MEMORIA PRÁCTICA 1

INTELIGENCIA ARTIFICIAL

Nuria Cuaresma Saturio y Luis Cárabe Fernández-Pedraza

Ejercicio 1

1.1

PSEUDOCÓDIGO

Entrada: x (primer vector)
y (segundo vector)

Salida: r (similitud coseno)

Procesamiento:

Si los dos vectores son válidos
devolvemos la división con numerador: productoVectorial(x,y)
y denominador: raizCuadrada(ProductoVectorial(x,x))* raizCuadrada(ProductoVectorial(x,x))
En caso contrario, devolvemos null

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-rec (x y)
;;; Calcula la similitud coseno de un vector de forma recursiva
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;; La semejanza coseno entre dos vectores que son listas vacÃas o que son
;;; (0 0...0) es NIL.
;;; 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 (and (is-ok x) (is-ok y) (not (or (null x) (null y))))
        (/ (sc-pvect-rec x y) (* (sqrt (sc-pvect-rec x x)) (sqrt (sc-pvect-rec y y))))
        nil))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-mapcar (x y)
;;; Calcula la similitud coseno de un vector usando mapcar
;;; Se asume que los dos vectores de entrada tienen la misma longitud.
;;; La semejanza coseno entre dos vectores que son listas vacÃas o que son
;;; (0 0...0) es NIL.
;;; 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 (and (is-ok x) (is-ok y) (not (or (null x) (null y))))
        (/ (sc-pvect-mapcar x y) (* (sqrt (sc-pvect-mapcar x x)) (sqrt (sc-pvect-mapcar y y))))
        nil))
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; is-ok (x)
;;; Comprueba si una lista no es de la forma (0 0...0)
;;; INPUT: x: vector, representado como una lista
;;; OUTPUT: t si no lo es, nil si lo es

(defun is-ok (x)
  (if (null (first x))
      nil
      (if (zerop (first x))
      (is-ok (rest x))
      t)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-pvect-rec (x y)
;;; Funcion auxiliar para sc-rec encargada de realizar el
;;; producto vectorial de manera recursiva
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;; OUTPUT: producto vectorial de x e y

(defun sc-pvect-rec (x y)
  (if (null (rest x))
  (* (first x) (first y))
  (+ (sc-pvect-rec (rest x) (rest y)) (* (first x) (first y)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-pvect-mapcar (x y)
;;; Función auxiliar para sc-mapcar encargada de realizar el
;;; producto vectorial
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;; OUTPUT: producto vectorial de x e y

(defun sc-pvect-mapcar (x y)
  (apply #'+ (mapcar #'* x y)))
```

EJEMPLOS con resultados obtenidos

```lisp
(setf l1 '(0 0 0 2))
(setf l2 '(0 0 0 0))
(setf l3 '())
(setf l4 '(2 3 4 5))

(sc-rec l1 l2) ;; NIL
```

(sc-mapcar l1 l2) ;; NIL
(sc-rec l1 l1) ;; 1.0
(sc-mapcar l1 l1) ;; 1.0
(sc-rec l1 l3) ;; NIL
(sc-mapcar l1 l3) ;; NIL
(sc-rec l1 l4) ;; 0.68041384
(sc-mapcar l1 l4) ;; 0.68041384
(sc-rec l4 l4) ;; 1.0
(sc-mapcar l4 l4) ;; 1.0

1.2

PSEUDOCÓDIGO

Entrada: cat (vector que representa una categoría)
vs (vector de vectores)
conf (nivel de confianz**a)**

Salida: r (lista ordenada según la similitud con cat)

Procesamiento:
Si los argumentos de entrada son válidos
Para cada elemento de vs: hacemos la similitud coseno de ese elemento con cat, si no cumple la confianza, lo eliminamos, si la cumple lo insertamos en orden

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf (cat vs conf)
;;; Devuelve aquellos vectores similares a una categoría
;;; INPUT: cat: vector que representa a una categoría, representado como una lista
;;; vs: vector de vectores
;;; conf: Nivel de confianza
;;; OUTPUT: Vectores cuya similitud con respecto a la categoría es superior al
;;; nivel de confianza, ordenado

(defun sc-conf (cat vs conf)
  (if (or (null cat) (null vs) (null conf))
      nil ;; Eliminamos los que no cumplen el grado de confianza
      (remove-if #'(lambda(x) (> conf (sc-rec x cat))) (sc-sort cat vs #'sc-rec))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-sort (cat vs fun)
;;; Ordena una lista según el grado de similitud
;;; INPUT: cat: vector que representa a una categoría, representado como una lista
;;; vs: vector de vectores
;;; fun: función con la que vamos a comparar
;;; OUTPUT: vectores ordenados segun la similitud con cat
```

```
(defun sc-sort (cat vs fun)
  (sort (copy-list vs) #'(lambda(x y) (> (funcall fun x cat) (funcall fun y cat)))))
```

EJEMPLOS con resultados obtenidos

```
(setf l5 '((56 0 678) (1 2 3) (6 7 8) (1000 0 123) (10 11 12)))
(setf categ '(1 2 3))

(sc-conf categ l5 0.9) ;; ((1 2 3) (6 7 8) (10 11 12))
(sc-conf categ l5 1)  ;;(( 1 2 3))
```

1.3

PSEUDOCÓDIGO

Entrada: cats (lista de categorías, es decir, vectores)
texts (lista de vectores)
func (función para evaluar la similitud coseno)

Salida: r (pares identificador de categoría con resultado de similitud coseno)

Procesamiento:
Para cada elemento de texts: buscamos el vector de categorías que más relación guarda con él según func y guardamos su id y dicho nivel de similitud

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-classifier (cats texts func)
;;; Clasifica a los textos en categorías.
;;;
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;; texts: vector de vectores, representado como una lista de listas
;;; func: función para evaluar la similitud coseno
;;; OUTPUT: Pares identificador de categoría con resultado de similitud coseno
;;;

(defun sc-classifier (cats texts func)
  (if (or (null cats) (null texts) (null func))
      nil
    (sc-classifier-rec cats texts func)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-special-sort (cats text func)
;;; Variante de sc-sort (maneja listas con un id como primer elemento)
```

```
;;; INPUT: cats: vector que representa distintas categorias, representado como una lista de listas
;;; text: lista con la que comparar
;;; func: funcion con la que vamos a comparar
;;; OUTPUT: vectores ordenados segun la similitud cats-text

(defun sc-special-sort (cats text func)
  (sort (copy-list cats) #'(lambda(x y)
                          (> (funcall func (rest x) (rest text)) (funcall func (rest y) (rest text))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-classifier-aux (cats text func)
;;; Funcion auxiliar para sc-classifier que compara un vector con
;;; todo el array de categorias y devuelve el id de la categoria
;;; mas afin junto con esa afinidad
;;; INPUT: cats: vector que representa distintas categorias, representado como una lista de listas
;;; text: lista con la que comparar
;;; func: funcion con la que vamos a comparar
;;; OUTPUT: lista de la forma (id distancia)

(defun sc-classifier-aux (cats text func)
  (let ((primer (first (sc-special-sort cats text func))))
    (list (first primer) (funcall func (rest text) (rest primer)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-classifier-rec (cats texts func)
;; Clasifica a los textos en categorías de manera recursiva
;;;
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;; texts: vector de vectores, representado como una lista de listas
;;; func: función para evaluar la similitud coseno
;;; OUTPUT: Pares identificador de categoría con resultado de similitud coseno

(defun sc-classifier-rec (cats texts func)
  (if (null (first texts))
      nil
    (cons (sc-classifier-aux cats (first texts) func) (sc-classifier-rec cats (rest texts) func))))
```

EJEMPLOS con resultados obtenidos

Forman parte del ejercicio 1.4.

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 lista1 (make-list 500 :initial-element '1))
(setf lista2 (make-list 500 :initial-element '2))
(setf prueba (list lista1 lista2))

(time (sc-classifier prueba prueba #'sc-rec))
(time (sc-classifier prueba prueba #'sc-mapcar))
```

El resultado de estas dos últimas ejecuciones es la siguiente:

```
CG-USER(52):
; cpu time (non-gc) 0.040000 sec user, 0.000000 sec system
; cpu time (gc)     0.076000 sec user, 0.000000 sec system
; cpu time (total)  0.116000 sec user, 0.000000 sec system
; real time  0.116951 sec (99.19%)
; space allocation:
;  351,206 cons cells, 8,362,112 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 25), minor: 133 (gc: 25)
((1 0.99999994) (1 0.99999994))
CG-USER(53):
; cpu time (non-gc) 0.004000 sec user, 0.000000 sec system
; cpu time (gc)     0.000000 sec user, 0.000000 sec system
; cpu time (total)  0.004000 sec user, 0.000000 sec system
; real time  0.003997 sec (100.1%)
; space allocation:
;  10,682 cons cells, 41,792 other bytes, 0 static bytes
; Page Faults: major: 0 (gc: 0), minor: 0 (gc: 0)
((1 0.99999994) (1 0.99999994))
```

Podemos observar como si usamos listas largas (de 500 elementos) la opción recursiva tarda más con respecto a la implementación con mapcar, 0.117 segundos contra 0.004.

Ejercicio 2

2.1

PSEUDOCÓDIGO

Entrada: f (función sobre la que tratar)
a (extremo menor del intervalo)
b (extremo mayor del intervalo)
tol (tolerancia)

Salida: r (raíz de la función en ese intervalo)

Procesamiento:

```
Si el (valor de la función en a * valor de la función en b) >= 0
        no hay solución, valor para una raíz
en caso contrario
        Si el (intervalo es menor que la tolerancia) o
                (el valor de la función en el punto medio = 0)
                devolvemos como raíz ese punto medio
        Si el  (valor de la función en a * valor de la función en el punto medio) < 0
                Llamamos de manera recursiva a la función con el intervalo (a, punto medio(a,b))
        En caso contrario
                Llamamos de manera recursiva a la función con el intervalo (punto medio(a,b), b)
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; bisect (f a b tol)
;;; Función para calcular una raíz aproximada de una función f
;;; INPUT: a: Extremo menor del intervalo
;;; b: Extremo mayor del intervalo
;;; tol: tolerancia para condición de parada
;;; f : función de la que buscamos raíces
;;; OUTPUT: raíz de f


(defun bisect (f a b tol)
  (if (>= (* (funcall f a) (funcall f b)) 0) ;;caso en el que no se encuentre raiz
        nil
        ;;intervalo menor que la tolerancia o caso en el que la raiz sea la media
        (if (or (< (- b a) tol) (= (funcall f (/ (+ a b) 2)) 0))
        (/ (+ a b) 2)
        (if(< (* (funcall f a) (funcall f (/ (+ a b) 2))) 0);;raiz entre a y la media
        (bisect f a (/ (+ a b) 2) tol)
        (bisect f (/ (+ a b) 2) b tol)))));;raiz entre b y la media
```

EJEMPLOS con resultados obtenidos

(bisect #'(lambda(x) (sin (* 6.26 x))) 0.1 0.7 0.001) ;;---> 0.5016602
(bisect #'(lambda(x) (sin (* 6.26 x))) 0.0 0.7 0.001) ;;---> NIL
(bisect #'(lambda(x) (sin (* 6.28 x))) 1.1 1.5 0.001) ;;---> NIL
(bisect #'(lambda(x) (sin (* 6.28 x))) 1.1 2.1 0.001) ;;---> NIL

2.2

PSEUDOCÓDIGO

Entrada: f (función sobre la que tratar)
lst (lista de valores entre los que buscar raices)
tol (tolerancia)

Salida: lst (raíces de la función en ese intervalo)

Procesamiento:

Buscamos raices entre los dos primero elementos de la lista
Si encontramos:
Los insertamos en una lista
Llamamos de manera recursiva con el resto de la lista
En caso contrario:
Llamamos de manera recursiva con el resto de la lista

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; allroot (f lst tol)
;;; Funcion para calcular una raices aproximadas de f
;;; INPUT: lst: lista de valores entre los que buscamos raices
;;; tol: tolerancia para condición de parada
;;; f : función de la que buscamos raices
;;; OUTPUT: lista con todas las raíces encontradas de f

(defun allroot (f lst tol)
  (if (not (null (rest lst)))
	(if (not (null (bisect f (first lst) (second lst) tol))) ;; calculamos las raices entre el primer y segundo puntos
	;; lo insertamos y llamamos recursibamente con el resto de la lista
	(append (list (bisect f (first lst) (second lst) tol))(allroot f (rest lst) tol))
	(allroot f (rest lst) tol))));; en caso de no encontrar en ese intervalo una raiz, no insertamos en la lista
```

EJEMPLOS con resultados obtenidos

(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))) '(0.25 0.9 0.75 1.25 1.75 2.25) 0.0001) ;; --> (0.5002166 1.0005188 1.5007629 2.001007)

2.3

PSEUDOCÓDIGO

Entrada: a: Extremo menor del intervalo
b: Extremo mayor del intervalo
tol: tolerancia para condición de parada
f : función de la que buscamos raíces
N: exponente para obtener el número de secciones en la que dividir el intervalo

Salida: r (raíz de la función en ese intervalo)

Procesamiento:
Calculamos la potencia de 2
Calculamos una lista con los elementos pedidos con 2^n elementos
Obtenemos las raices en ese intervalo de una funcion dada

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; power (N)
;;; Funcion para calcular potencias de 2
;;; INPUT: N: exponente al que quiero elevar 2
;;; OUTPUT: potencia de 2

(defun power (N)
  (if (= N 0) ;; caso de 2 elevado a 0
      1
      (* 2 (power (- N 1)))));; multiplicamos N veces 2

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lst (a iter p div)
;;; Función para obtener la lista de valores entre los que
;;; queremos calcular las raices de la funcion
;;; INPUT: a: Extremo menor del intervalo
;;; iter: Extremo mayor del intervalo
;;; max: numero maximo
;;; div: numero de elementos
;;; OUTPUT: raiz de f

(defun lst(a iter max div)
  (if (/= iter max)
      (cons (+ a (* iter div))(lst a (+ 1 iter) max div))
      (list (+ a (* iter div)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; allind (f a b N tol)
;;; Función que divide un intervalo en 2^(N) secciones y
;;; busca en cada sección una raíz de la función f
```

```
;;; INPUT: a: Extremo menor del intervalo
;;; b: Extremo mayor del intervalo
;;; tol: tolerancia para condición de parada
;;; f : función de la que buscamos raices
;;; N: exponente para obtener el número de secciones en las
;;; que dividir el intervalo
;;; OUTPUT: Lista con las raices encontradas

(defun allind (f a b N tol)
  (allroot f (lst a 0 (power N) (/(- b a) (power N))) tol))
```

EJEMPLOS con resultados obtenidos

```
 (allind #'(lambda(x) (sin (* 6.28 x))) 0.1 2.25 2 0.0001) ;; --> (0.50027096 1.000503 1.5007349
2.0010324)
 (allind #'(lambda(x) (sin (* 6.28 x))) 0.1 2.25 1 0.0001) ;; --> NIL
```

Ejercicio 3

3.1

PSEUDOCÓDIGO

Entrada: elt (elemento)
lst (vector)

Salida: r (elt combinado con todos los elementos de lst)

Procesamiento:

Para cada elemento de lst: creamos lista con el elemento y elt

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst (elt lst)
;;; Funcion que combina un elemento dado con todos los elementos de una
;;; lista
;;; INPUT: elt elemento
;;;        lst vector, representado como una lista
;;; OUTPUT: lista con el elemento combinado

(defun combine-elt-lst (elt lst)
  (if (or (null lst))
      nil
      (mapcar #'(lambda (x) (list elt x)) lst)))
```

EJEMPLOS con resultados obtenidos

```
(combine-elt-lst 'a nil) ;; --> NIL
(combine-elt-lst 'a '(1 2 3)) ;; --> ((A 1) (A 2) (A 3))
```

3.2

PSEUDOCÓDIGO

Entrada: lst1 (primera lista)
lst2 (segunda lista)

Salida: r (producto cartesiano de las dos listas)

Procesamiento:

Para cada elemento en lst1: combinamos dicho elemento con lst2
Devolvemos la lista de combinaciones

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst-rec (lst1 lst2)
;;; Funcion recursiva que realiza el producto cartesiano entre dos listas
```

```
;;; INPUT: lst1 primera lista
;;;        lst2 segunda lista
;;; OUTPUT: lista con el producto cartesiano

(defun combine-lst-lst-rec (lst1 lst2)
  (if (null (first lst1))
        nil
        (append (combine-elt-lst (first lst1) lst2) (combine-lst-lst-rec (rest lst1) lst2))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst (lst1 lst2)
;;; Funcion que realiza el producto cartesiano entre dos listas
;;; (llamando a combine-lst-lst-rec)
;;; INPUT: lst1 primera lista
;;;        lst2 segunda lista
;;; OUTPUT: lista con el producto cartesiano

(defun combine-lst-lst (lst1 lst2)
  ;; Comprobamos que no sean nil y llamamos a la funcion recursiva
  (if (or (null lst1) (null lst2))
        nil
        (combine-lst-lst-rec lst1 lst2)))
```

EJEMPLOS con resultados obtenidos

```
(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))
```

3.3

PSEUDOCÓDIGO

Entrada: lstolsts (lista de listas)

Salida: (lista de todas las disposiciones)

Procesamiento:

Para cada par de elementos de lstolsts: producto cartesiano

Creamos lista con ese producto y el resto de lstolsts y volvemos a llamar a la función.

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts (lstolsts)
;;; Funcion que calcula todas las posibles disposiciones de elementos
;;; pertenecientes a N listas de forma que en cada disposicion aparezca
;;; unicamente un elemento de cada lista. (Llama a combine-list-of-lsts-rec)
;;; INPUT: lstolsts vector de listas
;;; OUTPUT: lista con todas las disposiciones
```

```lisp
(defun combine-list-of-lsts (lstolsts)
  (if (null lstolsts)
	(list nil)
	(if (some #'null lstolsts)
	nil
	(if (null (rest lstolsts)) ;; Cubrimos el caso en el cual solo hay una lista
	(mapcar #'(lambda (x) (list x)) (first lstolsts))
	(combine-list-of-lsts-rec lstolsts))))) ;; Caso general

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; flatten (lst)
;;; Funcion dada en las transparencias para pasar de una estructura
;;; de arbol a una de lista (quitando parentesis)
;;; INPUT: lst lista
;;; OUTPUT: lista con todos los elementos al mismo nivel

(defun flatten (lst)
  (cond
   ((null lst) NIL)
   ((atom (first lst))
	(cons
	(first lst)
	(flatten (rest lst))))
   (t (append
	(flatten (first lst))
	(flatten (rest lst))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst-spe (elt lst)
;;; Funcion que combina un elemento dado con todos los elementos de una
;;; lista, el elemento resulta ser una lista, es decir, realiza el
;;; producto cartesiano de un elemento de la forma (A B) con una lista
;;; INPUT: elt elemento a combinar (es una lista)
;;;	lst vector, representado como una lista
;;; OUTPUT: lista con el elemento combinado

(defun combine-lst-lst-spe (elt lst)
  (if (or (null lst))
	nil
	(mapcar #'(lambda (x) (flatten (list elt x))) lst)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts-aux (lists l1)
;;; Funcion que realiza el producto cartesiano de la lista l1 con
```

```
;;; cada una de las sublistas del vector lists
;;; INPUT: lists vector de listas a las cuales se quiere hacer el
;;;                     producto cartesiano
;;;         l1 lista con la que se hace el producto cartesiano
;;; OUTPUT: lista con los productos cartesianos

(defun combine-list-of-lsts-aux (lists l1)
  (if (null (rest lists))
          (combine-lst-lst-spe (first lists) l1)
          (append (combine-lst-lst-spe (first lists) l1) (combine-list-of-lsts-aux (rest lists) l1))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts-rec (lstolsts)
;;; Funcion recursiva que calcula todas las posibles disposiciones de elementos
;;; pertenecientes a N listas de forma que en cada disposicion aparezca
;;; unicamente un elemento de cada lista.
;;; INPUT: lstolsts vector de listas
;;; OUTPUT: lista con todas las disposiciones

(defun combine-list-of-lsts-rec (lstolsts)
  (if (null (rest (rest lstolsts)))
          (combine-list-of-lsts-aux (first lstolsts) (first (rest lstolsts)))
          (combine-list-of-lsts-rec
          (cons (combine-list-of-lsts-aux (first lstolsts) (first (rest lstolsts))) (rest (rest lstolsts))))))
```

EJEMPLOS con resultados obtenidos

```
(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 '((1 2 3 4) (a b c)))
          ;; ((1 A) (1 B) (1 C) (2 A) (2 B) (2 C) (3 A) (3 B) (3 C) (4 A) ...)
(combine-list-of-lsts '((a b c) (+ -) (1 2 3 4)))
          ;;((A + 1) (A + 2) (A + 3) (A + 4) (A - 1) (A - 2) (A - 3) (A - 4) (B + 1) (B + 2) ...)
(combine-list-of-lsts '((a b c) (+ -) (1 2) (o p)))
          ;;((A + 1 O) (A + 1 P) (A + 2 O) (A + 2 P) (A - 1 O) (A - 1 P) (A - 2 O) (A - 2 P)
 (B + 1 O) (B + 1 P) ...)
```

Ejercicio 4

4.1.1

PSEUDOCÓDIGO

Entrada: x (expresion)

Salida: true (si atomo positivo)
nil (si no atomo positivo)

Procesamiento:

Si (no se trata de un valor de verdad, true o nil) y
si (no es un conector) y si (es un atomo):
Es un literal positivo
en caso contrario:
No es un literal positivo

CÓDIGO

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

EJEMPLOS con resultados obtenidos

```
(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
```

<u>4.1.2</u>

PSEUDOCÓDIGO

Entrada: x (expresion)

Salida: true (si negativo)
nil (no negativo)

Procesamiento:

Si (se trata de un objeto) y si (no hay mas de dos elementos)

si (el primer elemento es un not) y si (al quitar el not es un literal positivo):
Es un literal negativo
en caso contrario:
No es un literal negativo

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) (null (cddr x)) (eql (car x) +not+) (positive-literal-p (car (cdr x)))))
```

EJEMPLOS con resultados obtenidos

```
(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
```

<u>4.1.3</u>

PSEUDOCÓDIGO

Entrada: x (expresion)
Salida: true (si expresion literal)
nil (no expresion literal)

Procesamiento:
Si x es literal positivo o negativo: devolvemos true
Nil en caso contrario.

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.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; comprobamos si se trata d eun literal
(defun literal-p (x)
  (or (positive-literal-p x)
          (negative-literal-p x)))
```

EJEMPLOS con resultados obtenidos

```
(literal-p 'p)
(literal-p '(~ p))
;;; evaluan a T
(literal-p '(p))
(literal-p '(~ (v p q)))
;;; evaluan a  NIL
```

4.1.4

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Predicado para determinar si una expresion esta en formato prefijo
;;
;; RECIBE   : expresion x
;; EVALUA A : T si x esta en formato prefijo, NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun wff-prefix-p (x)
  (unless (null x)            ;; NIL no es FBF en formato prefijo (por convencion)
          (or (literal-p x)              ;; Un literal es FBF en formato prefijo
          (and (listp x)      ;; En caso de que no sea un literal debe ser una lista
          (let ((connector (first x))
                    (rest_1  (rest  x)))
          (cond
          ((unary-connector-p connector)  ;; Si el primer elemento es un connector unario
          (and (null (rest rest_1))  ;; deberia tener la estructura (<conector> FBF)
                    (wff-prefix-p (first rest_1))))
          ((binary-connector-p connector) ;; Si el primer elemento es un conector binario
          (let ((rest_2 (rest rest_1)))  ;; deberia tener la estructura
                    (and (null (rest rest_2))  ;; (<conector> FBF1 FBF2)
                    (wff-prefix-p (first rest_1))
                    (wff-prefix-p (first rest_2)))))
          ((n-ary-connector-p connector)  ;; Si el primer elemento es un conector enario
          (or (null rest_1)             ;; conjuncion o disyuncion vacias
                    (and (wff-prefix-p (first rest_1)) ;; tienen que ser FBF los operandos
                    (let ((rest_2 (rest rest_1)))
                    (or (null rest_2)             ;; conjuncion o disyuncion con un elemento
```

```
                    (wff-prefix-p (cons connector rest_2)))))))
        (t NIL)))))))          ;; No es FBF en formato prefijo
;;
```

EJEMPLOS con resultados obtenidos

```
(wff-prefix-p '(v))
(wff-prefix-p '(^))
(wff-prefix-p '(v A))
(wff-prefix-p '(^ (~ B)))
(wff-prefix-p '(v A (~ B)))
(wff-prefix-p '(v (~ B) A ))
(wff-prefix-p '(^ (V P (=> A (^ B (~ C) D))) (^ (<=> P (~ Q)) P) E))
;;; evaluan a T
(wff-prefix-p 'NIL)
(wff-prefix-p '(~))
(wff-prefix-p '(=>))
(wff-prefix-p '(<=>))
(wff-prefix-p '(^ (V P (=> A ( B ^ (~ C) ^ D))) (^ (<=> P (~ Q)) P) E))
;;; evaluan a NIL
```

PSEUDOCÓDIGO

Entrada: x (expresion)
Salida: true (si x usa formato infijol)
nil (no usa formato infijo)

Procesamiento:

Si es null no es fbf en formato prefijo, salimos
Si es un literal ya está en formato prefijo
Si es lista separamos el conector y el resto de la lista
Si el conector es unario tendra la forma (<conector> FBF)
Si el conector es binario tendra la forma (FBF1<conector> FBF2)
Si el conector es n-ario tendra la forma (FBF <v, ^> FBF <v, ^> FBF ...)

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
        (let ((op1 (car x))
                (ex1 (cadr x))
                (lex2 (cddr x)))
        (cond
        ((and (null ex1) (n-ary-connector-p op1)) t) ;; Por convencion
        ((unary-connector-p op1)  ;; Si el primer elemento es un conector unario
        (and (null lex2)            ;; deberia tener la estructura (<conector> FBF)
                (wff-infix-p ex1)))
        ((binary-connector-p ex1) ;; Si el segundo elemento es un conector binario
        (and (wff-infix-p op1)   ;; deberia tener la estructura (FBF <conector> FBF)
                (null (cdr lex2))
                (wff-infix-p (car lex2))))
        ((n-ary-connector-p ex1) ;; Si el segundo elemento es un conector n-ario
        (and (wff-infix-p op1)  ;; el primer elemento deberia ser FBF
                (nop-verify ex1 (cdr x))))
        (t NIL))))))) ;; No es FBF en formato infijo


;;
;; 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
```

4.1.5

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Convierte FBF en formato prefijo a FBF en formato infijo
;;
;; RECIBE   : FBF en formato prefijo
;; EVALUA A : FBF en formato infijo
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun prefix-to-infix (wff)
  (when (wff-prefix-p wff)
        (if (literal-p wff)
        wff
        (let ((connector      (first wff))
        (elements-wff (rest wff)))
        (cond
        ((unary-connector-p connector)
        (list connector (prefix-to-infix (second wff))))
        ((binary-connector-p connector)
        (list (prefix-to-infix (second wff))
        connector
        (prefix-to-infix (third wff))))
        ((n-ary-connector-p connector)
        (cond
        ((null elements-wff)      ;;; conjuncion o disyuncion vacias.
        wff)                ;;; por convencion, se acepta como fbf en formato infijo
        ((null (cdr elements-wff))  ;;; conjuncion o disyuncion con un unico elemento
        (prefix-to-infix (car elements-wff)))
        (t (cons (prefix-to-infix (first elements-wff))
              (mapcan #'(lambda(x) (list connector (prefix-to-infix x)))
              (rest elements-wff))))))
        (t NIL)))))) ;; no deberia llegar a este paso nunca
```

PSEUDOCÓDIGO

Entrada: wff (FBF en formato infijo)
Salida: (expresion en formato prefijo)

Procesamiento:
Si wff no esta en formato infijo: salimos
- Si es literal o wff = (^/v): salimos
- Si el primer elemento es operador unario: lo ponemos delante y evaluamos el resto
- Cogemos el operador:
  - Si es binario: lo ponemos delante, detrás los dos operandos evaluados
  - Si es n-ario: lo ponemos delante, detrás todos los operandos evaluados

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) ;; Si es un literal, ya esta de la forma prefijo
        wff
      (let ((connector (second wff))
            (1-element (first wff)))
        (cond
         ((n-ary-connector-p 1-element) ;; Cubrimos el caso de conjuncion o disyuncion vacias,
          wff)                          ;; por convencion se acepta como fbf en formato prefijo
         ((unary-connector-p 1-element) ;; Si el operador es un-ario, siempre va delante del elemento
          (list 1-element (infix-to-prefix (cadr wff))))
         ((binary-connector-p connector)
          (list connector
                (infix-to-prefix 1-element)
                (infix-to-prefix (third wff))))
         ((n-ary-connector-p connector)
          (cons connector
                    (mapcan #'(lambda(x) (list (infix-to-prefix x))) ;; Pasamos a prefijo todas las expresiones
                      (remove-if #'n-ary-connector-p wff )))) ;; Eliminamos los operadores de la lista
         (t NIL)))))) ;; no deberia llegar a este paso nunca cddr wff
```

EJEMPLOS con resultados obtenidos

```
(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

(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))
```

<u>4.1.6</u>

PSEUDOCÓDIGO

Entrada: wff (FBF)
Salida: t (si wff es cláusula)
nil (si no)

Procesamiento:
Si es una lista, su primer elemento es un or, y sus operandos son todos literales: t

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)
```

```
  (and (listp wff) (eql (first wff) +or+) (every #'literal-p (rest wff))))
```

EJEMPLOS con resultados obtenidos

```
(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
```

4.1.7

PSEUDOCÓDIGO

Entrada: wff (FBF)
Salida: t (si wff está en formato FNC)
nil (si no)

Procesamiento:
Si es una lista, su primer elemento es un and, y sus operandos son todos cláusulas: t

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)
  (and (listp wff) (eql (first wff) +and+) (every #'clause-p (rest wff))))
```

EJEMPLOS con resultados obtenidos

```
(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
```

<u>4.2.1</u>

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)) ;; Caso base: nil o literal, en ese caso,
      wff                             ;; devolvemos la fbf tal cual
    (let ((connector (first wff)))
      (if (eq connector +bicond+) ;; Comprobamos si el operador es el conector bicondicional
          (let ((wff1 (eliminate-biconditional (second wff)))   ;; Teniendo la expresion (<=> exp1 exp2),
eliminamos
                (wff2 (eliminate-biconditional (third  wff))))  ;; posibles conectores bicondicionales de exp1
y exp2
            (list +and+
                  (list +cond+ wff1 wff2)   ;; Creamos la lista de la forma (^ (=> exp1 exp2) (=> exp2 exp1))
                  (list +cond+ wff2 wff1)))
        (cons connector ;; Si el operador no es el conector bicondicional, analizamos el resto de la 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))))
```

4.2.2

PSEUDOCÓDIGO

Entrada: wff (expresion fbf)
Salida: (expresion fbf sin conector =>)

Procesamiento:
Si wff es null o literal: lo devolvemos tal cual
Si el operador es =>: pasamos la expresion de (=> exp1 exp2) a (v (~ exp1) exp2)
Si no, evaluamos el resto de la fbf

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)) ;; Caso base: nil o literal, en ese caso,
      wff                             ;; devolvemos la fbf tal cual
    (let ((connector (first wff)))
      (if (eq connector +cond+) ;; Comprobamos si el operador es el conector condicional
          (let ((wff1 (eliminate-conditional (second wff)))   ;; Teniendo la expresion (=> exp1 exp2), eliminamos
                (wff2 (eliminate-conditional (third  wff))))  ;; posibles conectores condicionales de exp1 y exp2
        (list +or+ ;; Creamos la lista de la forma (v (~ exp1) exp2)
              (list +not+ wff1)
              wff2))
```

```
    (cons connector ;; Si el operador no es el conector condicional, analizamos el resto de la 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)))
```

4.2.3

PSEUDOCÓDIGO

Entrada: wff (expresion fbf)
Salida: (expresion fbf con la negación unicamente en literales negativos)

Procesamiento:
Si wff es null o literal: lo devolvemos tal cual
Si el operador es “not”:
  Si hay doble negacion: (~ (~ exp1)) -> (exp1)
  Si hay operador and u or: aplicamos De Morgan
  Si no, evaluamos el resto de la fbf
Si no, evaluamos resto de la fbf

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)) ;; Caso base: nil o literal, en ese caso,
      wff                             ;; devolvemos la fbf tal cual
   (let ((connector (first wff)))
     (if (eq connector +not+) ;; Comprobamos si el operador es el not
        (if (eq (caadr wff) +not+) ;; Caso de doble negacion
            (reduce-scope-of-negation (cadr (cadr wff))) ;; (~ (~ exp1)) -> (exp1)
          (cons (exchange-and-or (caadr wff)) ;; Aplicamos la ley de De Morgan
                (mapcar #'(lambda(x) (reduce-scope-of-negation (list +not+ x)))
                  (rest (second wff)))))
       (cons connector ;; Si el operador no es not, analizamos el resto de la fbf
             (mapcar #'reduce-scope-of-negation (rest wff)))))))
```

```
(defun exchange-and-or (connector)
  (cond
   ((eq connector +and+) +or+)
   ((eq connector +or+) +and+)
   (t connector)))
```

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))
```

4.2.4

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Combina un elemento (elt) con los elementos de una lista
(defun combine-elt-lst (elt lst)
  (if (null lst)
      (list (list elt))
      (mapcar #'(lambda (x) (cons elt x)) lst)))

;;Saca los conectores de las listas
(defun exchange-NF (nf)
  (if (or (null nf) (literal-p nf)) ;; en caso de que sea null o un literal, devolvemos él mismo
      nf
      (let ((connector (first nf))) ;; obtenemos el conector de la forma normal
      (cons (exchange-and-or connector) ;; intercambiamos ambos conectores, and y or
      (mapcar #'(lambda (x)
                  (cons connector x))
      (exchange-NF-aux (rest nf))))))) ;; creamos listas con conector y los elementos de la lista combinados

;; Obtenemos todas las listas posibles de combinar un elemento de una lista dada
;; con todas las listas
(defun exchange-NF-aux (nf)
  (if (null nf)
```

```
        NIL ;; devolvemos null en caso de que la forma normal sea null
        (let ((lst (first nf))) ;; obtenemos al primer elemento de la forma normal
        (mapcan #'(lambda (x)
        (combine-elt-lst
                x
                (exchange-NF-aux (rest nf)))) ;;intercambiamos conectores
        (if (literal-p lst) (list lst) (rest lst))))));; obtenemos los elementos de una u otra lista

;;simplifica los conectores que son del mismo tipo
(defun simplify (connector lst-wffs)
  (if (literal-p lst-wffs)
        lst-wffs ;; si se trata de un literal devolvemos la lista
        (mapcan #'(lambda (x)
        (cond
        ((literal-p x) (list x));;si se trata de literal
        ((equal connector (first x));; si el conector es igual que el primer elemento de la lista
        (mapcan
                #'(lambda (y) (simplify connector (list y)))
                (rest x))) ;;simplificamos conectores de cada lista
        (t (list x))))  ;; devolvemos la lista
        lst-wffs)))

;;
(defun cnf (wff)
  (cond
   ((cnf-p wff) wff) ;; en caso de que sea un cnf positivo, la devolvemos
   ((literal-p wff);; si es un literal
        (list +and+ (list +or+ wff))) ;; insertamos en una lista un and con una lista de or y la wff
   ((let ((connector (first wff)));;En caso de que no
        (cond
        ((equal +and+ connector) ;; si el conector es un and
        (cons +and+ (simplify +and+ (mapcar #'cnf (rest wff))));; simplificamos el and
        ((equal +or+ connector) ;; caso de que sea un or
        (cnf (exchange-NF (cons +or+ (simplify +or+ (rest wff)))))))))) ;;simplificamos e intercambiamos el
conector


(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))
```

<u>4.2.5</u>

PSEUDOCÓDIGO

Entrada: x (expresion fbf)
Salida: x (expresion fnc)

Procesamiento:
Si no es null
eliminamos los primeros elementos de las clausulas, que son los conectores ya que es fbf

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 eliminaos)
;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-connectors (cnf)
   (if (null cnf)
            NIL
            (mapcar #'rest (rest cnf))) ;; eliminamos los conectores, que estan al principio al ser fbf
)

(eliminate-connectors '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))
```

PSEUDOCÓDIGO

Entrada: x (expresion FBF)
Salida: x (expreion fnc)

Procesamiento:
Si es un literal o null lo sevolvemos
En caso contrario:
pasamos a prefijo
eliminamos condicionales y bicondicionales
eliminamos resto de 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)
  (if (or (null wff) (literal-p wff));; si es literal o null
```

```
            wff
            (reduce-aux (reduce-cond (infix-to-prefix wff)))))

(defun reduce-cond (wff) ;; elimina condicionales y bicondicionales
  (eliminate-conditional (eliminate-biconditional wff)))

(defun reduce-aux (wff) ;; elimina conectores
  (eliminate-connectors (cnf (reduce-scope-of-negation wff))))
;;
```

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))
```

4.3.1

PSEUDOCÓDIGO

Entrada: : K - clausula
Salida: clausula equivalente sin literales repetidos

Procesamiento:
Si es un literal o null lo devolvemos
si se encuentra un elemento igual al primero volvemos a buscar eliminando el primero
En caso contrario mantenemos el primero y volvemos a buscar sin el primero

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)
  (cond ((null k)NIL)
            ((literal-p k) k);; si es un litera
            ((member (first k) (rest k) :test #'equal);; si se encuentra un elemento igual al primero
            (eliminate-repeated-literals (rest k))) ;; volvemos a buscar eliminando el primero
            (T (cons (first k)(eliminate-repeated-literals (rest k)))))) ;; si no mantenemos el primero y recursivo

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

;;;   (B (~ A) (~ C) C A)

4.3.2

PSEUDOCÓDIGO

Entrada: x expresion cnf
Salida: x expresion fnc

Procesamiento:
comparamos todas las clausulas con los literales eliminados
comparamos las clausulas de una cnf
si la primera es clausula unica la mantenemos y llamamos de nuevo con el resto de cnf
si no continuamos llamando con el resto, sin la primera

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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; comparamos todas las clausulas con los literales eliminados
(defun eliminate-repeated-clauses (cnf)
  (compare-clauses (mapcar #'eliminate-repeated-literals cnf)))

;; comparamos las clausulas de una cnf
(defun compare-clauses (cnf)
  (cond ((or (null cnf) (null(rest cnf))) cnf)
        ((uniq-clause (first cnf) (rest cnf));;comprobamos si la primera es unica
        (cons (first cnf) (compare-clauses (rest cnf))));; si es unica mantenemos la clausula
        (t (compare-clauses (rest cnf)))));;si no continuamos con el resto

;;devolvemos si una clausula es unica en una cnf
(defun uniq-clause (clause cnf)
   (if(every #'null (mapcar #'(lambda (x) (clauses-not-equal clause x)) cnf)) T
        NIL))

;; comprobamos si son o no iguales dos clausulas
(defun clauses-not-equal (clause1 clause2)
   (= (length (intersection clause1 clause2 :test 'equal)) (length clause1) (length clause2)))
;;
;; 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))
```

4.3.3

PSEUDOCÓDIGO

Entrada: k1, k2 (clausulas)
Salida: k1 si subsume a k2
nil si no

Procesamiento:
restamos k1-k2
si es null es que 2 contiene todos los elementos de k1, está subsumida
en caso contrario k2 no está subsumida

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.3
;; Predicado que determina si una clausula subsume otra
;;
;; RECIBE   : K1, K2 clausulas
;; EVALUA a : K1 si K1 subsume a K2
;;          NIL en caso contrario
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun subsume (K1 K2)
  (when (null (set-difference K1 K2 :test 'is-equal));; restamos la menor - mayor si no hay elementos esta subsumida
    (list K1)))

;; Evalua si dos literales son iguales
(defun is-equal (x y)
  (or (equal x y) (and (listp x) (listp y) (equal +not+ (first x)) (equal +not+ (first y)) (equal (second x) (second y)))))
```

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
```

4.3.4

PSEUDOCÓDIGO

Entrada: K clausula, cnf
Salida: cnf sin clausulas subsumidas

Procesamiento:
Commprobamos de principio a fin y de fin a principio de la cnf
si es null devolvemos
si la primera no esta subsumida en ninguna clausula de la cnf
la mantenemos y realizamos recursion con el resto
si no
hacemos recursion eliminando la primera

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.4
;; eliminacion de clausulas subsumidas en una FNC
;;
;; RECIBE   : K (clausula), cnf (FBF en FNC)
;; EVALUA A : FBF en FNC equivalente a cnf sin clausulas subsumidas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun eliminate-subsumed-clauses (cnf)
  (eliminate-clauses(     ;;eliminamos clausulas en un sentido
        nreverse(eliminate-clauses cnf))));;  y en el inverso

(defun eliminate-clauses (cnf)
  (cond ((or (null cnf) (null(rest cnf))) cnf)
        ((subsumed-clause (first cnf) (rest cnf))  ;; si la primera no esta subsumida
        (cons (first cnf) (eliminate-subsumed-clauses (rest cnf)))) ;;la mantenemos y recursion con el resto
        (t (eliminate-subsumed-clauses (rest cnf))))) ;; si no recursion eliminando la primera

;; comprobamos si esta subsumida en alguna clausula del cnf
(defun subsumed-clause (clause cnf)
  (when (every #'null (mapcar #'(lambda (x) (subsume x clause)) cnf)) T ))
```

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))
```

4.3.5

PSEUDOCÓDIGO

Entrada: K (clausula)
Salida: T si K es tautologia
NIL en caso contrario
Procesamiento:
Si es null devolvemos true
Si no:
Si el primer elemento es un literal positivo, buscamos el negado en el resto de la lista
Si el primer elemento es un literal negativo, buscamos el positivo en el resto de la lista

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 tautology-p (K)
  (if (null k) t
        (tautology-rec-p k)))

(defun tautology-rec-p (K)
  (unless (null K)
        (if (positive-literal-p (first K))
        ;; Si el primer elemento es un literal positivo, buscamos el negado en el resto de la lista
        (or (some #'(lambda (x) (is-equal x (list +not+ (first K)))) (rest K))
        (tautology-rec-p (rest K)))

        ;; Si el primer elemento es un literal negativo, buscamos el positivo en el resto de la lista
        (or (some #'(lambda (x) (is-equal x (list (second (first K))))) (rest K))
        (tautology-rec-p (rest K))))))
```

EJEMPLOS:

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

4.3.6

PSEUDOCÓDIGO

Entrada: x expresion cnf
Salida: cnf sin tautologias

Procesamiento:
Si es null la devolvemos
si es tautologia, recursiva con el resto
si no mantenemos la primera y recursiva con el resto

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)
  (cond ((null cnf)  cnf)
        ((tautology-p (first cnf)) (eliminate-tautologies (rest cnf))) ;; si es tautologia, recursiva con el resto
        (t (cons (first cnf) (eliminate-tautologies (rest cnf)))))) ;; si no mantenemos la primera y recursiva con
el resto
```

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
```

4.3.7

PSEUDOCÓDIGO

Entrada: x expresion cnf

Salida: cnf sin clausulas repetidas ni literales, ni subsumidas

Procesamiento:

Eliminamos literales de la cnf:
- si alguno es null la clausula sera una lista de null
- si no:
  - eliminamos los literales repetidos

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)
  (to-cnf (elim-literals cnf)))
```

```
(defun to-cnf (cnf)
  (cond ((null cnf) cnf)
        ((some #'null cnf) (list NIL));; comprobamos si alguno es null
        (t(eliminate-subsumed-clauses (eliminate-tautologies (eliminate-repeated-clauses cnf)))))) ;; realizamos
las eliminaciones


(defun elim-literals (cnf)
  (cond ((null cnf) cnf)
        ((some #'null cnf) (list NIL));;si alguno es null
    (t(cons(eliminate-repeated-literals (first cnf));; realizamos las eliminaciones
        (elim-literals (rest 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
```

4.4.1

PSEUDOCÓDIGO

Entrada: lambda (literal positivo)
cnf (FBF simplificada)
Salida: subconjunto de clausulas de cnf que no contienen el literal lambda ni ~lambda

Procesamiento:
Si lambda es positivo y cnf no es null:
Para cada elemento de cnf, comprobamos que no contenga el literal positivo o negativo.

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)
  (when (and (positive-literal-p lambda) (not (null cnf)))
    (remove-if-not #'(lambda(x) ;; Eliminamos clausula de la lista si el resultado de subsumir no es nil
                  (and (null (subsume (list lambda) x)) (null (subsume (list (list +not+ lambda)) x))))
cnf)))
```

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
```

4.4.2

PSEUDOCÓDIGO

Entrada: lambda (literal positivo)
cnf (FBF en FNC simplificada)
Salida: subconjunto de clausulas de cnf que no contienen el literal lambda ni ~lambda

Procesamiento:
Si lambda es positivo y cnf no es null:
Para cada elemento de cnf, comprobamos que contenga el literal positivo.

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)
  (when (and (positive-literal-p lambda) (not (null cnf)))
        (remove-if #'(lambda(x) ;; Eliminamos clausula de la lista si el resultado de subsumir con el
lambda es nil
                    (null (subsume (list lambda) x))) cnf)))
```

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

4.4.3

PSEUDOCÓDIGO

Entrada: lambda (literal positivo)
cnf (FBF en FNC simplificada)
Salida: subconjunto de clausulas de cnf que no contienen el literal lambda ni ~lambda

Procesamiento:
Si lambda es positivo y cnf no es null:
Para cada elemento de cnf, comprobamos que contenga el literal negativo.

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)
  (when (and (positive-literal-p lambda) (not (null cnf)))
        (remove-if #'(lambda(x) ;; Eliminamos clausula de la lista si el resultado de subsumir con el
~lambda es nil
                      (null (subsume (list (list +not+ lambda)) x))) cnf)))
```

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
```

4.4.4

PSEUDOCÓDIGO

Entrada: lambda (literal positivo)
K1 (primera clausula simplificada
K2 (segunda clausula simplificada)
cnf (FBF en FNC simplificada)

Salida: subconjunto de clausulas de cnf que no contienen el literal lambda ni ~lambda

Procesamiento:

Si lambda está en forma postiva en una clausula y en forma negativa en otra;
Union de K1 y K2 y eliminamos de esa lista los lambdas positivos y negativos

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)
  (when (or (and (some #'(lambda (x) (is-equal lambda x)) K1) ;; Comprobamos si podemos hacer res
           (some #'(lambda (x) (is-equal (list +not+ lambda) x)) K2))
           (and (some #'(lambda (x) (is-equal lambda x)) K2)
           (some #'(lambda (x) (is-equal (list +not+ lambda) x)) K1)))

           ;; Hacemos union de K1 y K2 para despues quitar de esa lista lambda y ~lambda
           (list (remove-if #'(lambda (x) (or (is-equal x lambda) (is-equal x (list +not+ lambda))))
           (union K1 K2 :test #'is-equal)))))
```

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
```

4.4.5

PSEUDOCÓDIGO

Entrada: lambda (literal positivo)
cnf (FBF en FNC simplificada)
Salida: (El conjunto RES_lambda de cnf)

Procesamiento:
Union de: -las clasusulas neutrales de cnf con respecto a lambda
- el conjunto res (aplicar resolucion entre todas las clausulas positivas y negativas de cnf con respecto a lambda

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 build-RES (lambda cnf)
  (eliminate-repeated-clauses
   (union (extract-neutral-clauses lambda cnf)
          (res-aux1 lambda
                    (extract-positive-clauses lambda cnf)
                    (extract-negative-clauses lambda cnf))
```

```
:test #'is-equal)))

;; Funcion que aplica res a las clausulas con lambdas negativos y con lambdas positivos

(defun res-aux1 (lambda k1 k2)
  (unless (or (null k1) (null k2))
	(union (res-aux2 lambda (first k1) k2) (res-aux1 lambda (rest k1) k2) :test #'is-equal)))

;; Funcion que recibe una clausula (k1) y hace res con todas las clausulas de k2

(defun res-aux2 (lambda k1 k2)
  (unless (null k2)
(union (resolve-on lambda k1 (first k2)) (res-aux2 lambda k1 (rest k2)) :test #'is-equal)))
```

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))
```

4.5

PSEUDOCÓDIGO

Entrada: cnf (FBF en FNC simplificada)
Salida: T si cnf es SAT
NIL si cnf es UNSAT

Procesamiento:
Por cada clausula de la cnf :
- comprobamos si es RES
- la simplificamos

Si obtenemos clausula vacía:
- clausula UNSAT

Si no se pueden hacer mas resoluciones y no es vacia:
- clausula SAT

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  RES-SAT-p (cnf)
  (cond ((null cnf) t)
        ((some #'null cnf) nil) ;; LLamamos a RES-SAT-rec-p
        (t (RES-SAT-rec-p (all-posit-atoms cnf) cnf))))

;; Funcion que realiza REC-SAT-p de manera recursiva, siguiendo el algoritmo propuesto en el
enunciado

(defun RES-SAT-rec-p (latoms cnf)
  (if (or (null latoms) (null cnf)) ;; Si hemos agotado la lista de atomos o cnf ya es null, es SAT
        t
        ;; Si no, miramos si hay algun nil en la simplificacion de build-RES
        (let ((resul (simplify-cnf(build-RES (first latoms) cnf))))
        (if (some #'null resul)
        nil ;; En este caso es UNSAT
        (RES-SAT-rec-p (rest latoms) resul))))) ;; Llamamos de nuevo a la funcion con la
simplificacion

;; Funcion que devuelve todos los atomos positivos de un FBF, los negativos los pasa a positivos, sin
eliminar repetidos

(defun all-posit-atoms (cnf)
  (mapcar #'(lambda (x)
        (if (positive-literal-p x)
              x
        (second x))) (all-atoms cnf)))


;; Esta funcion coge tanto literales positivos como negativos, solo queremos positivos
(defun all-atoms (cnf)
  (unless (null cnf)
        (union (first cnf) (all-atoms (rest cnf)) :test #'is-equal)))



;;
;;  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
```

PSEUDOCÓDIGO

Entrada: wff - FBF en formato infijo
w - FBF en formato infijo

Salida: T si w es consecuencia logica de wff
NIL en caso de que no sea consecuencia logica.

Procesamiento:
Pasamos la wff a infijo y simplificamos
Si w es un literal positivo hacemos lista su negativo
Si no con el positivo
Realizamos la union de ambos y comprobamos si no es SAT

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.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Miramos si RES de wff (en FNC simplificada) U ~w es SAT o UNSAT

(defun logical-consequence-RES-SAT-p (wff w)
  (not (RES-SAT-p (union
                    (simplify-cnf (wff-infix-to-cnf wff))
```

```
                (cond ((positive-literal-p w) (list (list (list +not+ w))))
                (t (list (list (second w)))))
                :test #'is-equal))))

;;
;;  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))))
```

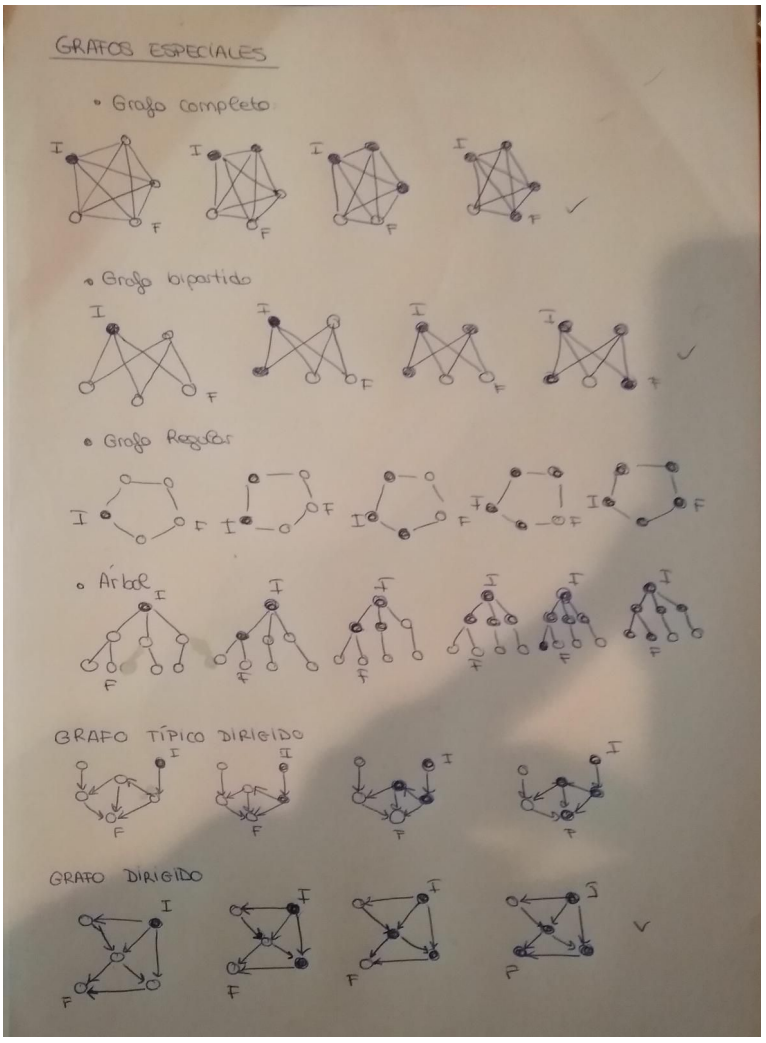
```
 'q)
;; NIL

(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

5.1 Ilustra el funcionamiento del algoritmo resolviendo a mano algunos ejemplos ilustrativos:



5.2 Escribe el pseudocódigo correspondiente al algoritmo BFS.

BFS (Grafo, nodo_ini):

Por cada vertice u del grafo
    u ← no visitado
    distance hasta u ← infinito
nodo_ini ← visitamos
distance hasta nodo_ini ← 0
cola de nodos visitados ← vacia
insertamos en la cola ← nodo_ini
mientras cola no este vacia
    u ← nodo sacado de la cola
    por cada nodo v adyacente a u
        si v no visitado, ni expandido
            entonces v ← visitamos
                distance a v ← distance a u + 1
                nodo padre de v ← u
                insertamos en la cola ← v

u ← expandido

5.3 Estudia con detalle la siguiente implementación del algoritmo BFS, tomada del libro"ANSI Common Lisp" de Paul Graham [http://www.paulgraham.com/acl.html]. Asegúrate de que comprendes su funcionamiento con algún grafo sencillo.
La función assoc devuelve la sublista dentro de una lista cuyo car sea el elemento que se le pase. Por ejemplo, si llamamos a la lista de arriba grafo, (assoc 'b grafo) devolvería la sublista (b d f).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Breadth-first-search in graphs
;; Realiza una búsqueda en anchura mejorada
;;
;; RECIBE   : end: Nodo final u objetivo
;;            queue: Cola de nodos adyacentes, o de caminos
;;            net: Grafo
;; EVALUA A : Camino realizado al nodo end
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun bfs (end queue net)
;;; Caso de error si no hay elementos en la cola
   (if(null queue) '()
;;; Nombramos elementos
      (let*((path (first queue)) ;; path es la primera lista de adyacencia                (node (first
path))) ;; node es el nodo padre
       (if(eql node end) ;; Si el nodo es el último, acabar
             (reverse path)
;;;realizamos una recursión
;;;se exploran los caminos con origen en el nodo con el que estamos
       (bfs end
             (append (rest queue)
                   (new-paths path node net))
             net)))))

;;Obtenemos la lista con los caminos que salen de node
(defun new-paths (path node net)
   (mapcar #'(lambda(n) (cons n path))
       (rest (assoc node net))))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.4 Pon comentarios en el código anterior, de forma que se ilustre cómo se ha implementado el pseudocódigo propuesto en 4.2).

5.5 Explica por qué esta función resuelve el problema de encontrar el camino más corto entre dos nodos del grafo:

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

Realiza una llamada a la función de búsqueda en anchura con el objetivo de encontrar el camino más corto y esto es porque a diferencia de búsqueda en profundidad, ésta va aumentando poco a poco de profundidad y siempre el camino siguiente que encontramos tiene más profundidad que el camino anterior, entonces cuando encontramos un camino de llegada al nodo destino sabemos que va a ser el más corto, porque si hubiera otro más corto lo habríamos encontrado antes, ya que va en orden creciente de profundidad.

5.6 Ilustra el funcionamiento del código especificando la secuencia de llamadas a las que da lugar la evaluación:
(shortest-path 'a 'f '((a d) (b d f) (c e) (d f) (e b f) (f)))
Realiza la siguiente llamada a bfs:

```
        (bfs 'f '((a)) '((a d) (b d f) (c e) (d f) (e b f) (f)))
```

Luego a la funcion new-paths

```
        (new-paths '(a) 'a '((a d)(bdf)(ce)(df)(ebf)(f)))  --> obteniendo ((d a))
```

vuelve a llamar a bfs

```
        (bfs 'f '((d a))) '((a d)(b d f) (c e) ( d f) (e b f) (f)))
```

de nuevo a new-paths

```
        (new-paths '(d a) 'd '((a d)(bdf)(ce)(df)(ebf)(f)))  --> obteniendo ((f d a))
```

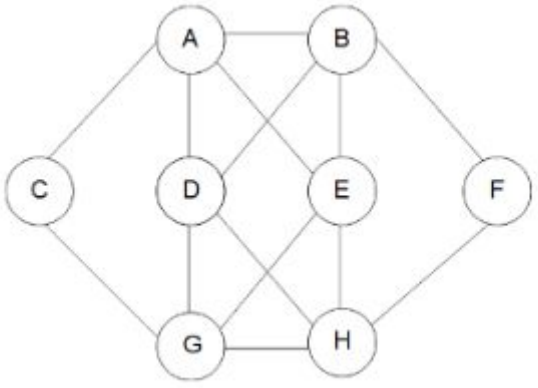
y por ultimo a bfs

```
        (bfs 'f '((f d a))) '((a d)(b d f) (c e) ( d f) (e b f) (f)))
```

y obtenemos el shortest-path -----> (A D F)

La macro trace es especialmente útil para este tipo de tareas. Con untrace se desactivan las trazas.

5.7 Utiliza el código anterior para encontrar el camino más corto entre los nodos F y C en el siguiente grafo no dirigido. ¿Cuál es la llamada concreta que tienes que hacer? ¿Qué resultado obtienes con dicha llamada?



Tienes que introducir la llamada:
(shortest-path 'f 'c '((a c b d e)(b a e d f)(c a g)(d a g h b)(e b h a g)(f b h)(g h e d c)(h e d g)))

Obtenemos:

(F B A C)

5.8 El código anterior falla entra en una recursión infinita y el problema de búsqueda no tiene solución. Ilustra con un ejemplo este caso problemático y modifica el código para corregir este problema:
(defun bfs-improved (end queue net) ...)
(defun shortest-path-improved (end queue net) ...)

El bucle infinito se produce cuando hay un ciclo cerrado entre los nodos del grafo, por ejemplo a se une con b, b con c y c con a de nuevo.

Podemos solucionarlo eliminando los nodos que ya has visitado anteriormente.