates k :test #'equal))

;;
;; EJEMPLO:
;;
(eliminate-repeated-literals '(a b (~ c) (~ a) a c (~ c) c a))
;;;   (B (~ A) (~ C) C A)
```

Ejercicio 4.3.2:

- **PSEUDOCÓDIGO**

Entrada: Una FNC

Salida: La misma FNC sin clausulas repetidas

Función:

Eliminamos todas las clausulas repetidas, comprobamos que las clausulas no esten en mismo orden

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.2
;; eliminacion de clausulas repetidas en una FNC
;;
;; RECIBE   : cnf - FBF en FNC (lista de clausulas, conjuncion implicita)
;; EVALUA A : FNC equivalente sin clausulas repetidas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun equal-clauses (a b) ;; Comprueba si dos clausulas son iguales
  (and (subsetp a b :test #'equal)   ;; a contenido en b
       (subsetp b a :test #'equal))) ;; b contenido en a

(defun eliminate-repeated-clauses (cnf)
  (remove-duplicates
   (mapcar #'eliminate-repeated-literals
     cnf)
   :test #'equal-clauses))
```

```
;;
;; EJEMPLO:
;;
(eliminate-repeated-clauses '(((~ a) c) (c (~ a)) ((~ a) (~ a) b c b) (a a b) (c (~ a) b  b) (a b)))
;;; ((C (~ A)) (C (~ A) B) (A B))
```

Ejercicio 4.3.3:

- **PSEUDOCÓDIGO**

Entrada: Dos cláusulas

Salida: (K1) o F, si K1 subsume a K2

Función:

Si todos los elementos de K1 están en K2, devolvemos (K1), si no: NIL

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.3
;; Predicado que determina si una clausula subsume otra
;;
;; RECIBE   : K1, K2 clausulas
;; EVALUA a : (list K1) si K1 subsume a K2
;;          NIL en caso contrario
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun subsume (K1 K2)
  (when (subsetp K1 K2 :test #'equal)
    (list k1)))

;;
;;  EJEMPLOS:
;;
(subsume '(a) '(a b (~ c)))
;; ((a))
(subsume NIL '(a b (~ c)))
;; (NIL)
(subsume '(a b (~ c)) '(a) )
;; NIL
(subsume '( b (~ c)) '(a b (~ c)) )
;; (( b (~ c)))
(subsume '(a b (~ c)) '( b (~ c)))
;; NIL
(subsume '(a b (~ c)) '(d  b (~ c)))
;; nil
(subsume '(a b (~ c)) '((~ a) b (~ c) a))
;; ((A B (~ C)))
(subsume '((~ a) b (~ c) a) '(a b (~ c)) )
;; nil
```

Ejercicio 4.3.4:

- **PSEUDOCÓDIGO**

Entrada: Una CNF

Salida: La misma CNF sin clausulas subsumidas

Función:

Para cada elemento comprobar si es subsumido. Si no lo es, añadirlo al resultado final.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.4
;; eliminacion de clausulas subsumidas en una FNC
;;
;; RECIBE   : cnf (FBF en FNC)
;; EVALUA A : FBF en FNC equivalente a cnf sin clausulas subsumidas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun is-subsumed (elt lst)
  (cond
    ((null lst) ;; Caso base, no ha sido subsumido
      NIL)
    ((subsume (first lst) elt) ;; Si es subsumido, devuelve True
      T)
    (T
      (is-subsumed elt (rest lst))))) ;; Recursion


(defun eliminate-subsumed-clauses-rec (processed new)
  (if (null new)
    processed
   (let ((f (first new)) (r (rest new)))
    (if (or (is-subsumed f processed) ;; Si f es subsumido por alguna clausula, no se añade a processed
          (is-subsumed f r))
      (eliminate-subsumed-clauses-rec processed r)
     (eliminate-subsumed-clauses-rec (cons f processed) r))))) ;; Si no ha sido subsumido, se añade a processed


(defun eliminate-subsumed-clauses (cnf)
  (eliminate-subsumed-clauses-rec nil cnf))

;;
;;  EJEMPLOS:
;;
(eliminate-subsumed-clauses
 '((a b c) (b c) (a (~ c) b)  ((~ a) b) (a b (~ a)) (c b a)))
;;; ((A (~ C) B) ((~ A) B) (B C)) ;; el orden no es importante
(eliminate-subsumed-clauses
 '((a b c) (b c) (a (~ c) b) (b)  ((~ a) b) (a b (~ a)) (c b a)))
;;; ((B))
(eliminate-subsumed-clauses
 '((a b c) (b c) (a (~ c) b) ((~ a))  ((~ a) b) (a b (~ a)) (c b a)))
;;; ((A (~ C) B) ((~ A)) (B C))
```

Ejercicio 4.3.5:

- **PSEUDOCÓDIGO**

Entrada: Una cláusula

Salida: T si es una tautología, NIL en caso contrario

Función:

Cogemos el primer elemento y su negado.

Si el negado está en la lista (sin el primer elemento) -> T

Si no está comprobamos el siguiente elemento.

Si ninguno falla, devuelve NIL

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.5
;; Predicado que determina si una clausula es tautologia
;;
;; RECIBE   : K (clausula)
;; EVALUA a : T si K es tautologia
;;          NIL en caso contrario
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun change-sign (K)
  (if (positive-literal-p K) ;; Negación del primer elemento
      (list +not+ K)
    (second K)))

(defun tautology-p (K)
  (unless (null K)
    (let* ((el (first K))
           (nel (change-sign el)))
      (if (member nel K :test #'equal)
          T
        (tautology-p (rest K))))))

;;
;;  EJEMPLOS:
;;
(tautology-p '((~ B) A C (~ A) D)) ;;; T
(tautology-p '((~ B) A C D))       ;;; NIL
```

Ejercicio 4.3.6:

- **PSEUDOCÓDIGO**

Entrada: Una cnf

Salida: La misma cnf sin tautologías

Función:

Para cada cláusula se comprueba si es una tautología y se elimina

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.6
;; eliminacion de clausulas en una FBF en FNC que son tautologia
;;
;; RECIBE   : cnf - FBF en FNC
;; EVALUA A : FBF en FNC equivalente a cnf sin tautologias
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-tautologies (cnf)
  (remove-if #'tautology-p cnf))

;;
;;  EJEMPLOS:
;;
(eliminate-tautologies
 '(((~ b) a) (a (~ a) b c) ( a (~ b)) (s d (~ s) (~ s)) (a)))
;; (((~ B) A) (A (~ B)) (A))

(eliminate-tautologies '((a (~ a) b c)))
;; NIL
```

Ejercicio 4.3.7:

- **PSEUDOCÓDIGO**

Entrada: Una cnf

Salida: una cnf simplificada

Función:

Eliminamos literales repetidos en cada clausula

Eliminamos clausulas repetidas

Eliminamos tautologías

Eliminamos clausulas subsumidas

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.7
;; simplifica FBF en FNC
;;       * elimina literales repetidos en cada una de las clausulas
;;       * elimina clausulas repetidas
;;       * elimina tautologias
;;       * elimina clausulass subsumidas
;;
;; RECIBE   : cnf  FBF en FNC
;; EVALUA A : FNC equivalente sin clausulas repetidas,
;;           sin literales repetidos en las clausulas
;;           y sin clausulas subsumidas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun simplify-cnf (cnf)
  (eliminate-subsumed-clauses
   (eliminate-tautologies
    (eliminate-repeated-clauses
     (mapcar #'eliminate-repeated-literals cnf)))))

;;
;;  EJEMPLOS:
;;
(simplify-cnf '((a a) (b) (a) ((~ b)) ((~ b)) (a b c a)  (s s d) (b b c a b)))
;; ((B) ((~ B)) (S D) (A)) ;; en cualquier orden
```

Ejercicio 4.4.1:

- **PSEUDOCÓDIGO**

Entrada: Una cnf

Un literal

Salida: La lista de cláusulas neutrales de cnf con respecto al literal

Función:

Eliminar una cláusula si lambda o no lambda son miembros de ella.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.1
;; Construye el conjunto de clausulas lambda-neutras para una FNC
;;
;; RECIBE   : cnf    - FBF en FBF simplificada
;;          lambda - literal positivo
;; EVALUA A : cnf_lambda^(0) subconjunto de clausulas de cnf
;;          que no contienen el literal lambda ni ~lambda
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun extract-neutral-clauses (lambda cnf)
  (let ((nlambda (change-sign lambda)))
    (remove-if #'(lambda (x) (or (member lambda x :test #'equal)
                                 (member nlambda x :test #'equal)))
               cnf))) ;;Si lambda y nlambda estan en una clausla, la quitamos

;;
;;  EJEMPLOS:
;;
(extract-neutral-clauses 'p
                         '((p (~ q) r) (p q) (r (~ s) q) (a b p) (a (~ p) c) ((~ r) s)))
;; ((R (~ S) Q) ((~ R) S))


(extract-neutral-clauses 'r NIL)
;; NIL

(extract-neutral-clauses 'r '(NIL))
;; (NIL)

(extract-neutral-clauses 'r
                         '((p (~ q) r) (p q) (r (~ s) q) (a b p) (a (~ p) c) ((~ r) s)))
;; ((P Q) (A B P) (A (~ P) C))

(extract-neutral-clauses 'p
                         '((p (~ q) r) (p q) (r (~ s) p q) (a b p) (a (~ p) c) ((~ r) p s)))
;; NIL
```

Ejercicio 4.4.2:

- **PSEUDOCÓDIGO**

Entrada: Una cnf

Un literal

Salida: La lista de cláusulas positivas de cnf con respecto al literal

Función:

Eliminar toda cláusula que no contiene dicho literal

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.2
;; Construye el conjunto de clausulas lambda-positivas para una FNC
;;
;; RECIBE   : cnf    - FBF en FNC simplificada
;;          lambda - literal positivo
;; EVALUA A : cnf_lambda^(+) subconjunto de clausulas de cnf
;;          que contienen el literal lambda
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun extract-positive-clauses (lambda cnf)
```

```
   (remove-if #'(lambda (x) (not (member lambda x :test #'equal)))
         cnf)) ;;Si lambda no esta en una clausla, la quitamos
;;
;;  EJEMPLOS:
;;
(extract-positive-clauses 'p
                  '((p (~ q) r) (p q) (r (~ s) q) (a b p) (a (~ p) c) ((~ r) s)))

;; ((P (~ Q) R) (P Q) (A B P))


(extract-positive-clauses 'r NIL)
;; NIL
(extract-positive-clauses 'r '(NIL))
;; NIL
(extract-positive-clauses 'r
                  '((p (~ q) r) (p q) (r (~ s) q) (a b p) (a (~ p) c) ((~ r) s)))
;; ((P (~ Q) R) (R (~ S) Q))
(extract-positive-clauses 'p
                  '(((~ p) (~ q) r) ((~ p) q) (r (~ s) (~ p) q) (a b (~ p)) ((~ r) (~ p) s)))
;; NIL
```

Ejercicio 4.4.3:

- **PSEUDOCÓDIGO**

Entrada: Una cnf

Un literal

Salida: La lista de cláusulas positivas de cnf con respecto al literal

Función:

Eliminar toda cláusula que no contiene al negado de dicho literal

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.3
;; Construye el conjunto de clausulas lambda-negativas para una FNC
;;
;; RECIBE   : cnf    - FBF en FNC simplificada
;;          lambda - literal positivo
;; EVALUA A : cnf_lambda^(-) subconjunto de clausulas de cnf
;;          que contienen el literal ~lambda
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun extract-negative-clauses (lambda cnf)
  (let ((nlambda (change-sign lambda)))
    (remove-if #'(lambda (x) (not (member nlambda x :test #'equal)))
          cnf))) ;;Si nlambda no esta en una clausla, la quitamos

;;
;;  EJEMPLOS:
;;
(extract-negative-clauses 'p
                  '((p (~ q) r) (p q) (r (~ s) q) (a b p) (a (~ p) c) ((~ r) s)))
;; ((A (~ P) C))

(extract-negative-clauses 'r NIL)
;; NIL
(extract-negative-clauses 'r '(NIL))
;; NIL
```

```
(extract-negative-clauses 'r
                    '((p (~ q) r) (p q) (r (~ s) q) (a b p) (a (~ p) c) ((~ r) s)))
;; (((~ R) S))
(extract-negative-clauses 'p
                    '(( p (~ q) r) ( p q) (r (~ s) p q) (a b p) ((~ r) p s)))
;; NIL
```

Ejercicio 4.4.4:

- **PSEUDOCÓDIGO**

Entrada: Una elemento y dos cláusulas

Salida: El resolvente de ambas cláusula respecto al elemento

Función:

Sea nlambda la negación de lambda

Si lambda pertenece a K1:

Si nlambda pertenece a K2:

Unir K1 y K2 quitando lambda a K1 y nlambda a K2.

Si nlambda pertenece a K1:

Si lambda pertenece a K2:

Unir K1 y K2 quitando nlambda a K1 y lambda a K2.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.4
;; resolvente de dos clausulas
;;
;; RECIBE   : lambda      - literal positivo
;;          K1, K2      - clausulas simplificadas
;; EVALUA A : res_lambda(K1,K2)
;;                     - lista que contiene la
;;                       clausula que resulta de aplicar resolucion
;;                       sobre K1 y K2, con los literales repetidos
;;                       eliminados
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun resolve-on (lambda K1 K2)
  (let ((nlambda (change-sign lambda)))
  (cond ((member lambda K1 :test #'equal)
        (when (member nlambda K2 :test #'equal)
          (list (union (remove lambda K1 :test #'equal)
               (remove nlambda K2 :test #'equal)
               :test #'equal))))
       ((member nlambda K1 :test #'equal)
        (when (member lambda K2 :test #'equal)
          (list (union (remove nlambda K1 :test #'equal)
               (remove lambda K2 :test #'equal)
```

```
            :test #'equal))))
      (t NIL))))
;;
;;  EJEMPLOS:
;;
(resolve-on 'p '(a b (~ c) p) '((~ p) b a q r s))
;; (((~ C) B A Q R S))

(resolve-on 'p '(a b (~ c) (~ p)) '( p b a q r s))
;; (((~ C) B A Q R S))

(resolve-on 'p '(p) '((~ p)))
;; (NIL)


(resolve-on 'p NIL '(p b a q r s))
;; NIL

(resolve-on 'p NIL NIL)
;; NIL

(resolve-on 'p '(a b (~ c) (~ p)) '(p b a q r s))
;; (((~ C) B A Q R S))

(resolve-on 'p '(a b (~ c)) '(p b a q r s))
;; NIL
```

Ejercicio 4.4.5:

- **PSEUDOCÓDIGO**

Entrada: Una cnf y un elemento (lambda)

Salida: El RES de la cnf respecto a lambda

Función:

Une:

Cláusulas neutras de cnf respecto a lambda

Para todas las cláusulas positivas de cnf:

Resolver con todas las cláusulas negativas de cnf

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.5
;; Construye el conjunto de clausulas RES para una FNC
;;
;; RECIBE   : lambda - literal positivo
;;          cnf    - FBF en FNC simplificada
;;
;; EVALUA A : RES_lambda(cnf) con las clauses repetidas eliminadas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun resolve-on-list (lambda K1 lst)
  (mapcar #'(lambda (x) (resolve-on lambda K1 x))
    lst))

(defun build-RES (lambda cnf)
```

```
 (let ((nc (extract-negative-clauses lambda cnf))
       (pc (extract-positive-clauses lambda cnf)))
  (union (extract-neutral-clauses lambda cnf)
         (mapcan #'(lambda (x) (first (resolve-on-list lambda x nc)))
                 pc)
         :test #'equal-clauses)))
;;
;;  EJEMPLOS:
;;
(build-RES 'p NIL)
;; NIL
(build-RES 'P '((A  (~ P) B) (A P) (A B)));; ((A B))
(build-RES 'P '((B  (~ P) A) (A P) (A B)));; ((B A))

(build-RES 'p '(NIL))
;; (NIL)

(build-RES 'p '((p) ((~ p))))
;; (NIL)

(build-RES 'q '((p q) ((~ p) q) (a b q) (p (~ q)) ((~ p) (~ q))))
;; ((P) ((~ P) P) ((~ P)) (B A P) (B A (~ P)))

(build-RES 'p '((p q) (c q) (a b q) (p (~ q)) (p (~ q))))
;; ((A B Q) (C Q))
```

Ejercicio 4.5:

- **PSEUDOCÓDIGO**

Entrada: Una cnf

Salida: T si la cnf es SAT, NIL si es UNSAT

Función:

Para cada literal de cnf:

Si es (NIL) -> NIL

Si es NIL -> T

Resolver dicho literal y pasar al siguiente con el resultado obtenido.

- **CÓDIGO**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.5
;; Comprueba si una FNC es SAT calculando RES para todos los
;; atomos en la FNC
;;
;; RECIBE   : cnf - FBF en FNC simplificada
;; EVALUA A :          T  si cnf es SAT
;;             NIL  si cnf es UNSAT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun get-literals-aux (K1 els) ;;Devuelve una lista con los literales no repetidos de una clausula
  (if (null K1)
      els
    (if (or (member (first K1) ;;Si esta en positivo
                    els
                    :test #'equal)
```

```
        (member (change-sign (first K1)) ;;Si esta en negativo
              els
              :test #'equal))
      (get-literals-aux (rest K1) ;;No introducir el literal
                  els)
    (get-literals-aux (rest K1) ;;Introducir el literal
                (cons (first K1) els)))))

(defun get-literals (cnf els) ;;Devuelve una lista con los literales no repetidos de una fnc
  (if (null cnf)
     els
   (get-literals (rest cnf) (get-literals-aux (first cnf) els))))

(defun  RES-SAT-p-aux (cnf literals)
  (cond ((null cnf) t) ;;Tautología
       ((null (first cnf)) NIL) ;;Contradicción
       (t (RES-SAT-p-aux (simplify-cnf
                    (build-RES
                     (first literals) cnf))
                   (rest literals))))) ;;Continuar recursion

(defun RES-SAT-p (cnf)
  (RES-SAT-p-aux cnf (get-literals cnf nil)))


;;
;;  EJEMPLOS:
;;
;;
;; SAT Examples
;;
(RES-SAT-p nil)  ;;; T
(RES-SAT-p '((p) ((~ q)))) ;;; T
(RES-SAT-p
 '((a b d) ((~ p) q) ((~ c) a b) ((~ b) (~ p) d) (c d (~ a)))) ;;; T
(RES-SAT-p
 '(((~ p) (~ q) (~ r)) (q r) ((~ q) p) ((~ q)) ((~ p) (~ q) r))) ;;;T
;;
;; UNSAT Examples
;;
(RES-SAT-p '(nil))        ;;; NIL
(RES-SAT-p '((S) nil))    ;;; NIL
(RES-SAT-p '((p) ((~ p)))) ;;; NIL
(RES-SAT-p
 '(((~ p) (~ q) (~ r)) (q r) ((~ q) p) (p) (q) ((~ r)) ((~ p) (~ q) r))) ;;; NIL
```

Ejercicio 4.6:

- **PSEUDOCÓDIGO**

Entrada: dos FBF

Salida: T si la segunda FBF es consecuencia lógica de la primera FBF

Función:

Unimos la primera FBF transformada a FNC con la segunda FBF negada transformada a FNC

Resolvemos.

Si es SAT -> NIL

Si es UNSAT -> T

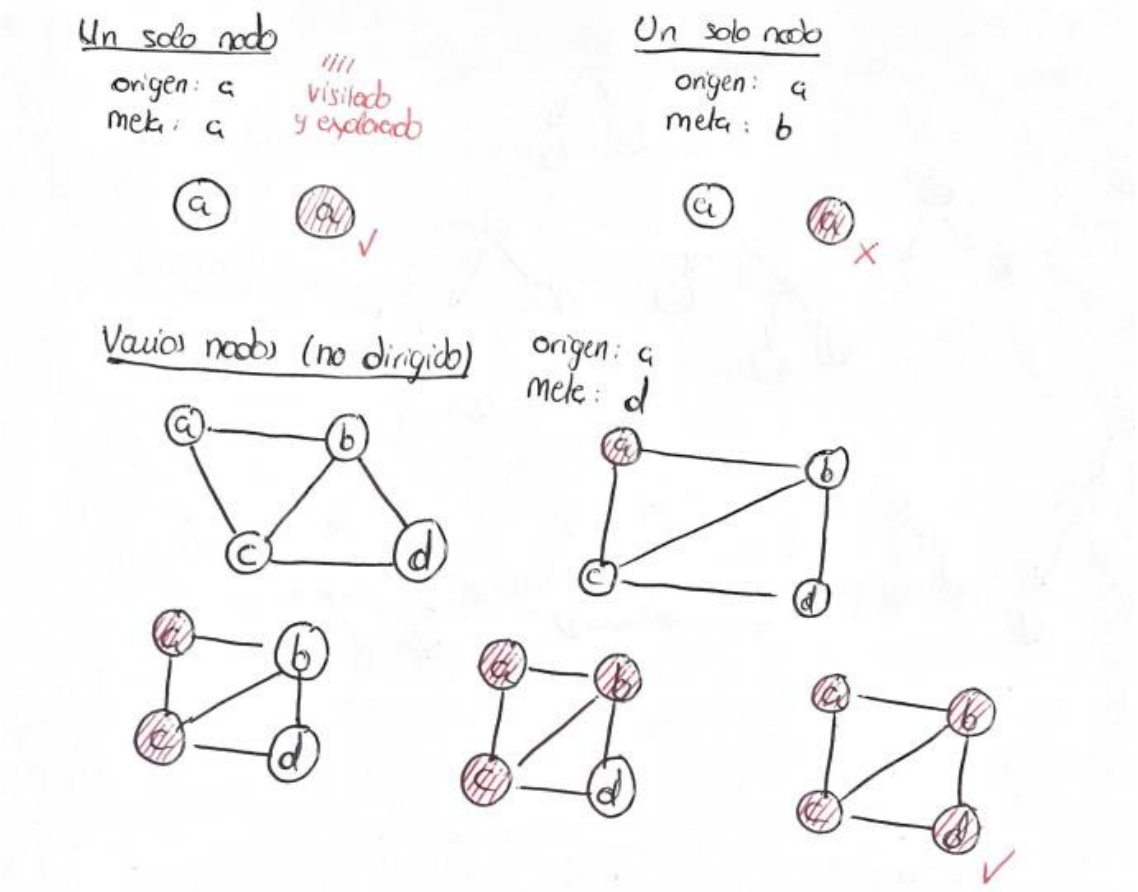
- **CÓDIGO**

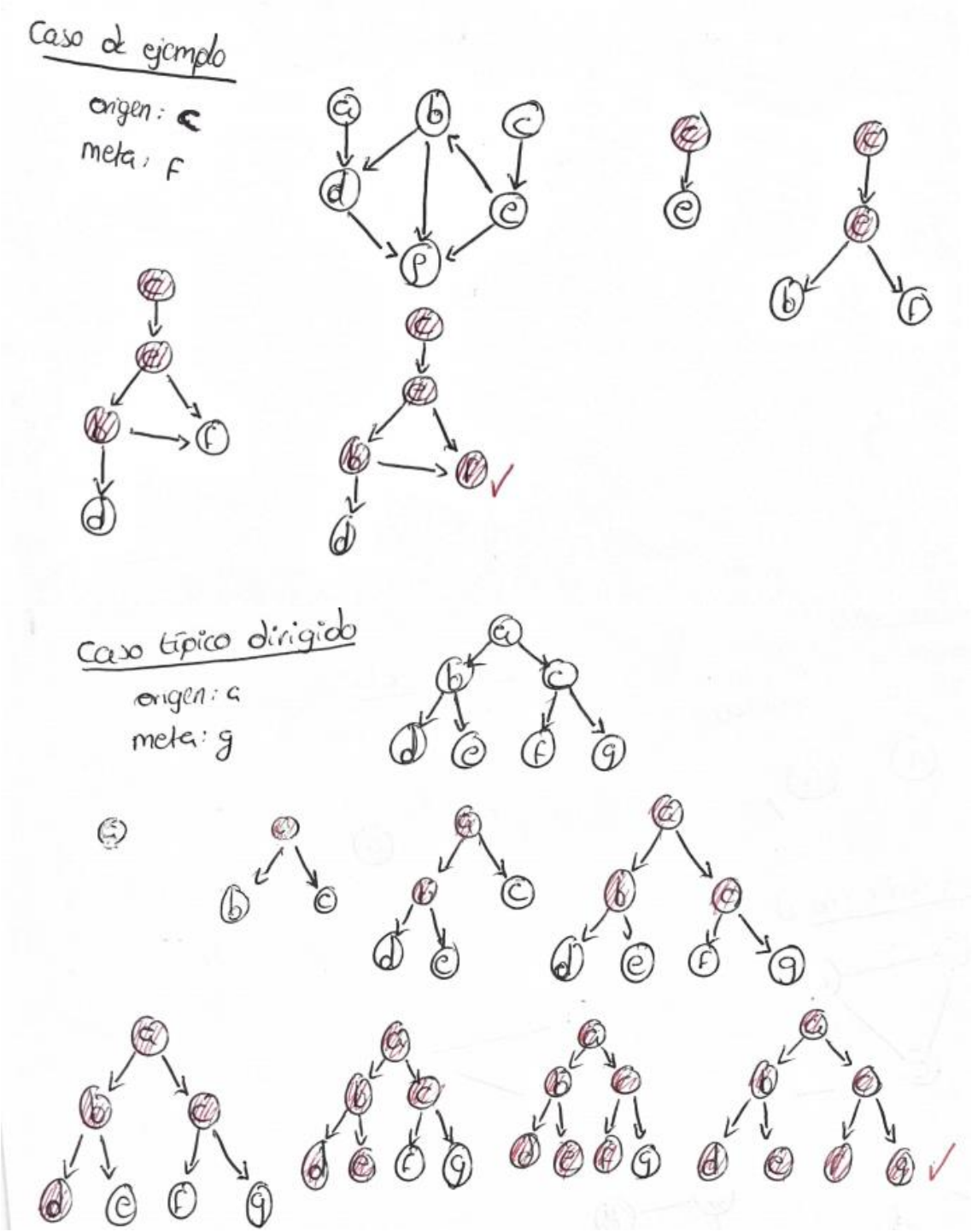
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.6:
;; Resolucion basada en RES-SAT-p
;;
;; RECIBE   : wff - FBF en formato infijo
;;          w   - FBF en formato infijo
;;
;; EVALUA A : T   si w es consecuencia logica de wff
;;          NIL en caso de que no sea consecuencia logica.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun logical-consequence-RES-SAT-p (wff w)
  (not (RES-SAT-p (union (wff-infix-to-cnf wff)
       (wff-infix-to-cnf (change-sign w))))))

;;
;;  EJEMPLOS:
;;
(logical-consequence-RES-SAT-p NIL 'a) ;;; NIL
(logical-consequence-RES-SAT-p NIL NIL) ;;; NIL
(logical-consequence-RES-SAT-p '(q ^ (~ q)) 'a) ;;; T
(logical-consequence-RES-SAT-p '(q ^ (~ q)) '(~ a)) ;;; T

(logical-consequence-RES-SAT-p '((p => (~ p)) ^ p) 'q)
;; T

(logical-consequence-RES-SAT-p '((p => (~ p)) ^ p) '(~ q))
;; T

(logical-consequence-RES-SAT-p '((p => q) ^ p) 'q)
;; T

(logical-consequence-RES-SAT-p '((p => q) ^ p) '(~q))
;; NIL

(logical-consequence-RES-SAT-p
 '(((~ p) => q) ^ (p => (a v (~ b))) ^ (p => ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
 '(~ a))
;; T

(logical-consequence-RES-SAT-p
 '(((~ p) => q) ^ (p => (a v (~ b))) ^ (p => ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
 'a)
;; T

(logical-consequence-RES-SAT-p
 '(((~ p) => q) ^ (p => ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
 'a)
;; NIL

(logical-consequence-RES-SAT-p
 '(((~ p) => q) ^ (p => ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
 '(~ a))
;; T

(logical-consequence-RES-SAT-p
 '(((~ p) => q) ^ (p <=> ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
 'q)
```

```
;; NIL

(logical-consequence-RES-SAT-p
 '(((~ p) => q) ^ (p <=> ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
 '(~ q))
;; NIL

(or
 (logical-consequence-RES-SAT-p '((p => q) ^ p) '(~q))      ;; NIL
 (logical-consequence-RES-SAT-p
  '(((~ p) => q) ^ (p => ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
  'a) ;; NIL
 (logical-consequence-RES-SAT-p
  '(((~ p) => q) ^ (p <=> ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
  'q) ;; NIL
 (logical-consequence-RES-SAT-p
  '(((~ p) => q) ^ (p <=> ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
  '(~ q)))
```

EJERCICIO 5

Ejercicio 5.2:





Ejercicio 5.2:

- **PSEUDOCÓDIGO**

Entrada: el nodo meta, una cola de caminos, y el grafo

Salida: el camino encontrado hasta la meta, o NULL si no lo encuentra

Función:

```
function bfs (meta, cola, grafo):
        if(cola == null):
```

```
            return null ;;No se ha encontrado la meta

        else:
            camino_actual = first(cola) ;; Camino al nodo actual
            nodo = first(camino_actual) ;; Nodo actual

            if(nodo == meta):
                return invertir(camino_actual) ;; Invertimos el camino, ya que lo hemos creado al
                                               ;; reves

            else:
                for hijo in nodo.hijos():
                    cola.remove()
                ;; Para cada hijo, sacamos el camino recorrido hasta llegar a el y lo introducimos en la
                cola de caminos.

                    cola.append(camino_a_(hijo))
                ;;Es importante que el hijo se inserte al principio del camino,
                ;;para que sea el nodo que se expanda en la siguiente iteracion.
                ;;Por tanto, el camino se ira guardando al reves.
                    return bfs(meta, cola, grafo)
```

Ejercicios 5.3 y 5.4:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 5.3 y 5.4
;; Estudia y comenta la siguiente implementación del
;; del algoritmo BFS
;;
;; RECIBE   : END (objetivo), QUEUE (cola de nodos a explorar)
;;          NET (grafo a explorar)
;; EVALUA A : Una lista con el camino encontrado hasta el objetivo
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun bfs (end queue net)
    (if (null queue) '() ;; Si la cola esta vacia, se ha terminado
        (let* ((path (first queue))
            (node (first path)))
        (if (eql node end) ;; Test objetivo
            (reverse path) ;; Le da la vuelta al path para que
            (bfs end       ;; salga en orden el camino
            ;; Al hacer el append con el rest de la cola, es como si hiciesemos un remove()
            (append (rest queue)
                (new-paths path node net)) ;; Explora el nodo, y mete los caminos
                                           ;;descubertos al final de la cola
                net)))))

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

```
;;
;; Función que descubre los hijos de un nodo, y devuelve una lista con
;; los nodos que se han descubierto hasta llegar al hijo. Estos caminos estan invertidos,
;; ya que en cada llamada se inserta el hijo en ultimo lugar
;;
;; REBIBE : PATH (lista de adyacencia), NODE (nodo a explorar), NET (grafo)
;; EVALUA A: Una lista con un camino por cada hijo de node
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun new-paths (path node net)
                        (mapcar #'(lambda(n) ;; La funcion lambda inserta los hijos de node
                                        (cons n path)) ;; en los caminos
                        (rest (assoc node net))))
```

Ejercicios 5.5 y 5.6:

Para que funcione correctamente la función *bfs*, es necesario que la entrada *queue* sea una lista de listas. Esto es porque la cola no va a ser una lista de nodos, sino una lista de caminos hasta el nodo.
BFS va a extraer el primer camino (la primera *sublista*) y, de ese camino, extraerá el primer nodo (que es el nodo que querremos expandir). A su vez, la función *new-paths* es la encargada de "descubrir" los hijos de un nodo. Para ello, va anexando cada hijo al camino que recibe como argumento, y terminará devolviendo una lista de caminos hasta cada nuevo hijo.
BFS hará un *append* de la lista con los nuevos caminos a *queue*, que es la cola de la búsqueda en anchura.
Como BFS extrae elementos del principio de la cola, y los que se añaden lo hacen al final, los primeros nodos en ser descubiertos serán los que estén en niveles superiores y, por tanto, la primera solución que se encuentre será la que está más arriba (la más corta).

A continuación se muestra una captura de pantalla de la traza de ejecución de las funciones SHORTEST-PATH, BFS, NEW-PATHS, para la siguiente llamada:

(shortest-path 'a 'f '((a d) (b d f) (c e) (d f) (e b f) (f)))

```
0[5]: (SHORTEST-PATH A F
                     ((A D)
                      (B D F)
                      (C E)
                      (D F)
                      (E B F)
                      (F)))
  1[5]: (BFS
         F
         ((A))                                  ;; Se llama a bfs con una lista de listas del nodo raíz
         ((A D) (B D F) (C E) (D F) (E B F) (F)))
    2[5]: (NEW-PATHS
           (A)
           A                                    ;; Se llama a new-paths con el camino actual (A)
           ((A D)                               ;; Y con el nodo a explorar A
            (B D F)
            (C E)
            (D F)
            (E B F)
            (F)))
    2[5]: returned ((D A))                      ;; Se devuelve una lista con el camino (invertido)
    2[5]: (BFS                                  ;; hasta cada hijo de A ((D A)=
           F
           ((D A))                              ;; Se llama a BFS, con una cola de caminos hasta los
           ((A D)                               ;; nodos-hoja
            (B D F)
            (C E)
            (D F)
            (E B F)
            (F)))
      3[5]: (NEW-PATHS
             (D A)
             D                                  ;; Se expande D
             ((A D)
              (B D F)
              (C E)
              (D F)
              (E B F)
              (F)))
      3[5]: returned ((F D A))                  ;; Se devuelve el camino hasta el único hijo de D
      3[5]: (BFS
             F
             ((F D A))
             ((A D)
              (B D F)
              (C E)
              (D F)
              (E B F)
              (F)))
      3[5]: returned (A D F)                    ;; Como ya se ha encontrado la meta, se devuelve el camino
    2[5]: returned (A D F)                      ;; ordenado
  1[5]: returned (A D F)
0[5]: returned (A D F)
(A D F)
```

En la captura aparecen comentarios del funcionamiento de la llamada. Se puede apreciar que la salida es la esperada.

Ejercicio 5.7:

La llamada correcta para encontrar el camino más corto entre F y C para el código anterior sería:

(setf net '((a b c d e) (b a d e f) (c a d) (d a b g h) (e a b g h) (f b h) (g c d e h) (h d e f g)))
(shortest-path 'f 'c net)

El resultado obtenido en este caso es (f b a c)

Sin embargo, el resultado va a depender del orden en el que definas las aristas del grafo. Por ejemplo, en el siguiente caso, usaremos el mismo grafo pero declarando las aristas en otro orden:

(setf net '((a b c d e) (h d e f g) (g c d e h) (b a d e f) (c a d) (d a b g h) (e a b g h) (f h b)))
(shortest-path 'f 'c net)

En este caso, el resultado obtenido sigue estando en el mismo nivel (recorre 4 nodos), pero a través de otro camino: (f h g c)

Ejercicio 5.8:

La siguiente llamada ejemplifica el caso en el que el código anterior entra en una recursión infinita. Esto ocurre cuando hay lazos (bucles) en nuestro grafo y la solución no se encuentra dentro del mismo:

(setf infinite '((a b c) (b a c) (c a b)))
(shortest-path-improved 'a 'e infinite)

El siguiente código corrige el problema al comprobar que el nodo que queremos expandir no ha sido visitado previamente.

```
(defun bfs-improved (end queue net)
        (if (null queue) '() ;; Si la cola esta vacia, se ha terminado
                (let* ((path (first queue))
                        (node (first path)))
                (if (eql node end) ;; Test objetivo
                        (reverse path) ;; Le da la vuelta al path para que
                        (bfs-improved end       ;; salga en orden el camino
                                (if (member node (rest path)) ;; comprobamos que no se haya
                                                              ;; pasado ya por ese nodo
                                        (rest queue)
                                        (append (rest queue) ;; Al hacer el append con el rest
                                                             ;; de la cola, es como si
                                                             ;; hiciesemos un remove()
                                                (new-paths-improved path node net)))
                        ;; Explora el nodo, y mete los caminos descubertos al final de la cola
                                net)))))


(defun new-paths-improved (path node net)
                (mapcar #'(lambda(n) ;; La funcion lambda inserta los hijos de nodo
                        (cons n path)) ;; en los caminos
                        (rest (assoc node net))))
```

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