PRÁCTICA 1 INTELIGENCIA ARTIFICIAL

Grupo 2301, Pareja 03

Alejandro Cabana Suárez y Aitor Arnaiz del Val

1 Similitud Coseno

1.1 Implementa una función que calcule la similitud coseno entre dos vectores, x e y, de dos formas:

1.1.1 De forma recursiva:

```
(defun dot-product-rec (u v)
      (if (or (null u) (null v))
            0
            (+ (* (first u) (first v))
                  (dot-product-rec (rest u) (rest v)))))
```

Esta función calcula el producto escalar de dos vectores de forma recursiva. Decidimos separar el cálculo del producto escalar y de la raiz de la norma-2 de los vectores por darle una mayor simplicidad y modularidad al problema y, por consiguiente, al código.

```
(defun 2-norm-rec (v)
    (sqrt (dot-product-rec v v)))
```

La función 2-norm-rec realiza el cálculo de la raiz del producto escalar consigo mismo, esto es, la norma-2 del vector. Se decidió implementar de esta forma para reaprovechar el código de la función anterior.

```
(defun sc-rec (x y)
    (if (or (null x)
            (null y)
            (every #'zerop x)
            (every #'zerop y))
    nil
    (/ (dot-product-rec x y)
            (* (2-norm-rec x)
                  (2-norm-rec y)))))
```

La función sc-rec hace uso de las dos funciones definidas anteriormente para hacer el cálculo de la similitud coseno entre dos vectores de forma recursiva, como puede apreciarse en el código.

1.1.2 Utilizando mapcar:

Al igual que en la implementación recursiva de la función que calcula la similitud coseno, hemos optado por simplificar y dividir el trabajo en dos funciones auxiliares: por un lado, dot-product-mapcar, que calcula el producto escalar de dos vectores utilizando la directiva mapcar de lisp, así como 2-norm-mapcar, que hace una llamada a la función anterior, al igual que lo hacía su versión recursiva. Por último, hemos implementado la función sc-mapcar, que devuelve la similitud coseno de dos vectores haciendo llamadas a las dos funciones descritas anteriormente:

```
(defun dot-product-mapcar (u v)
    (reduce #' + (mapcar #' * u v)))


(defun 2-norm-mapcar (v)
    (sqrt (dot-product-mapcar v v)))


(defun sc-mapcar (x y)
    (if (or (null x)
            (null y)
            (every #'zerop x)
            (every #'zerop y))
    nil
    (/ (dot-product-mapcar x y)
            (* (2-norm-mapcar x)
                    (2-norm-mapcar y)))))
```

1.2 Codifica una función que reciba un vector que representa a una categoría, un conjunto de vectores y un nivel de confianza que esté entre 0 y 1. La función deberá retornar el conjunto de vectores ordenado según mas se parezcan a la categoría dada si su semejanza es superior al nivel de confianza.

Para este apartado, hemos definido dos funciones: por una parte, sc-conf-ind hace el mismo trabajo que la función pedida en el enunciado de este apartado pero a efectos individuales, es decir, devuelve conses cuyo primer elemento es la similitud coseno entre el vector categoría y uno de los vectores pasados como argumento, y cuyo rest es este último vector vector, o NIL en caso de que esta similitud sea menor que la umbral pasada como parámetro;

```
(defun sc-conf-ind (cat v conf)
    (let ((sc (sc-rec cat v)))
            (if (or (null sc) (<= sc conf))
                    nil
                    (cons sc v))))
```

Por otra parte, hemos definido asímismo la función sc-conf, que es la que realiza el trabajo especificado en el enunciado mediante llamadas a sc-conf-ind.

```
(defun sc-conf (cat vs conf)
    (mapcar #'rest
            (sort
                    (remove nil
                            (mapcar #'(lambda (z) (sc-conf-ind cat z conf)) vs))
                    #'> :key #'first)))
```

El único problema que se nos planteaba en este punto era que si alguno de los vectores no tenia una similitud coseno con el vector categoría mayor de la umbral se metía un NIL en su posición de la lista, y por esta razón nos hemos visto obligados a aplicar la directiva remove para quitar estos nil de nuestro vector resultado.

Por otra parte, como teníamos que devolver el vector ordenado de mayoir similitud a menor, le aplicamos la función sort a esta lista, una vez desprovista de elementos NIL.

1.3 Clasificador por similitud coseno. Codifica una función que reciba un conjunto de categorías, cuyo primer elemento es su identificador, un conjunto de vectores que representan textos, cuyo primer elemento es su identificador, y devuelva una lista que contenga pares definidos por un identificador de la categoría que maximiza la similitud coseno con respecto a ese texto y el resultado de la similitud coseno. El tercer argumento es la función usada para evaluar la similitud coseno (mapcar o recursiva).

Para este apartado hemos definido una sola función, la función sc-classifier, que por así decirlo, nos "indexa" por el vector de textos la categoría que maximiza la similitud coseno para cada uno de esos textos, devolviendo así una lista de pares (identificador de categoría, similitud coseno de esa categoría con respecto al texto actual)

```
(defun sc-classifier (cats texts func)
    (if (null texts)
        nil
        (cons (reduce #'(lambda (x y)
                (if (> (first (rest x)) (first (rest y)))
                x
                y))
        (mapcar #'(lambda (cat)
                (list (first cat)
                        (funcall func (rest cat) (rest (first texts)))))
        cats))
    (sc-classifier cats (rest texts) func))))
```

Como aspectos reseñables en la implementación de esta función podemos señalar los siguientes:

Por un lado, que es una función recursiva, que se llama a sí misma texto a texto, hasta que no queden más textos que comparar con las categorías en cats (vector de categorías).

Por otra parte, cabe destacar que como estamos accediendo a listas de números al aplicar la directiva reduce con la función "mayor que" para acceder a los números y al no tratarse de conses los elementos sobre los que trabajamos (como así era hasta el cambio de enunciado) debemos acceder a los números guardados en estas listas aplicando la función first, como se ve en la quinta línea de la función

Por último, podemos destacar el uso de funcall para llamar a la función pertinente (la cual se pasa como parámetro) dependiendo de si queremos utilizar la versión recursiva o con mapcar de obtención de la similitud coseno.

1.4 Haz pruebas llamando a sc-classifier con las distintas variantes de la distancia coseno y para varias dimensiones de los vectores de entrada. Verifica y compara tiempos de ejecución. Documenta en la memoria las conclusiones a las que llegues.

Ejecutando la función sc-classifier para varios tamaños de sublistas y y distinto número de sublistas por lista podemos apreciar que para un número pequeño de sublistas y pocos elementos por sublista funciona mejor la versión sc-mapcar, aunque esta mejora de tiempos es prácticamente inapreciable, mientas que según vamos incrementando tanto el tamaño de las sublistas como el número de sublistas podemos apreciar una mejora del rendimiento de la versión recursiva (sc-rec) con respecto a la que usa mapcar.

Esto puede deberse a que mapcar pasa dos veces por cada vector, una al sumar y otra al multiplicar, mientras que la implementación recursiva hace una sola pasada por cada vector, lo que reduce los tiempos.

2 Raíces de una función.

2.1 Implemente una función bisect que aplique el método de bisección para encontrar una raíz de una función en un intervalo dado;

Implementamos la función lógica xnor, que devuelve T si y sólo si sus dos argumentos son T o los dos son NIL. Esto nos servirá para comprobar si dos números tienen el mismo signo, ya que los signos los representaremos como booleanos (T: +, NIL: -).

```
(defun xnor (a b)
    (or (and a b)
        (and (not a)
             (not b))))
```

El cuerpo de bisect es un gran cond que comprueba primero si a o b ya son 0, o si da la casualidad de que su punto medio sea 0, en cuyo caso ya hemos encontrado la raíz.

Si ninguno lo es, entonces mira qué extremo del intervalo tiene distinto signo que el punto medio al aplicarles la función, y se llama a sí misma recursivamente con el punto medio y el extremo elegido, como nuevos extremos de intervalo.

```
(defun bisect (f a b tol)
    (let* ((m (/ (+ a b) 2))
           (fa (funcall f a))
           (fb (funcall f b))
           (fm (funcall f m))
           (siga (> fa 0))
           (sigb (> fb 0))
           (sigm (> fm 0)))
    (cond ((xnor siga sigb)
           nil)
           ((= fa 0)
           a)
```

```
((= fb 0)
b)
((= fm 0)
m)
((< (- b a) tol)
m)
((xnor sigm sigb)
(bisect f a m tol))
(t
(bisect f m b tol)))))
```

2.2 Implemente una función allroot que recibe una función, una lista de valores y una tolerancia. Si entre los elementos l[i] y l[i + 1] de la lista hay una raíz, la función devuelve el valor de la raíz. El resultado es una lista con todas las raíces encontradas.

Ahora queremos hacer lo mismo que en la función del apartado anterior, solo que para todos los elementos de una lista, así que el primer impulso es usar mapcar para conseguirlo. Sin embargo, si en alguno de estos intervalos no se encuentra raíz, bisect devolverá NIL. Como no queremos que aparezcan NIL en la solución, en lugar de usar mapcar recorremos la lista recursivamente, comprobamos el resultado de bisect para cada par y sólo lo añadimos si no es NIL.

```
(defun allroot (f lst tol)
    (if (null (rest lst))
        nil
        (let ((root (bisect f (first lst) (second lst) tol)))
            (if (null root)
                (allroot f (rest lst) tol)
                (cons root
                    (allroot f (rest lst) tol))))))
```

2.3 Implemente una función allind que recibe una funcion, dos límites del intervalo, un valor entero N y la tolerancia. La función divide el intervalo en 2 N secciones y busca en cada una una raíz de la función (dentro de la tolerancia). Devuelve una lista con las raices encontradas.

Como nos piden dividir el intervalo en 2^N trozos, pensamos en dividir recursivamente el intervalo en mitades N veces.
Utilizando el propio argumento N de la función como contador, sabemos cuándo parar la recursión. Cuando llega a 0, aplicamos bisec al fragmento de intervalo resultante, y se irán concatenando los raíces encontradas. De nuevo, para evitar que se cuelen NIL, los filtramos en este paso.

```
(defun allind (f a b N tol)
    (if (= N 0)
        (remove nil (list (bisect f a b tol)))
        (let ((m (/ (+ a b) 2)))
```

```
        (append (allind f a m (- N 1) tol)
                (allind f m b (- N 1) tol)))))
```

3 Combinación de listas

3.1 Define una función que combine un elemento dado con todos los elementos de una lista:

Para solucionar este ejercicio, basta con aplicar (list elt e) a cada elemento e de la lista. Esto lo hacemos utilizando mapcar con una lambda que cumple esa función.

```
(defun combine-elt-lst (ele lst)
    (mapcar #'(lambda (list1)
                (list ele list1)) lst))
```

3.2 Diseña una función que calcule el producto cartesiano de dos listas:

Tenemos que hacer lo que en la funcion del apartado anterior, pero para cada elemento de una lista; así que utilizamos mapcan para conseguirlo (ya que queremos los resultados de cada operación concatenados).

```
(defun combine-lst-lst (lst1 lst2)
    (mapcan #'(lambda (e)
                (combine-elt-lst e lst2))
        lst1))
```

3.3 Diseña una función que calcule todas las posibles disposiciones de elementos pertenecientes a N listas de forma que en cada disposición aparezca únicamente un elemento de cada lista:

Lo primero que se nos pasa por la cabeza al pensar este ejercicio es utilizar reduce para reducir la lista con combine-lst-lst.

Imaginemos que implementamos la función así y se la aplicamos a ((a b) (+ -) (1 2)). La primera iteración de reduce daría como resultado ((a +) (a -) (b +) (b -)), y la siguiente tendría que volver a aplicar combine-lst-lst a este resultado y (1 2). A primera vista parece que saldrá lo que queremos, pero lo que obtenemos es (((a +) 1) ((a +) 2) ((a -) 1) ...).

Nosotros querríamos exactamente este comportamiento, solo que en lugar de que se cree una nueva lista al combinar dos elementos, si el primero es una lista debería añadirse el segundo elemento al final.

Para conseguir esto, creamos una versión alternativa de combine-lst-lst y combine-elt-lst. La diferencia única diferencia entre estas versiones y las originales es que combine-elt-lst-alt comprueba si el elemento que se le pasa es una lista bien formada, y si lo es, le concatenamos el elemento oportuno al final.

```
(defun combine-elt-lst-alt (ele lst)
    (mapcar #'(lambda (list1)
                (if (proper-list ele)
                        (append ele (list list1))
                (list ele list1)))
        lst))


(defun combine-lst-lst-alt (lst1 lst2)
    (mapcan #'(lambda (e)
                (combine-elt-lst-alt e lst2))
        lst1))
```

Por supuesto, para poder hacer esto, hemos tenido que implementar una función proper-list que comprobara si algo es una lista bien formada (ya que listp en realidad hace lo mismo que consp solo que devolviendo T para NIL).

```
(defun proper-list(l)
    (or (null l)
        (and (consp l)
                (proper-list (rest l)))))
```

Además, añadimos una condición extra en combine-list-of-lsts para que actúe correctamente cuando la lista de listas sólo contiene una lista.

```
(defun combine-list-of-lsts (lstolsts)
    (if (null (rest lstolsts))
                (mapcar #'list (first lstolsts))
        (reduce #'combine-lst-lst-alt lstolsts)))
```

4 Inferencia en lógica proposicional

4.1 Predicados en LISP para definir literales, FBFs en forma prefijo e infijo, cláusulas y FNCs

4.1.1 Escribe una función LISP para determinar si una expresión es un literal positivo. Completa el código de la función positive-literal-p.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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 (connector-p x))
       (not (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
```

PSEUDOCODIGO:

```
ENTRADA: expresión logica

SALIDA: T si es un literal positivo, NIL si no

PROCESAMIENTO:
Si expresion recibida:
    Es atomo
    No es conector
    No es un valor de verdad
Entonces: Es un literal positivo
Si no, no lo es.
```

4.1.2 Escribe una función LISP para determinar si una expresión es un literal negativo. Completa el código de la función negative-literal-p.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
       (not (null x))
       (eql (first x) +not+)
       (positive-literal-p (second x))
       (null (cddr 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


PSEUDOCÓDIGO:

  ENTRADA: expresion

  SALIDA: T si es un literal negativo, NIL si no

  PROCESAMIENTO:
  Si expresion recibida:
      Es una lista
      No es NIL
      Su first es +not+
      El second es un literal positivo
      Solo tiene dos elementos
 Entonces es un literal negativo
```

4.1.3 Escribe una función LISP para determinar si una expresión es un literal. Completa el código de la función literal-p.

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


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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 conector 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:
(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
```

PSEUDOCODIGO:

ENTRADA: expresion

SALIDA: T si la expresion es un literal, NIL si no

PROCESAMIENTO:

Si la expresion:
    Es un literal positivo o
    Es un literal negativo
La expresion es un literal

4.1.4 La función wff-prefix-p facilitada en el fichero p1ej4.lisp implementa un predicado para determinar si una determinada expresión está en formato prefijo. Tomando el código de esta función como referencia, implementa una nueva función LISP para determinar si una expresión dada está en formato infijo. Completa el código de la función wff-infix-p. Puedes utilizar funciones auxiliares si lo consideras necesario.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 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-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* ((op_1      (first x))
                    (conn_1    (second x))
                    (rest_1    (cddr x))
                    (op_2      (first rest_1))
                    (rest_2    (rest rest_1))
                    (conn_2    (first rest_2)))
             (cond
              ((unary-connector-p op_1)   ;; Si el primer elemento es un conector unario
               (and (null rest_1)         ;; deberia tener la estructura (<conector> FBF)
                    (wff-infix-p conn_1)))
              ((n-ary-connector-p op_1)   ;; Si el primer elemento es un conector enario
               (null (rest x)))           ;; conjuncion o disyuncion vacias
              ((binary-connector-p conn_1) ;; Si el segundo elemento es un conector binario
               (and (null (rest rest_1)) ;; deberia tener la estructura (FBF1 <conector> FBF2)
                    (wff-infix-p op_1)
                    (wff-infix-p op_2)))
              ((n-ary-connector-p conn_1) ;; Si el segundo elemento es un conector enario
               (and (wff-infix-p op_1)     ;; el primer operando tiene que ser FBF
                    (or (and (null rest_2)            ;; si solo hay dos operandos
                             (wff-infix-p op_2))      ;; el segundo debe ser una FBF
                        (and (eql conn_2 conn_1)   ;; si hay mas operandos, el segundo conector
                                                   ;; debe ser igual al primero
                             (wff-infix-p rest_1))))) ;; y quitando el primer operando y
                                                      ;; conector, debe ser una FBF
              (t NIL)))))                  ;; No es FBF en formato infijo
```

PSEUDOCODIGO:

ENTRADA: Expresion

SALIDA: T si la expresion esta en formato prefijo, NIL si no

PROCESAMIENTO:

Si (es un literal o
una lista) y
el primer elem es un conector enario y lo hacemos sobre NIL o
el segundo elem es conector binario o
el segundo elem es conector enario y
separamos casos posibles en este caso (2 operandos/mas operandos)
Entonces: esta en prefijo

```
;;
;; 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

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

;;
;;  EJEMPLOS:
;;
(prefix-to-infix '(v))          ; (V)
(prefix-to-infix '(^))          ; (^)
(prefix-to-infix '(v a))        ; A
(prefix-to-infix '(^ a))        ; A
(prefix-to-infix '(^ (¬ a)))    ; (¬ a)
(prefix-to-infix '(v a b))      ; (A v B)
(prefix-to-infix '(v a b c))    ; (A V B V C)
(prefix-to-infix '(^ (V P (=> A (^ B (¬ C) D))) (^ (<=> P (¬ Q)) P) E))
;;; ((P V (A => (B ^ (¬ C) ^ D))) ^ ((P <=> (¬ Q)) ^ P) ^ E)
(prefix-to-infix '(^ (v p (=> a (^ b (¬ c) d))))) ; (P V (A => (B ^ (¬ C) ^ D)))
(prefix-to-infix '(^ (^ (<=> p (¬ q)) p ) e))     ; (((P <=> (¬ Q)) ^ P) ^ E)
(prefix-to-infix '( v (¬ p) q (¬ r) (¬ s)))       ; ((¬ P) V Q V (¬ R) V (¬ S))
```

4.1.5 La función prefix-to-infix facilitada en el fichero p1ej4.lisp transforma una FBF en formato prefijo a una FBF en formato infijo. Tomando el código de esta función como referencia, implementa una nueva función LISP para transformar FBFs en formato infijo a formato prefijo. Completa el código de la función infix-to-prefix.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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 ((op_1      (first wff))
            (connector (second wff))
            (op_2      (third wff))
            (rest_1    (cddr wff)))
        (cond
         ((unary-connector-p op_1)
          (list op_1 (infix-to-prefix connector)))
         ((n-ary-connector-p op_1)           ;; conjuncion o disyuncion vacias.
          wff)
         ((binary-connector-p connector)
          (list connector
                (infix-to-prefix op_1)
                (infix-to-prefix op_2)))
```

```
         ((n-ary-connector-p connector)
          (cons connector                                    ;; concateno el conector
                (mapcan #'(lambda (x)
                                  (unless (connector-p x)   ;; con los operandos en infijo
                                    (list (infix-to-prefix x))))
                  wff))))))))
```

PSEUDOCODIGO:

ENTRADA: FBF en infijo

SALIDA: FBF en prefijo

PROCESAMIENTO:

Si estamos ante una wff en infijo:
Si es un literal: era una prefijo, devolvemos
Si era una conjuncion o disyuncion vacias, devolvemos la wff que nos pasaban
si no, cogemos todos los elementos de la wff en prefijo y les metemos el conector al

```
;;
;; 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))
```

4.1.6 Escribe una función LISP para determinar si una FBF en formato prefijo es una cláusula (disyuntiva). Completa el código de la función clause-p. Puedes utilizar funciones auxiliares si lo consideras necesario.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
             (not (literal-p wff))
             (eql (first wff) +or+))
    (literals-p (rest wff))))

;; Determina si todos los elementos de una lista son literales
(defun literals-p (lst)
(1)  (or (null lst)
(2)      (and (literal-p (first lst))
(3)           (literals-p (rest lst)))))

(1) Caso base (recoge clausula vacia)
(2) Recursiva y no con mapcan para que pare al encontrar
(3) un elemento no literal, y no recorra toda la lista
```

```
PSEUDOCODIGO:

  ENTRADA: FBF en formato prefijo

  SALIDA: T si la FBF es clausula, NIL si no.

  PROCESAMIENTO:
      Si la FBF:
          Es una fbf y
          No es un literal y
          El conector es un or
      Vemos si los elementos de la wff menos el conector son literales y en tal caso es una
```

```
;;
;; 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
```

4.1.7 Escribe una función LISP para determinar si una FBF en formato prefijo está en FNC. Completa el código de la función cnf-p. Puedes utilizar funciones auxiliares si lo consideras necesario.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
             (not (literal-p wff))
             (eql (first wff) +and+))
    (clauses-p (rest wff))))

;; Determina si todos los elementos de una lista son clausulas
(defun clauses-p (lst)
```

```
(1)  (or (null lst)
(2)      (and (clause-p (first lst))
(3)           (clauses-p (rest lst)))))

(1) Caso base (recoge conjuncion vacia)
(2) Recursiva y no con mapcan para que pare al encontrar
(3) un elemento no clausula, y no recorra toda la lista

PSEUDOCODIGO:

  ENTRADA:FBF prefijo

  SALIDA: T si la FBF esta en FNC, NIL si no

  PROCESAMIENTO:
     Si la fbf estaba en prefijo y
        la fbf no es un literal y
        el primer elemento es un and:
            Miro a ver si el resto de la fbf (salvo conector) es una clausula y devuelvo el


;;
;; 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
```

4.2 Algoritmo de transformación de una FBF a FNC

4.2.1 La función eliminate-biconditional facilitada en el fichero p1ej4.lisp elimina el conector bicondicional en una FBF. Incluye comentarios en el código de la función eliminate-biconditional.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
(1)  (if (or (null wff) (literal-p wff))
      wff
    (let ((connector (first wff)))
(2)      (if (eq connector +bicond+)
(3)          (let ((wff1 (eliminate-biconditional (second wff)))
(4)                (wff2 (eliminate-biconditional (third  wff))))
(5)            (list +and+
(6)                  (list +cond+ wff1 wff2)
(7)                  (list +cond+ wff2 wff1)))
(8)        (cons connector
(9)              (mapcar #'eliminate-biconditional (rest wff)))))))



(1) Un literal no tiene conector bicondicional que eliminar
(2) si el conector es bicondicional (<=> fbf1 fbf2)
(3) elimina bicondicionales de fbf1
(4) elimina bicondicionales de fbf2
(5) devuelve (^                   )
(6) (=> fbf1 fbf2)
(7) (=> fbf2 fbf1)
(8) si es otro conector
(9) elimina bicondicionales de todos los operandos


PSEUDOCODIGO:

  ENTRADA: FBF prefijo con un bicondicional

  SALIDA: FBF equivalente sin el bicondicional

  PROCESAMIENTO: Si la fbf era null la devuelvo;
                 Cojo los argumentos del bicondicional (second y third) y elimino este haci
```

```
                                un and de los dos argumentos del bicondicional con condicion
                                simples en ambos sentidos


;;
;; 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 Escribe una función LISP para eliminar el conector condicional en una FBF. Com- pleta el código de la función eliminate-conditional.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
(1) (if (or (null wff) (literal-p wff))
      wff
    (let ((connector (first wff)))
(2)      (if (eq connector +cond+)
(3)          (let ((wff1 (eliminate-conditional (second wff)))
(4)                (wff2 (eliminate-conditional (third  wff))))
(5)            (list +or+
(6)                  (list +not+ wff1)
(7)                  wff2))
(8)        (cons connector
(9)              (mapcar #'eliminate-conditional (rest wff)))))))


(1) Un literal no tiene conector condicional que eliminar
(2) si el conector es condicional (=> fbf1 fbf2)
(3) elimina condicionales de fbf1
(4) elimina condicionales de fbf2
(5) devuelve (v)
(6) (¬ fbf1)
(7) fbf2
(8) si es otro conector
(9) elimina condicionales de todos los operandos


PSEUDOCODIGO:
```

```
  ENTRADA: FBF con conectores =>

  SALIDA: FBF equivalente sin conectores =>

  PROCESAMIENTO:
      Cogemos los argumentos del => y lo cambiamos a su
      equivalente +not+arg1 v arg2


;;
;; 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 (¬
```

4.2.3 Escribe una función LISP para reducir el ámbito de la negación en una FBF. Completa el código de la función reduce-scope-of-negation. Puedes utilizar la función auxiliar exchange-and-or que te facilitamos, así como utilizar otras funciones auxiliares si lo consideras nece- sario.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
(1)  (if (or (null wff) (literal-p wff))
      wff
    (let ((connector (first wff)))
(2)      (if (eq connector +not+)
          (let* ((wff_2       (cadr wff))
                 (connector_2 (first wff_2)))
            (cond
(3)             ((eql connector_2 +not+)
(4)              (reduce-scope-of-negation (cadr wff_2)))
(5)             ((or (eql connector_2 +and+)
(6)                  (eql connector_2 +or+))
(7)              (reduce-scope-of-negation
(8)               (cons (exchange-and-or connector_2)
                     (mapcar #'(lambda (x)
(9)                                 (list +not+ x))
```

```
(10)                        (rest wff_2)))))
             (t NIL))) ;; No deberia darse nunca
(11)        (cons connector
(12)              (mapcar #'reduce-scope-of-negation (rest wff)))))))

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

(1) Un literal no puede reducir su negacion
(2) si el conector es negacion (¬ fbf)
(3) y el conector de la fbf negada tambien es negacion (¬ (¬ fbf2))
(4) elimina las dos negaciones, y devuelve la fbf2 con negaciones reducidas
(5) si el conector de la fbf negada es and (¬ (^ fbf2 ... fbfn))
(6) o es or (¬ (v fbf2 ... fbfn))
(7) reduzco las negaciones del resultado de
(8) cambiar el conector (and por or y viceversa)
(9) y negar
(10) cada fbf a las que estaba aplicado
(11) si el primer conector no es una negacion
(12) reduzco el ambito de las negaciones de todas las fbf


PSEUDOCODIGO:

    ENTRADA: FBF que no contiene los conectores <=>, =>

    SALIDA: FBF equivalente en formato prefijo en la que la negacion  aparece unicamente en

    PROCESAMIENTO: Para cada sublista de la lista:
                              Si el primer elemento es +not+:
                                     y si el segundo tambien es un not:
                                             Se llama recursivamente con toda la lista menos
                                     Y si el segundo no es un +not:
                                                  si es un +and+ lo cambiamos por un +or+ y
                                                  niega todos los demas elementos




;;
;;  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 La función cnf facilitada en el fichero p1ej4.lisp, junto con las funciones auxiliares simplify, exchange-NF, exchange-NF-aux y combine-elt-lst, traduce una FBF (en formato prefijo, sin conectores condicional y bicondicional y con la negación siempre en literales negativos) a FNC. Incluye comentarios en el código de todas estas funciones. Una vez implementado el algoritmo de transformación a FNC, implementarás algunas funciones adicionales para simplificar las listas que representan la FBF.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun combine-elt-lst (elt lst)
(1)  (if (null lst)
(2)      (list (list elt))
(3)    (mapcar #'(lambda (x) (cons elt x)) lst)))

(4)(defun exchange-NF (nf)
(5)  (if (or (null nf) (literal-p nf))
      nf
    (let ((connector (first nf)))
(6)      (cons (exchange-and-or connector)
           (mapcar #'(lambda (x)
(7)                           (cons connector x))
(8)                (exchange-NF-aux (rest nf)))))))

(9)(defun exchange-NF-aux (nf)
(10)  (if (null nf)
      NIL
    (let ((lst (first nf)))
(11)      (mapcan #'(lambda (x)
(12)                  (combine-elt-lst
                   x
(13)                    (exchange-NF-aux (rest nf))))
(14)        (if (literal-p lst) (list lst) (rest lst))))))

(15)(defun simplify (connector lst-wffs)
(16)  (if (literal-p lst-wffs)
      lst-wffs
(17)    (mapcan #'(lambda (x)
                (cond
(18)                 ((literal-p x) (list x))
```

```
(19)                 ((equal connector (first x))
                  (mapcan
(20)                       #'(lambda (y) (simplify connector (list y)))
                    (rest x)))
                 (t (list x))))
      lst-wffs)))

(defun cnf (wff)
  (cond
(21)   ((cnf-p wff) wff)
(22)   ((literal-p wff)
(23)    (list +and+ (list +or+ wff)))
   ((let ((connector (first wff)))
      (cond
(24)       ((equal +and+ connector)
(25)        (cons +and+ (simplify +and+ (mapcar #'cnf (rest wff)))))
(26)       ((equal +or+ connector)
(27)        (cnf (exchange-NF (cons +or+ (simplify +or+ (rest wff))))))))))


(1) si la lista es vacia
(2) devuelve ((elt)), que es como si hubiera hecho el caso general
    con (NIL) como lista

(3) a cada elemento de la lista (que va a ser una fbf) le añade
    elt delante (... (elt . list[i]) ...)

(4) en esencia aplica la propiedad distributiva: (v a (^ b c) d) pasa
    a ser (^ (v a b d) (v a c d))

(5) añade los conectores apropiados a lo que devuelve exchange-NF-aux
(6) cambia el conector (and por or y viceversa)
(7) le añade el conector original
(8) a cada una de las combinaciones de los elementos de nf
    (exchange-nf-aux quita los conectores al combinar)

(9) aplica la propiedad distributiva, pero sin tener en cuenta los conectores,
    luego los añade exchange-NF

(10) (a (^ b c) d) pasa a ser ((a b d) (a c d))
(11) cada elemento de nf
(12) lo combina
(13) con todas las combinaciones de los siguientes
(14) quitando los conectores de haberlos
(15) simplifica conectores que son iguales al que se le pasa:
(16) Con conector v, ((v a b) (^ c d) (v e)) pasa a ser (a b (^ c d) e)
(17) para cada fbf de la lista
(18) si es un literal, lo devuelve (en una lista para poder concatenar)
(19) si el conector de la fbf es el que busca
(20) simplifica el mismo conector dentro de la fbf
(21) si ya esta en FNC, ya esta
```

```
(22) si es un literal
(23) añade and y or para que este en FNC (^ (v a))
(24) si es una conjuncion
(25) simplifica los and de las fbfs en FNC (^ (^ a b) c) = (^ a b c)
(26) si es una disyuncion
(27) simplifica los or, con lo cual queda una disyuncion de literales y conjunciones,
     les aplica la propiedad distributiva, dejando una conjuncion de disyunciones
     y finalmente vuelve a llamarse a si misma para simplificar cada una de las fbfs


EJEMPLOS:

(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))
```

4.2.5 Escribe una función LISP que, dada una FBF en FNC, elimine los conectores conjunción ($\wedge$) y disyunción ($\vee$) para pasar a un formato de lista de listas en el que la conjunción de cláusulas y la disyunción de literales dentro de cada cláusula están sobreentendidas. Completa el código de la función eliminate-connectors.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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-connectors (cnf)
  (when (cnf-p cnf)
    (eliminate-connectors-rec cnf)))

(defun eliminate-connectors-rec (fbf)
  (mapcar #'(lambda (x)
              (if (literal-p x)
                  x
                (eliminate-connectors-rec x)))
    (rest fbf)))


PSEUDOCODIGO:

    ENTRADA: FBF en FNC con conectores ^, v

    SALIDA: FBF en FNC (con conectores ^, v eliminados)

    PROCESAMIENTO:
        Para cada clausula de la cnf excepto el conector:
            se llama recursivamente.



EJEMPLOS:

(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)))

(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))
```

4.2.6 Escribe una función LISP que transforme una FBF en notación infija a FNC con conjunciones y disyunciones implícitas. Completa el código de la función wff-infix-to-cnf.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
  (when (wff-infix-p wff)
    (eliminate-connectors             ;; conectores implicitos
     (cnf                             ;; PASO 4
      (reduce-scope-of-negation       ;; PASO 3
       (eliminate-conditional         ;; PASO 2
        (eliminate-biconditional      ;; PASO 1
         (infix-to-prefix wff))))))))) ;; pasar a infijo


PSEUDOCODIGO:

    ENTRADA: FBF

    SALIDA: FBF to FNC

    PROCESAMIENTO: Pasamos a prefijo la FBF, eliminamos el bicondicional del resultado de e
                   Eliminamos condicionales, reducimos el ambito de la negacion, convertimo
                   Eliminamos conectores
```

```
;;
;; 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 Simplificación de FBFs en FNC

4.3.1 Escribe una función LISP que elimine los literales repetidos en una cláusula. Completa el código de la función eliminate-repeated-literals.

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

(1)(defun eliminate-repeated-literals (k)
(2)  (unless (null k)
(3)    (adjoin (first k)
(4)            (eliminate-repeated-literals (rest k))
(5)            :test #'equal)))

(2) tratando la clausula como un conjunto
(3) añade cada literal
(4) al resto de la lista sin literales repetidos


PSEUDOCODIGO:

    ENTRADA: clausula

    SALIDA: clausula sin literales repetidos

    PROCESAMIENTO: Añade tratando como un conjunto cada elemento de la lista al
                   resto de la lista sin literales repetidos.


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

4.3.2 Escribe una función LISP que elimine las cláusulas repetidas en una FNC. Com- pleta el código de la función eliminate-repeated-clauses. Puedes utilizar funciones auxiliares si lo consideras necesario.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(1) (defun eliminate-repeated-clauses (cnf)
(2)   (unless (null cnf)
(3)     (adjoin (eliminate-repeated-literals (first cnf))
(4)             (eliminate-repeated-clauses (rest cnf))
(5)             :test-not #'(lambda (set1 set2)
(6)                           (set-exclusive-or set1 set2 :test #'equal)))))

(2) tratando la FNC como un conjunto
(3) añade la primera clausula (eliminando sus literales repetidos)
(4) al resto de la lista sin clausulas repetidas
(5) comparandolas con igualdad de conjuntos


PSEUDOCODIGO:

    ENTRADA: FNC

    SALIDA: FNC sin clausulas repetidas

    PROCESAMIENTO: Añade tratando como un conjunto cada clausula de la lista
                   (sin literales repetidos) al resto de la lista sin clausulas repetidas.


;; (si xor de dos es nil es porque los dos son vacios o los dos tienen
;; las mismas clausulas)

;; Elimino literales repetidos para cumplir el ejemplo

;;
;; 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 Escribe una función LISP que determine si una cláusula subsume a otra. Completa el código de la función subsume.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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 (subsetp K1 K2 :test #'equal) ;; K1 subsume a K2 <=> K1 contenido en K2
    (if (null K1)
        (list K1)
      K1)))
```

PSEUDOCODIGO:

ENTRADA: Dos clausulas K1, K2

SALIDA: K1 si esta subsume a K2, NIL si no

PROCESAMIENTO: Hace lo mismo que subsetp, pero con estas clausulas

```
;;
;;  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 Escribe una función LISP que elimine las cláusulas subsumidas en una FNC. Com- pleta el código de la función eliminate-subsumed-clauses. Puedes utilizar funciones auxiliares si lo consideras necesario.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(1) (defun eliminate-subsumed-clauses (cnf)
(2)   (unless (null cnf)
(3)     (my-adjoin (eliminate-repeated-literals (first cnf))
(4)                (eliminate-subsumed-clauses (rest cnf)))))

(3) añade la primera clausula (sin literales repetidos)
(4) al resto de la NFC ya sin clausulas subsumidas




(1)(defun my-adjoin (k cnf)
(2)  (let ((1st (first cnf))
(3)        (rst (rest cnf)))
(4)    (cond ((null cnf)
(5)           (list k))
(6)          ((subsume 1st k)
(7)           cnf)
(8)          ((subsume k 1st)
(9)           (my-adjoin k rst))
(10)          (t
(11)           (cons 1st
(12)                 (my-adjoin k rst))))))

(4) Si llega al final de la lista es porque ningun elemento subsume a k
(5) asi que devuelve k
(6) Si un elemento subsume a k
(7) no hace falta añadirlo, se devuelve la cnf tal cual
(8) Si k subsume a un elemento, ese elemento no importa
(9) asi que se añade k al resto de la cnf
(10) Si k no subsume ni es subsumido por el primer elemento
(11) no me salto el elemento, y lo concateno
(12) Con el resultado de añadir k al resto de la cnf
```

PSEUDOCODIGO:

ENTRADA: cnf (FBF en FNC)

SALIDA: FBF en FNC equivalente a cnf sin clausulas subsumidas

PROCESAMIENTO: añade cada cláusula (sin literales repetidos)
al resto de la NFC sin cláusulas subsumidas

```
;;
;;  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 Escribe una función LISP que determine si una cláusula es tautología. Completa el código de la función tautology-p. Puedes utilizar funciones auxiliares si lo consideras necesario.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
  (unless (null K)                      ;; NIL es falso
    (let ((1st (first K))
          (rst (rest K)))
      (or (is-negation-of-any 1st rst)  ;; o bien el primer elemento es negacion de alguno
                                        ;; de los demas
          (tautology-p rst)))))         ;; o bien se le ha hecho un or con una tautologia

(defun negate-literal (l)
  (if (positive-literal-p l)
      (list +not+ l)
    (cadr l)))

(defun is-negation-of-any (l K)
  (unless (null K)
    (or (equal l (negate-literal (first K)))
        (is-negation-of-any l (rest K)))))
```

PSEUDOCODIGO:

ENTRADA: clausula

SALIDA: T si la clausula es tautologia, NIL si no

```
    PROCESAMIENTO: si el primer elemento es negacion de alguno de los demas o se le
                   ha hecho negacion con alguna tautologia entonces devolvemos T; si no, NI
```

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

4.3.6 Escribe una función LISP que elimine las cláusulas en una FNC que son tautología. Completa el código de la función eliminate-tautologies.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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))
```

```
PSEUDOCODIGO:

    ENTRADA: FNC

    SALIDA: FBF en FNC equivalente a la entrada pero sin tautologias

    PROCESAMIENTO: recorro la fnc argumento eliminando tautologias
```

```
;;
;;  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 Escribe una función LISP que simplifique FBFs en FNC mediante la aplicación sucesiva de las eliminaciones anteriores: eliminación de literales repetidos en cada una de las cláusulas, eliminación de cláusulas repetidas, eliminación de tautologías y eliminación de cláusulas subsumidas. Completa el código de la función simplify-cnf.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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 cnf)))) ;; eliminate-repeated-clauses ya elimina
                                        ;; los literales repetidos


PSEUDOCODIGO:

    ENTRADA: cnf FBF en FNC

    SALIDA: FNC equivalente sin clausulas repetidas,
              sin literales repetidos en las clausulas
              y sin clausulas subsumidas

    PROCESAMIENTO: elimina clausulas repetidas de la cnf pasada, elimina tautologias de
                   lo que de esto y elimina clausulas subsumidas en esta.


;;
;;  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 Construcción de RES

4.4.1 Escribe una función LISP que calcule el conjunto de cláusulas lambda-neutras para una FNC. Completa el código de la función extract-neutral-clauses.

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

```
;; EJERCICIO 4.4.1
;; Construye el conjunto de clausulas lambda-neutras para una FNC
;;
;; RECIBE   : cnf    - FBF en FNC 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)
  (remove-if #'(lambda (k)
                 (contains-it-or-negation lambda k))
             cnf))

(defun contains-it-or-negation (l k)
  (unless (null k)
    (or (equal (first k) l)
        (equal (first k) (negate-literal l))
        (contains-it-or-negation l (rest k)))))
```

PSEUDOCODIGO:

```
    ENTRADA: cnf    - FBF en FNC simplificada
             lambda - literal positivo

    SALIDA: cnf_lambda^(0) subconjunto de clausulas de cnf
            que no contienen el literal lambda ni ¬lambda

    PROCESAMIENTO: Elimina las clausulas que contienen a lambda o a su
                   negacion.
```

```
;;
;;  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 Escribe una función LISP que calcule el conjunto de cláusulas lambda-positivas para una FNC. Completa el código de la función extract-positive-clauses.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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-not #'(lambda (k)
                     (member lambda k :test #'equal))
                 cnf))



PSEUDOCODIGO:

    ENTRADA: cnf    - FBF en FNC simplificada
             lambda - literal positivo

    SALIDA: cnf_lambda^(+) subconjunto de clausulas de cnf
            que contienen el literal lambda

    PROCESAMIENTO: Elimina las clausulas que no contienen a lambda.



;;
;;  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 Escribe una función LISP que calcule el conjunto de cláusulas lambda-negativas para una FNC. Completa el código de la función extract-negative-clauses.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
  (remove-if-not #'(lambda (k)
                     (member (negate-literal lambda) k :test #'equal))
                 cnf))
```

PSEUDOCODIGO:

```
    ENTRADA: cnf    - FBF en FNC simplificada
             lambda - literal positivo

    SALIDA: cnf_lambda^(-) subconjunto de clausulas de cnf
            que contienen el literal ¬lambda

    PROCESAMIENTO: elimina las clausulas que no contienen a ¬lambda
```

```
;;
;;  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 Escribe una función LISP que calcule el resolvente entre dos cláusulas res_lambda(K1, K2). El resolvente res_lambda(K1, K2) es el resultado de aplicar resolucion en lambda sobre K1 y K2, con los literales repetidos eliminados. Completa el código de la función resolve-on.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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 ((¬lambda (negate-literal lambda)))
    (when (or (and (member lambda K1 :test #'equal)
                   (member ¬lambda K2 :test #'equal))
              (and (member lambda K2 :test #'equal)
                   (member ¬lambda K1 :test #'equal)))
      (list
       (set-difference (union K1 K2 :test #'equal)
                       (list lambda ¬lambda) :test #'equal)))))
```

PSEUDOCODIGO:

```
    ENTRADA: lambda      - literal positivo
             K1, K2      - clausulas simplificadas

    SALIDA: res_lambda(K1,K2)
            lista que contiene la
                  clausula que resulta de aplicar resolucion
                  sobre K1 y K2, con los literales repetidos
```

```
                  eliminados

    PROCESAMIENTO: Si una clausula contiene a lambda y otra a su negacion:
                         Devuelve la union de las dos clausulas sin lambda y no lambda



;;
;;  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 Escribe una función LISP que calcule el conjunto RES_lambda(alpha) para una FNC alpha. Completa el código de la función build-RES. Puedes utilizar funciones auxiliares si lo consideras necesario.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
  (unless (null cnf)
    (let ((pos-k (extract-positive-clauses lambda cnf))
          (neg-k (extract-negative-clauses lambda cnf))
          (neut-k (extract-neutral-clauses lambda cnf)))
```

```
      (eliminate-repeated-clauses
       (append neut-k
               (mapcan #'(lambda (x)
                           (resolve-on lambda (first x) (second x)))
                 (combine-lst-lst pos-k
                                  neg-k)))))))

;; Cogidas de un ej anterior
(defun combine-elt-lst (ele lst)
  (mapcar #'(lambda (list1)
              (list ele list1)) lst))

(defun combine-lst-lst (lst1 lst2)
  (mapcan #'(lambda (e)
              (combine-elt-lst e lst2))
    lst1))
```

PSEUDOCODIGO:

ENTRADA: lambda - literal positivo
cnf - FBF en FNC simplificada

SALIDA: RES_lambda(cnf) con las clauses repetidas eliminadas

PROCESAMIENTO:

```
;;
;;  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 Algoritmo para determinar si una FNC es SAT

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
  (if (null cnf)  ;; Si es conjuncion vacia, es tautologia
      T           ;; SAT
    (RES-SAT-p-rec (positive-literals cnf) cnf))) ;; alpha_0 = cnf

(defun RES-SAT-p-rec (pos-lits alpha)
  (cond
   ((equal alpha (list NIL))  ;; Si llega a la clausula vacia
    NIL)                      ;; UNSAT
   ((null pos-lits) ;; Si no quedan literales positivos, no se puede seguir resolviendo
    T)              ;; SAT
   (T                                                    ;; En cualquier otro caso
    (RES-SAT-p-rec (rest pos-lits)                       ;; con el siguiente literal pos
                   (simplify-cnf                         ;; simplificar alpha_j
                    (build-RES (first pos-lits) alpha)))))) ;; alpha_j = RES_lambda(alpha_j

(defun positive-literals (cnf) ;; Devuelve los literales positivos de cnf
  (mapcan #'(lambda (k)
              (remove-if #'negative-literal-p k))
    cnf))
```

PSEUDOCODIGO:

ENTRADA: cnf - FBF en FNC simplificada

SALIDA: T si cnf es SAT
        NIL si cnf es UNSAT

PROCESAMIENTO:

```
;;
;;  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
```

4.6 Algoritmo para determinar si una FBF es consecuencia lógica de una base de conocimiento

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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                            ;; si no es SAT entonces es consecuencia logica
   (RES-SAT-p                     ;; mira si es SAT
    (simplify-cnf                 ;; simplifica cnf
     (wff-infix-to-cnf            ;; convierte en cnf
      (list wff                   ;; (wff      )
            +and+                 ;;      ^
            (list +not+ w)))))))  ;;        ¬w
```

PSEUDOCODIGO:

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: Añadimos la meta negada e intentamos resolver y llegar

```
                    a la clausula vacia; si llegamos, Es SAT, devolvemos NIL,
                    si no, T



;;
;;  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
```

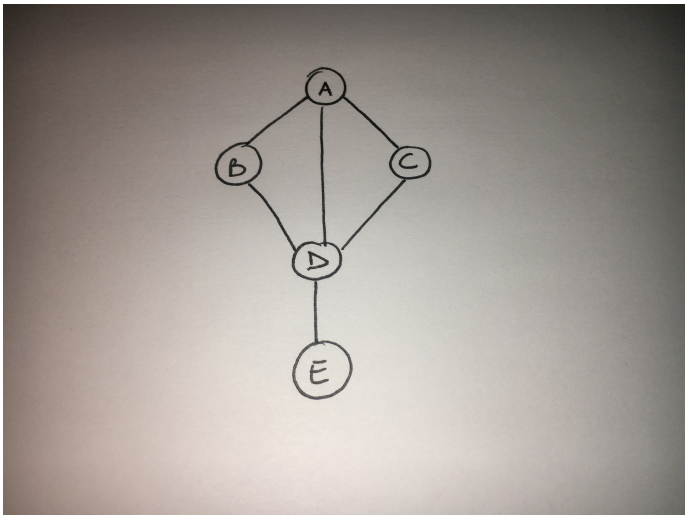
```
 '(((¬ p) => q) ^ (p <=> ((¬ a) ^ b)) ^ ( (¬ p) => (r  ^ (¬ q))))
 '(¬ q))
;; NIL

(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)))
```

5 Búsqueda en anchura

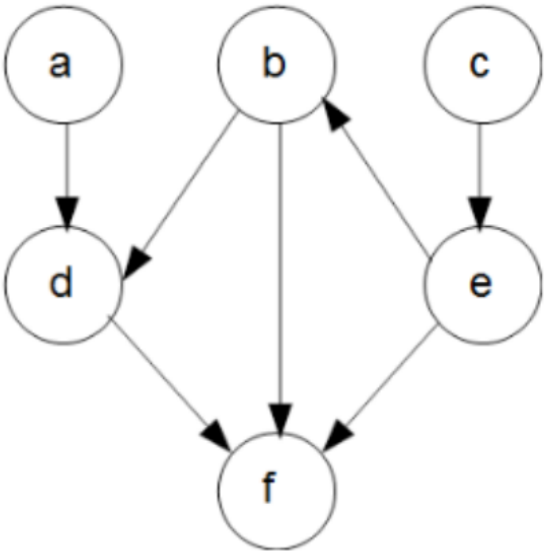
5.1 Ilustra el funcionamiento del algoritmo resolviendo a mano algunos ejemplos ilustrativos:

5.1.1 Grafos especiales



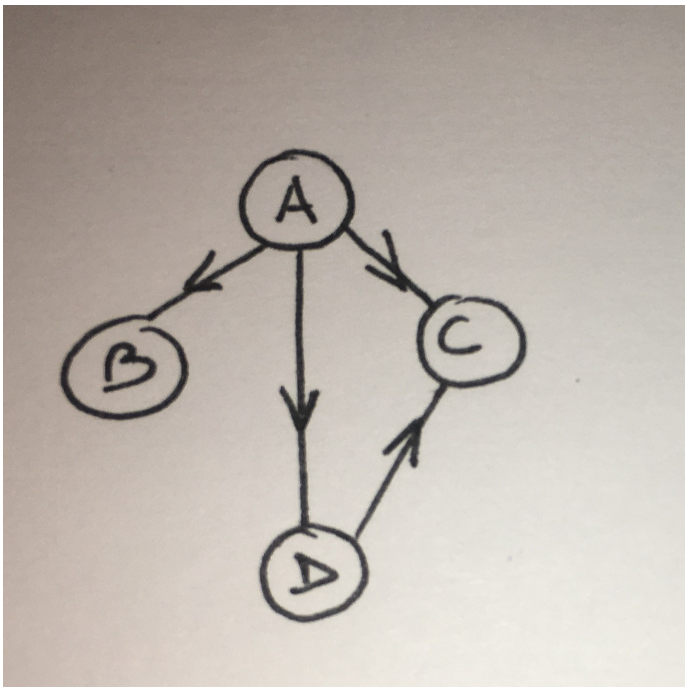
Empezamos la busqueda en el nodo A, expandimos, encolamos B, D, C; expandimos B; Expandimos D, encolamos E; expandimos C; Expandimos E, es nodo meta y acabamos.

5.1.2 Grafos dirigidos: ejemplo



Empezamos la búsqueda en el nodo A buscando el nodo F:
Expandimos A; Metemos D en la cola; Expandimos D y metemos F en la cola; Expandimos F y como es el nodo meta acabamos.

5.1.3 Grafos dirigidos: otros



Empezando la búsqueda del nodo D desde el nodo A:

Exploramos B, lo añadimos a la cola
Exploramos D, lo añadimos a la cola
Exploramos C, lo añadimos a la cola
Expandimos B, es terminal, no metemos nada a la cola Expandimos D, es el nodo meta;
Acabamos BFS.

5.2 Escribe el pseudocódigo correspondiente al algoritmo BFS.

PSEUDOCÓDIGO:

```
BFS(grafo G, nodo_fuente s)
  {
     // recorremos todos los vértices del grafo inicializándolos a NO_VISITADO,
     // distancia INFINITA y padre de cada nodo NULL
     for u ∈ V[G] do
     {
        estado[u] = NO_VISITADO;
        distancia[u] = INFINITO; /* distancia infinita si el nodo no es alcanzable */
        padre[u] = NULL;
     }
     estado[s] = VISITADO;
     distancia[s] = 0;
     padre[s] = NULL;
     CrearCola(Q); /* nos aseguramos que la cola está vacía */
     Encolar(Q, s);
     while !vacia(Q) do
     {
```

```
        // extraemos el nodo u de la cola Q y exploramos todos sus nodos adyacentes
        u = extraer(Q);
        for  v ∈ adyacencia[u]  do
        {
           if estado[v] == NO_VISITADO then
           {
                estado[v] = VISITADO;
                distancia[v] = distancia[u] + 1;
                padre[v] = u;
                Encolar(Q, v);
           }
        }
     }
  }
```

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.

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


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

5.4 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.

(2)si la cola que nos pasan esta vacia no hay camino

(3)definimos el primer elemento de la cola como path, empezare a explorar por esa rama

(4)empiezo a explorar el primer nodo de la rama, node

(5)si el nodo que estoy explorando es el nodo meta, ya he acabado

(6)en tal caso, devuelvo el camino que he hecho (lo volteo primero,

```
pues estaba al reves por la logica del programa)

(7)llamo recursivamente a la funcion para que siga explorando en anchura

(8)concateno al final de mi cola el resto de caminos que tengo que explorar
(al final para hacer bfs y no dfs)
```

```
(1)(defun bfs (end queue net)
(2)     (if (null queue) '()
(3)         (let* ((path (first queue))
(4)             (node (first path)))
(5)             (if (eql node end)
(6)                 (reverse path)
(7)                 (bfs end
(8)                    (append (rest queue)
(9)                       (new-paths path node net))
(10)                   net)))))
```

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

Esta función realiza la busqueda en anchura (bfs) entre dos nodos de un grafo (net) enlaces de costes uniformes con lo que si hay solucion, la optima siempre se encontrara a la menor profundidad con lo que esta funcion encuentra siempre el camino mas corto en el grafo net entre los nodos start y end.

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

```
0: (SHORTEST-PATH A F ((A D) (B D F) (C E) (D F) (E B F) (F)))
    1: (BFS F ((A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
      2: (NEW-PATHS (A) A ((A D) (B D F) (C E) (D F) (E B F) (F)))
      2: NEW-PATHS returned ((D A))
      2: (BFS F ((D A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
        3: (NEW-PATHS (D A) D ((A D) (B D F) (C E) (D F) (E B F) (F)))
       3: NEW-PATHS returned ((F D A))
       3: (BFS F ((F D A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
       3: BFS returned (A D F)
     2: BFS returned (A D F)
   1: BFS returned (A D F)
 0: SHORTEST-PATH returned (A D F)
```

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?

```
(setf grafo7 '((a b c d e)(b a d e f)(c a g)(d a b g h)
              (e a b g h)(f b h)(g c d e h)(h d e f g)))
```

```
(shortest-path 'f 'c grafo7)
```

El resultado obtenido es el correcto: (F B A C)

5.8 El código anterior falla (entra en una recursión infinita) cuando hay ciclos en el grafo y el problema de búsqueda no tiene solución. Ilustra con un ejemplo este caso problemático y modifica el código para corregir este problema:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; hay-elementos-repetidos (camino-recorrido)
;;;
;;; Funcion para comprobar si hay elementos repetidos en nuestro path
;;;
;;; INPUT: camino-recorrido: nuestro path
;;; OUTPUT: True si hay elementos repetidos en el path (camino-recorrido),
;;; NIL (False) si no
;;;

(defun hay-elementos-repetidos (camino-recorrido)
  (or (null camino-recorrido)
      (and (not (member (first camino-recorrido) (rest camino-recorrido)))
           (hay-elementos-repetidos (rest camino-recorrido)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; new-paths-improved (path node net)
;;; Version mejorada de la funcion new-paths
;;;
;;; INPUT: path: lista de caminos actual
;;; node: nodo del cual queremos ver todos los caminos que salen de el
;;; net: lista de listas de adyacencia del grafo que queremos explorar
;;; OUTPUT: lista de caminos que salen de nuestro nodo
;;;

(defun new-paths-improved (path node net)
    (if (null (hay-elementos-repetidos path))
        NIL
        (mapcar #'(lambda(n)
                (cons  n  path))
                (rest (assoc node net)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; bfs-improved (end queue net)
;;; Breadth-first-search in graphs improved
;;;
;;; INPUT: end: nodo meta
;;; queue: lista con los caminos a explorar proximamente y en orden como sublistas
;;; net: lista de listas de adyacencia del grafo que queremos explorar
;;; OUTPUT: lista de nodos que forman el camino optimo entre el elemento que se le pasa
;;; en queue y end si bfs tiene exito; NIL si no hay camino
```

```
;;;

(1)(defun bfs-improved (end queue net)
(2)    (if (null queue) '() ;si la cola que nos pasan esta vacia no hay camino
(3)        (let* ((path (first queue))
(4)               (node (first path)))
(5)             (if (eql node end)
(6)                (reverse path)
(7)               (bfs-improved end
(8)                (append (rest queue)
(9)                        (new-paths-improved path node net))
(10)                net)))))

(3) definimos el primer elemento de la cola como path, empezare a explorar por esa rama

(4) empiezo a explorar el primer nodo de la rama, node

(5) si el nodo que estoy explorando es el nodo meta, ya he acabado

(6) en tal caso, devuelvo el camino que he hecho (lo volteo primero, pues estaba al
reves por la logica del programa)

(7) llamo recursivamente a la funcion para que siga explorando en anchura

(8) concateno al final de mi cola el resto de caminos que tengo que explorar
(al final para hacer bfs y no dfs)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; shortest-path-improved (start end net)
;;;
;;; Version mejorada de la funcion shortest-path
;;;
;;; INPUT: start: nodo de partida
;;; end: nodo meta
;;; net: lista de listas de adyacencia del grafo que queremos explorar
;;; OUTPUT: lista con el camino entre start y end si existe; NIL si no existe tal camino
;;;

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

(setf grafofalla '((a b c) (b a c) (c a b) (d b)))
Grafo con bucle entre a, b y c y un enlace dirigido de d a b

(shortest-path-improved 'd 'a grafofalla)
Es posible llegar de D a cualquier otro nodo, pero no al reves
Devuelve (D B A), se puede solucionar

(shortest-path-improved 'a 'd grafofalla)
No es posible ir de A a D

(enlace D->B dirigido y entre A, B, C no dirigidos y bucle)
Devuelve NIL, no se ha podido encontrar camino