Práctica 1 Inteligencia Artificial

Lucía Asencio y Juan Riera

Febrero 2018

1 Ejercicio 1: Similitud Coseno

1.1

PSEUDOCÓDIGO

Entrada: x, y: vectores, representados como listas
Salida: similitud coseno entre los vectores
Procesamiento:
Se calculan los productos escalares de los operandos de la siguiente manera:
– Caso recursivo:
1) Si cualquiera de los dos vectores es NIL se devuelve 0
2) Si no, se devuelve el resultado de multiplicar los primeros elementos de los dos vectores y sumarlo con el siguiente nivel de recursión.
– Caso de mapcar:
1) Se aplica mapcar con la multiplicación entre los vectores 2) Se aplica reduce a todo el vector con la operacion dela suma Por último calculamos el coseno del ángulo con la fórmula

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; prodEscRec (x y)
;;; Calcula el producto escalar de dos vectores de forma recursiva.
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: producto escalar
;;;
(defun prodEscRec (x y)
    (if (or (null x) (null y))
        0
      (+ (* (first x) (first y)) (prodEscRec (cdr x) (cdr y)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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 vacias 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)
   ;;Falta comprobar la division por 0
    (/ (prodEscRec x y) (sqrt (* (prodEscRec x x) (prodEscRec y y)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; prodEscMapcar (x y)
;;; Calcula el producto escalar de dos vectores usando mapcar
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: producto escalar
;;;

(defun prodEscMapcar (x y)
    (reduce #'+ (mapcar #'* x y)))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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 vacias 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)
   ;; Falta comprobar dvision por 0
    (/ (prodEscMapcar x y) (sqrt (* (prodEscMapcar x x) (prodEscMapcar y y)))))
```

1.2

PSEUDOCÓDIGO

Entrada: Nivel de confianza, vector de categorías y vector de vectores
Salida: Vector de vectores cuyasimilitud con la categoria supera el nivel de confianza ordenados
Procesamiento: 1) Se ordenan los vectores con sort 2) Se seleccionan los que superan el umbral

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; select (conf cat vs)
;;; Selecciona, de entre unos vectores ya ordenados (vs)
;;; los que superan el umbral de similaridad
;;;
;;; INPUT: conf: Nivel de confianza
;;; cat: vector que representa a una categoria, representado como una lista
;;; vs: vector de vectores
;;; OUTPUT: Vectores cuya similitud con respecto a la categoria es superior al
;;; nivel de confianza, ordenados

(defun select (conf cat vs)
   (if (>= (sc-rec (car vs) cat) conf)
      vs
      (select conf cat (cdr vs))))

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



(defun sc-conf (cat vs conf)
   (select conf cat (sort (copy-list vs)
      #'(lambda (x y) (< (sc-rec x cat) (sc-rec y cat))))))
```

1.3

PSEUDOCÓDIGO

Entrada: Vector de categorías, vector de textos y funcion con la que calcular la similaridad.

Salida: pares compuestos por el identificador de la categoría y el resultado de similitud coseno

Procesamiento:

Para cada texto se busca la categoría de la siguiente manera: 1) Se forma la tupla con el identificador de la categoria y el resultado de la funcion coseno. 2) Se continúa la recursión, iterando sobre todas las categorías. 3) Si se llega al final (NIL) se devuelve una tupla con valor coseno inferior al mínimo dela función, de forma que siempre será superada a menos que sea la última, en cuyo caso esta tupla será un indicativo de error. 4) Tras la recursión la funcion devuelve la tupla con "second" máximo, el suyo propio, o el devuelto por la recursión. De esta manera el devuelto al final es el máximo de todos.

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; maxcadr (x y)
;;; Compara los segundos elementos de los vectores, y devuelve el
;;; que tenga el maximo.
;;;
;;; INPUT: x, y: los vectores a comparar
;;; OUTPUT: vector con el mayor "second"
;;;

(defun maxcadr (x y)
   (if(> (second x) (second y)) x y))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; classify (cats text func)
;;; Halla la categoria que se ajusta mas al texto text.
;;;
;;; INPUT: cats: vector de categorias
;;; text: texto para el que se quiere averiguar categoria
;;; func: funcion con la que calcular la similaridad
;;; OUTPUT: par formado por la categoria que mas se ajusta
;;; al texto y por el nivel de similaridad categoria-texto
;;;

(defun classify (cats text func)
   (if (null cats) '(-2 0)
      (maxcadr (list (caar cats) (funcall func (cdr text) (cdar cats)))
           (classify (cdr cats) text func))))
```

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

(defun sc-classifier (cats texts func)
   (if (null texts) nil
      (cons (classify cats (car texts) func)
         (sc-classifier cats (cdr texts) func))))
```

COMENTARIOS

- Tras haber medido tiempos de ejecución hemos obtenido los siguientes resultados:
  - Ambos algoritmos llegan al mismo resultado correcto.
  - La versión recursiva del algoritmo es del orden de 6 veces mas rápida que la versión de mapcar.

2 Ejercicio 2: Raíces de una función

2.1

PSEUDOCÓDIGO

Entrada: f, función y [a, b] intervalo en que buscar raíces
Salida: raíz encontrada
Procesamiento:
Si f(a)f(b) > 0, no se cumplen hipótesis
Si f(a)=0, a es raiz
Si f(b)=0, b es raiz
Si (b-a) < tol, punto-medio es raiz
Si f(a)f(puntomedio) < 0, busca raiz en [a, puntomedio]
En otro caso, busca raiz en [puntomedio, b]

CÓDIGO

```
;; Finds a root of f between the points a and b using bisection.
;;
;; If f(a)f(b)>=0 there is no guarantee that there will be a root in the
;; interval, and the function will return NIL.
;; INPUT:
```

```
;; f: function of a single real parameter with real values whose root we want to find
;; a: lower extremum of the interval in which we search for the root
;; b: b>a upper extremum of the interval in which we search for the root
;; tol: tolerance for the stopping criterion: if b-a < tol the function returns
;; (a+b)/2 as a solution.
;; OUTPUT: Root of the function

(defun bisect (f a b tol)
   (let ((medio (/ (+ a b) 2)))
       (cond ((< 0 (* (funcall f a) (funcall f b))) NIL )
            ((= 0 (funcall f a)) a)
            ((= 0 (funcall f b)) b)
            ((> tol (- b a)) medio)
            ((>= 0 (* (funcall f a) (funcall f medio))) (bisect f a medio tol))
            ((>= 0 (* (funcall f b) (funcall f medio))) (bisect f medio b tol)))))
;; Ejemplos
;; (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) --> 0.0
```

COMENTARIOS

- En un ejemplo del enunciado, en caso de que la raíz fuera uno de los extremos del intervalo, no se detectaba como raíz. Nos pareció más lógico encontrar las raíces en [a, b] y no es (a, b) y así lo hemos implementado.

2.2

PSEUDOCÓDIGO

Entrada: f, función; l, lista ordenada, tol para la condición de parada del algoritmo
Salida: lista con raíces encontradas
Procesamiento:
Si la lista no esta vacía,
1) Busca raíz entre los 2 1ºs eltos de la lista
2) Evalúa en el resto de la lista
3) Concatena resultados de 1) y 2)

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;; EJERCICIO 2.2 ;;;;;;;;;;;;;;;;;;

;; Usamos esta funcion auxiliar, que se comporta como bisect pero
;; devuelve las raices en una lista, para evitar tener que llamar dos veces
;; a la funcion bisect con cada iteracion de mapcan.

(defun bisect-aux (f a b tol)
  (unless (or (null a) (null b))
```

```
    (let ((medio (/ (+ a b) 2)))
        (cond ((< 0 (* (funcall f a) (funcall f b))) NIL )
              ((= 0 (funcall f a)) (list a))
              ((= 0 (funcall f b)) (list b))
              ((> tol (- b a)) (list medio))
              ((>= 0 (* (funcall f a) (funcall f medio))) (bisect-aux f a medio tol))
              ((>= 0 (* (funcall f b) (funcall f medio))) (bisect-aux f medio b tol))))))

;; Finds all the roots that are located between consecutive values of a list
;; of values
;;
;; INPUT:
;; f: function of a single real parameter with real values whose root
;; we want to find
;; lst: ordered list of real values (lst[i] < lst[i+1])
;; tol: tolerance for the stopping criterion: if b-a < tol the function
;; returns (a+b)/2 as a solution.
;;
;; Whenever sgn(f(lst[i])) != sgn(f(lst[i+1])) this function looks for a
;; root in the corresponding interval.
;;
;; OUTPUT:
;; A list o real values containing the roots of the function in the
;; given sub-intervals

(defun allroot (f lst tol)
  (unless (null (rest lst))
    (mapcan #'(lambda(x y) (bisect-aux f x y tol))
            lst
            (rest lst))))

(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)
;; (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)
;; (allroot #'(lambda(x) (sin (* 6.28 x))) '() 0.0001) --> NIL
;;  (allroot #'(lambda(x) (sin (* 6.28 x))) '(0.1 0.2 0.3) 0.0001) --> NIL
```

COMENTARIOS

- Usar mapcan con la función original provoca que haya NIL's en la lista resultado, y cualquier comprobación del estilo "Si bisect(f)!= NIL, (list bisect(f))" para eliminar los NILs tenía como consecuencia una segunda e innecesaria ejecución de la función bisect. Lo mismo ocurría con let (según hemos entendido, el let provocaría también esta doble ejecución).Por tanto, hemos hecho uso de una función auxiliar cuya funcionalidad es la misma que la del bisect original pero que en caso de devolver una raíz, la devuelve dentro de una lista para que mapcan pueda concatenarla.

2.3

PSEUDOCÓDIGO

Entrada: f función, [a, b] intervalo donde buscar raíces, N 2^N es el tamaño def la partición, tol define la condición de parada del algoritmo
Salida: lista con las raíces encontradas
Procesamiento:
Actual = a,
Mientras actual <= b,
1) Busca raices en [actual, actual + tamintervalo]
2) Evalua en actual + tamintervalo
3) Concatena resultado de 1) y 2)

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;; EJERCICIO 2.3 ;;;;;;;;;;;;;;;;;;

;; Funcion auxiliar
;; Busca raices en intervalos de tamanio incr en (actual, fin)
;;

(defun introot (f incr fin actual tol)
   (unless (<= fin actual)
       (union (bisect-aux f actual (+ actual incr) tol)
            (introot f incr fin (+ actual incr) tol))))


;; Divides an interval up to a specified length and find all the roots of
;; the function f in the intervals thus obtained.
;; INPUT:
;; f: function of a single real parameter with real values whose root we want to find
;; a: lower extremum of the interval in which we search for the root
;; b: b>a upper extremum of the interval in which we search for the root
;; N: Exponent of the number of intervals in which [a,b] is to be divided:
;; [a,b] is divided into 2^N intervals
;; tol: tolerance for the stopping criterion: if b-a < tol the function
;; returns (a+b)/2 as a solution.
;;
;; The interval (a,b) is divided in intervals (x[i], x[i+i]) with
;; x[i]= a + i*dlt; a root is sought in each interval, and all the roots
;; thus found are assembled into a list that is returned.
;;
;; OUTPUT: List with all the found roots.

(defun allind (f a b N tol)
   (introot f (/ (- b a) (expt 2 N)) b a tol))

;; EXAMPLES
```

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

;; (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 x)) 0 16 4 0.0001)
;; --> (0 102943/32768 205887/32768 308831/32768 411775/32768 514719/32768)
```

COMENTARIOS

- Hemos encontrado un error en la función que no nos da tiempo a resolver. Como la función encuentra raíces en intervalos de la forma $[a, a+i], [a+i, a+2i], ... [a+ki, a+(k+1)i]...$ en caso de encontrarse una raíz en un extremo de intervalo, ésta aparece repetida en el resultado. Podemos resolver esto eliminando repetidos o, de manera más eficiente y menos correcta (perderíamos una raíz en caso de ser f(b) = 0) modificar la función bisect para que encontrara raíces en $[a+ki, a+(k+1)i)$

3 Ejercicio 3: Combinación de listas

3.1

PSEUDOCÓDIGO

```
Entrada:  elt elemento que combinar con lst lista
Salida:  lista con la combinacion
Procesamiento:
Si elt != null and lst != null,
 1) Combina elt con first(lst)
 2) Evalua en resto de lst
 3) Concatena resultados de 1) y 2)
```

CÓDIGO

```
;; Combina un elto dado con todos los eltos de una lista
;;
;; INPUT:
;; elt: elemento a combinar con la lista
;; lst: llista quue sera combinada con el elemnto
;;
;; OUTPUT: lista que contenga la combinacion

(defun combine-elt-lst (elt lst)
  (unless (or (null lst) (null elt))
    (cons (list elt (first lst))
        (combine-elt-lst elt (rest lst)) )))
```

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

3.2

PSEUDOCÓDIGO

Entrada: lst1, lst2 listas a combinar
Salida: lista con producto cartesiano
Procesamiento:
Si lst1 != null and lst2 != null,
 1) Combina first(lst1) con lst2
 2) Evalua con resto de lst1 y lst2
 3) Concatena resultados de 1) y 2)

CÓDIGO

```
;; Funcion que devuelve el producto cartesiano de dos listas
;;
;; INPUT:
;; lst1, lst2 : listas a combinar
;;
;; OUTPUT: lista que contenga la combinacion

(defun combine-lst-lst (lst1 lst2)
   (unless (or (null lst1) (null lst2))
      (append (combine-elt-lst (first lst1) lst2)
          (combine-lst-lst (rest lst1) lst2))))

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

COMENTARIOS

- (Ningún comentario hasta ahora)

3.3

PSEUDOCÓDIGO

Entrada: lstolsts
Salida: lista que contenga todas las combinaciones
Procesamiento:
Función principal
1) Si la lista es nil, hemos llegado al final de la recursión y devolvemos una lista que contenga nil.
2) En el caso general, se combina el primer elemento de la lista de listas recibida como argunemto con el resultado de continuar la recurión con el resto de la lista. Dicho resultado es una lista de listas. Para realizar esta combinación utilizamos la función combine-list-lsts, que simplemente añade cada elemento de la lista pasada como primer argunemto a cada una de las listas recibidas en la lista de listas pasada como segundo argumento. El resultado es una lista de listas con todas las combinaciones entre elementos de la primera y listas de la segunda.

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;; EJERCICIO 3.3 ;;;;;;;;;;;;;;;;;;

;; Devuelve una lista resultado de aniadir al principio de todas
;; las listas contenidas en lst el elemnto elt
;;
;; INPUT:
;; elt : elemento a aniadira todas las listas
;; lst : lista de listas a las que aniadir el elemento
;;
;; OUTPUT: lista que contenga todas las combinaciones

(defun append-elt-lst (elt lst)
   (if (null lst) (list (list elt))
       (if (null (cdr lst))
          (list (append (list elt) (first lst)))
          (cons (append (list elt) (first lst))
            (append-elt-lst elt (rest lst)) ))))



;; Recibe una lista A y una lista de listas B y aniade cada elemento
;; de A a cada una de las listas de B de todas las maneras posibles
;; un elemento de cada lista
;;
;; INPUT:
;; lst : lista de la que tomar elementos
;; lsts: lista de listas a las que aniadir los elementos de lst
;;
```

```
;; OUTPUT: lista que contenga todas las combinaciones


(defun combine-list-lsts (lst lsts)
    (unless (null lsts)
    (if (null (cdr lst))
        (append-elt-lst (car lst) lsts)
        (append (append-elt-lst (car lst) lsts) (combine-list-lsts (cdr lst) lsts)))))


;; 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
;;
;; INPUT:
;; lstolsts : listas a combinar
;;
;; OUTPUT: lista que contenga todas las combinaciones


(defun combine-list-of-lsts (lstolsts)

    (if (null lstolsts)
        '(NIL)
        (unless (null (car lstolsts))
        (combine-list-lsts (car lstolsts) (combine-list-of-lsts (cdr lstolsts))))))
```

COMENTARIOS

- Algunas de las funciones auxiliares utilizadas en este ejercicio se podrían sustituir por funciones ya implementadas en lisp, pero consideramos más didáctico crearlas nosotros.

  Esta función utliza dos funciones auxiliares:
  - combine-list-lsts añade cada uno de los elementos de la primera lista a cada una de las listas del segundo argumento. - append-elt-lst añade al principio de cada lista de la lista de listas pasada como segundo argumento el elemento pasado como primer argumento

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 Determinación de si una exresión es un literal positivo

PSEUDOCÓDIGO

Entrada: Cualquier objeto lisp
Salida: T si ese objeto es un literal positivo o NIL en caso contrario
Procesamiento:
Comprobamos que el argumento de entrada es un átomo, pero que no es ni un conector ni un valor de verdad. En ese caso devolvemos true y devolvemos NIL en caso contrario.

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.1
;; Predicado para determinar si una expresion en LISP
;; es un literal positivo
;;
;; RECIBE  : expresion
;; EVALUA A : T si la expresion es un literal positivo,
;;           NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun positive-literal-p (x)
   ;; Tiene que cumplirse que es un atomo lisp, y que no es ni un
   ;; conector ni un valor deverdad
   (and (atom x) (not (connector-p x)) (not (truth-value-p x))))
```

4.1.2 Determinación de si una expresión es un literal negativo

PSEUDOCÓDIGO

Entrada: Cualquier objeto lisp
Salida: T si la entrada era un literal negativo y NIL en caso contrario.
Procesamiento:
1. Comprobamos que el argumento de entrada es una lista
2. Si lo es, comprobamos que tiene como únicos elementos un símbolo de negación y un literal positivo

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.2
```

```
;; Predicado para determinar si una expresion
;; es un literal negativo
;;
;; RECIBE  : expresion x
;; EVALUA A : T si la expresion es un literal negativo,
;;           NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun negative-literal-p (x)
    ;; Tiene que ser una lista, compuesta por:
    (if (listp x) (and (eql (car x) +not+)  ;; Un simbolo de negacion
         (positive-literal-p (cadr x))  ;; un literal positivo
                  (null (cddr x)))      ;; y nada mas
     NIL))
```

COMENTARIOS

4.1.3 Determinación de si un una expresión es un literal

PSEUDOCÓDIGO

Entrada: Cualquier objeto de lisp
Salida: T si es un literal, NIL si no lo es
Procesamiento:
Comprobamos si es o un literal positivo o un literal negativo mediante un 'or'.

CÓDIGO

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

4.1.4 Determinación de si una expresión está en formato infijo

PSEUDOCÓDIGO

Entrada: Una expresión
Salida: T si la expresión de entrada estaba en formato infijo, y NIL en caso contrario.
Procesamiento:

1. Comprobamos que no recibimos un valor de verdad (T o NIL)
2. Y comprobamos que o es un literal, o es una lista. Si es un literal, devolvemos T.
3. En caso de que sea una lista, deducimos que el argumento es una expresión, y por tanto miramos el primer elemento.
4. Si el primer elemento es un conector unario, comprobamos que el segundo elemento es una expresión infijo, y devolvemos T en tal caso.
5. Si el primer elemento es un conector n-ario, tiene que aparecer solo, ya que en caso contrario, al ser una expresión infijo, la expresion sería del tipo
   (A connector)
   Donde A es una expresion infijo y conector el conector n-ario en cuestión. Comprobamos que se cumple este hecho y devolvemos T si se cumple.
6. Si el primer elemento es un literal, miramos si va sucedido de un connector binario o n-ario, en el primer caso, comprobamos si éste a su vez va sucedido de un literal o una expresión infijo, y devolvemos T en tal caso y NIL en el contrario.
7. Si en lugar de un conector binario, el conector es n-ario, comprobamos que se cumple o que la expresión consta de tres elementos (literal, conector, expresión infija), o que si quitáramos los primeros dos elementos de la lista, la expresión resultante sería una expresión infijo (por ejemplo, en el caso (a v b v c) si quitamos 'a v' queda (b v c) que es una expresión infijo, luego (a v b v c) lo es.
8. Si el primer el elemento que miramos en el paso 3 es una lista, comprobamos que es una expresión infijo, y continuamos desde el paso 6 tratándola como si fuera un literal.

CÓDIGO

```
(defun wff-infix-p (x)
  (unless (truth-value-p x)    ;; Por convenio los valores de verdad no
                               ;; son expresiones infijo

    (or (literal-p x)        ;; Un literal es FBF en formato infijo
        (and (listp x)       ;; En caso de que no sea un literal debe ser una lista

          (cond

            ((unary-connector-p (first x))
               ;; Si el primer elemento es un conector unario (negacion)
               ;; tiene que estar sucedido por un unico elemento,
               ;; es decir, (cddr x) tiene que ser nil y el segundo
               ;; elemento tiene que ser una expresion infijo o un literal
```

```
    (unless (not (null (cddr x)))
        (wff-infix-p (second x))))

     ;; Si es un conector n-ario y aparece antes que ningun
     ;; otro literal, tiene que ser el unico elemento de la lista,
     ;; ya que si hubiera un literal en la expresion tendria que ir
     ;; antes del conector al ser expresion infijo.
((n-ary-connector-p (first x))

    (null (cdr x)))

    ;; Si aparece un literal, debe de estar sucedido por un operador
    ;; binario o n-ario, que a su vez deben estar sucedidos por
    ;; una o mas expresiones infijo o literales segun corresponda
((literal-p (first x))
    (or
        (if (binary-connector-p (second x))
            (and (null (cdddr x)) (wff-infix-p (third x))))
        (if (n-ary-connector-p (second x))
            (or (and (null (fourth x)) (wff-infix-p (third x)))
                (and (eql (fourth x) (second x)) (wff-infix-p (cddr x)))))))

;; Si el primer elemento es una lista debe ser una expresion infijo
;; y debe estar sucedida por un conector n-ario o binario como en
;; el caso anterior, que a su vez debe estar sucedido por un literal
;; o expresion infijo
((listp (first x))
       (and (wff-infix-p (car x))
              (or
                  (if (binary-connector-p (second x))
                      (and (null (cdddr x)) (wff-infix-p (third x))))
                  (if (n-ary-connector-p (second x))
                      (or (and (null (fourth x)) (wff-infix-p (third x)))
                          (and (eql (fourth x) (second x)) (wff-infix-p (cddr
                              x)))))))))))))
```

4.1.5 Conversión de infijo a prefijo

PSEUDOCÓDIGO

Entrada: FBF infijo
Salida: FBF prefija
Procesamiento:
1. Si la FBF es literal o un conector n-ario: devuelve la FBF
2. Si la FBF es un $\neg K$): devuelve ($\neg$ infijo(K))
3. Si la FBF es un (K_1 binary-conn K_2): devuelve (infijo(K_1) binary-conn infijo(K_2))
4. Si la FBF es un (K_1 n-ary K_2 n-ary ... K_n): devuelve (n-ary infijo(K_1)...infijo(K_n)

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 (or (literal-p wff) (n-ary-connector-p wff)) ; '(v), '(~ a), 'a
      wff
      (let ((op1 (first wff)) ;op1 puede ser un operando o un ~
            (op2 (second wff));op2 puede ser un connectr o un operando
            (op3 (third wff)))
        (cond
          ((unary-connector-p op1) ; caso (~wff) -> (~(inf-pre wff))
           (list op1 (infix-to-prefix op2)))
          ((binary-connector-p op2) ; caso (wff1 <=> wff2) -> (<=> (inf-pre wff1) (inf-pre wff2))
           (list op2 (infix-to-prefix op1) (infix-to-prefix op3)))
          ((n-ary-connector-p op2) ; caso (wff1 ^ ... ^ wff n) -> (^ (inf-pre wff1) ... (inf-pre wffn))
           (cons op2 (mapcan #'(lambda(x) (unless (connector-p x) (list (infix-to-prefix x)))) wff))
              ))))))
;;
;; 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 Determinación de si una FBF en formato prefijo es una cláusula

PSEUDOCÓDIGO

Entrada: FBF en formato prefijo
Salida: Evaluacion de si esta FBF es una cláusula.
Procesamiento: 1. Comprobamos que la entrada es una lista.
2. Comprobamos que su primer elemento es un símbolo or.
3. Comprobamos que ese elemento va sucedido de literales o de nada, ya que () es una expresión válida. Esta última comprobación se hace con una función auxiliar que funciona de forma recursiva.

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.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;
;; Funcion auxiliar que devuelve true si recibe como
;; argumento una lista de literales o nil
;;;;
(defun list-of-literals-or-nil (lst)
    (or (null lst) ;; O el argumento de entrada es un NIL
    ;; Comprobamos que el elemento actual es una lista y continuamos la recursion
        (and (literal-p (first lst)) (list-of-literals-or-nil (rest lst)))))
```

```
;;;;
;; Funcion principal
;;;;
(defun clause-p (wff)
    (and (listp wff)                    ;; Recibe una lista
         (eql (first wff) +or+)    ;; cuyo primer elemento es un or
           ;; que va sucedido de una lista de literales o de nada
           ;; por ejemplo (v a b) es una clausula, pero (v) tambien
               (list-of-literals-or-nil (rest wff))))
```

COMENTARIOS

- La función auxiliar list-of-literals-or-nil devuelve t si elargumento que recibe es una lista e literales o nil.

4.1.7 Determinación de si una expresión está en FNC

PSEUDOCÓDIGO

Entrada: FBF en formato prefijo
Salida: T si la fbf está en fnc, y NIL si no
Procesamiento: 1. Comprobamos que el argumento es una lista.
2. Comprobamos que el primer elemento es un and.
3. Comprobamos que está sucedido de una lista de cláusulas o de nada, como ocurre en ().
Si cualquiera de estas comprobaciones no se cumpliera, devolvemos nil. Para comprobar el paso 3 utilizamos una funcion auxiliar con el siguiente pseudocódigo:
1. Si la lista no tiene elementos, devuelvo T. 2. Si no, compruebo que el primer elemento es una cláusula. 3. Si lo es, vuelvo al paso uno, pero en vez de con la lista completa, con la lista sin el primer elmento. Si no lo es devuelvo NIL.

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 1.7
;; Predicado para determinar si una FBF esta en FNC
;;
;; RECIBE  : FBF en formato prefijo
;; EVALUA A : T si FBF esta en FNC con conectores,
;;           NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;; Funcion auxiliar que devuelve true en caso de recibir
;; como argumento una lista de clausulas o nil
;;;;
(defun list-of-clauses-or-nil-p (lst)
   (or (null lst)
       (and (clause-p (first lst))
```

```
          (list-of-clauses-or-nil-p (rest lst)))))

(defun cnf-p (wff)
  (and (listp wff)                ;; Tiene que ser una lista
       (eql (first wff) +and+)      ;; cuyo primer elemento tiene que ser
                                   ;; un and
          ;; que tiene que estar sucedido de una lista de clausulas
          ;; o de nada (como ocurre en (^), que es una cnf)
       (list-of-clauses-or-nil-p (rest wff))))
```

COMENTARIOS

- Se ha uilizado una funciona auxiliar que devuelve true si recibe como argumento una lista de cláusulas o nil.

4.2 Algoritmo de transformación de una FBF a FNC

4.2.1 Eliminación del bicondicional

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.1: Incluya comentarios en el codigo adjunto
;;
;; Dada una FBF, evalua a una FBF equivalente
;; que no contiene el connector <=>
;;
;; RECIBE  : FBF en formato prefijo
;; EVALUA A : FBF equivalente en formato prefijo
;;           sin connector <=>
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun eliminate-biconditional (wff)
  (if (or (null wff) (literal-p wff)) ;; Si recibe como argumento nill
      wff                             ;; o un literallo devuelve

    (let ((connector (first wff)))           ;; Si no, mira si el conector
      (if (eq connector +bicond+)            ;; es un bicondicional
          (let ((wff1 (eliminate-biconditional (second wff)))
                (wff2 (eliminate-biconditional (third wff))))
                ;; Si lo es, elimina el bicondicional en las dos
                ;; expresiones que rodean al propio bicondicional
                ;; y despues aplica la transformacion.
            (list +and+
                  (list +cond+ wff1 wff2)
                  (list +cond+ wff2 wff1)))
                  ;; Si no es ninguno de los casos anteriores,
                  ;; el primer elemento debe ser un operador de otro
```

```
            ;; tipo y aplicamos la eliminacion del bicondicional
            ;; en todos sus operandos
      (cons connector
            (mapcar #'eliminate-biconditional (rest wff)))))))
```

4.2.2 Eliminación del condicional

PSEUDOCÓDIGO

Entrada: una expresion en formato prefijo sin el conector <=>
Salida: una expresión equivalente a la de entrada, pero sin conector =>
Procesamiento:

1. Comprobamos si el argumento de entrada es nil o un literal y, si es así, lo devolvemos tal cual.
2. En caso contrario, debemos de estar ante una expresión, así que comprobamos el conector, que es el primer elemento.
3. Si éste es un condicional, aplicamos la eliminación de condicional (recursivamente) a los dos operandos, y procedemos a la transformación (a => b) -> ((+not+ a) v b)
4. Si es cualquier otro tipo de operador, aplicamos la eliminación del condicional a todos sus operandos

CÓDIGO

```
(defun eliminate-conditional (wff)
  (if (or (null wff) (literal-p wff)) ;; Si el argumento es un literal
      wff                             ;; o nil devolvemos el argumento
    (let ((connector (first wff)))
       ;; En cualquier otro caso, estamos ante una expresion.
       ;; Comprobamos el conector.
      (if (eq connector +cond+)
          (let ((wff1 (eliminate-conditional (second wff)))
                (wff2 (eliminate-conditional (third wff))))
                ;; Si es un condiconal, tras aplicar la
                ;; eliminacion del condicional a los dos operandos,
                ;; procedemos a la transformacion:
                ;; (a => b) -> ((+not+ a) v b)
            (list +or+
                  (list +not+ wff1)
                  wff2))

                 ;; Si no, es un operador de otro tipo
                 ;; Aplicamos la eliminacion del condicional
                 ;; a todos los operandos
          (cons connector
                (mapcar #'eliminate-conditional (rest wff)))))))
```

4.2.3 Reducción del ámbito de la negación

PSEUDOCÓDIGO

Entrada: FBF en formato prefijo sin los conectores <=> ni =>
Salida: FBF equivalente en formato prefijo en la que la negación aparece únicamente en literales negativos
Procesamiento:
1. Comprobamos si recibimos un literal, en tal caso lo devovemos tal cual.
2. En cualquier otro caso debe ser una expresión, y por tanto una lista, miramos los primeros dos elementos de dicha lista.
3. Si el primer elemento es una negación, devolvemos el resultado de intentar aplicar las leyes de De Morgan sobre esa expresión. Esto quiere decir lo siguiente (pseudo-código dela funcion 'morgan'):
   3.1. Si recibimos algo que no sea una lista, devolvemos una lista que contenga el símbolo de negacion y el elemento recibido como argumento.
   3.2 Si es una lista, intentamos negarla, ya sea devolviendo su equivalente positivo si es un literal negativo, o aplicando las leyes de De Morgan si es una expresión con un conector n-ario.
4. Si por el contrario es un conector n-ario reducimos el rango de la negación (recursivamente) en todos sus operandos.

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


;; Funcion auxiliar que aplica las leyes de De Morgan a una expresion
;; que se quiere negar. Por ejemplo:
;; '(+not+ (v a b)) = (morgan '(v a b)) = (^ (+not+ a) (+not b))
;; Ademas esta funcion intentara reducir el alcance de la negacion
;; en todas las subexpresiones que reciba

(defun morgan (wff)
   (if (listp wff)                  ;; Si es una lista
       (cond ((eql (first wff) +not+) ;; y es una expresion negada
                  ;; Se intenta reducir el alcance de la negacion
                  ;; y se devuelve el resultado
              (reduce-scope-of-negation (second wff)))
```

```
          ((n-ary-connector-p (first wff)) ;; Si es un operador n-ario
              (cons (exchange-and-or (first wff)) ;; intercambiamos and y or
                    (negar-lista (rest wff))))) ;; y negamos todos los
                                                ;; elementos de la lista

       (cons +not+ wff))) ;; Si no es una expresion, debe ser un literal,
                          ;; lo negamos y lo devolvemos


;;;;
;; Funcion auxiliar que niega todos los elementos de la lista
;;;;
(defun negar-lista (wff)
   (unless (null wff)
       (cons (negar (first wff)) (negar-lista (rest wff)))))


;;;;
;; Funcion auxiliar que niega el elemento que reciba.
;;;;
(defun negar (elt)
          ;; Si es un literal negativo lo convierte a positivo
   (cond ((negative-literal-p elt) (second elt))
          ;; Si es un literal positivo lo convierte a negativo
          ((positive-literal-p elt) (list +not+ elt))
          ;; Si es una expresion intentara negarla aplicando las leyes
          ;; de De Morgan haciendo un llamada a la funcion morgan
          ((listp elt) (morgan elt))))

;;;;
;; Funcion auxiliar que recibe una lista de expresiones
;; e intenta reducir el rango de la negacion en todas ellas.
;; Su utilidad se ve mas clara al leer el codigo y comentarios
;; de la funcion principal reduce-scope-of-negation
;;;;
(defun redScopeNeg-n-ary (wff)
   (unless (null wff)
       (cons (reduce-scope-of-negation (first wff))
             (redScopeNeg-n-ary (rest wff)))))
;;;;
;; Funcion principal
;;;;
(defun reduce-scope-of-negation (wff)
   (if (literal-p wff)         ;; Si recibe un literal
       wff                     ;; lo devuelve
       (let    ((firstEl (first wff))
              (restEl (rest wff))
              (secondEl (second wff)))
```

```
        (cond              ;; Si no es un literal
                           ;; es una expresion

            ((eql firstEl +not+)  ;; Si es una expresion negada
                (morgan secondEl))  ;; aplicamos las leyes de De Morgan

                ;; Por otro lado, si es un conector n-ario, intento
                ;; reducir el rango de la negacion en todos sus
                ;; operandos, haciendo uso de redScopeNeg-n-ary
            ((n-ary-connector-p firstEl)
                (if (null secondEl)
                    wff
                    (cons firstEl (redScopeNeg-n-ary restEl))))))))


;;;;
;; Funcion auxiliar que intercambia and por or y viceversa
;;;;
(defun exchange-and-or (connector)
  (cond
   ((eq connector +and+) +or+)
   ((eq connector +or+) +and+)
   (t connector)))
```

COMENTARIOS

- Esta función tiene muchas funciones auxiliares, cada una de la cuales tiene una funcion concreta y sencilla. Aquí explicamos brevemente qué hace cada una:
  - morgan: sirve para negar una expresión aplicando las leyes de De Morgan si corresponde, pero realmente sirve para negar cualquiertupo de expresión. Puede llamar a negar-lista.
  - negar-lista: recibe como argumento una lista, y niega todos sus elementos como corresponda, llamando a la función 'negar'
  - negar: niega el elemento que reciba como argumento, ya sea negándolo directamente por ser un literal negativo o positivo o aplicando las leyes de De Morgan llamando a 'morgan'
  - redScopeNeg-n-ary: recibe como argumento una lista de expresiones sobre las que llamar a la función reduce-scope-of-negation. Devuelve una lista de todos los resultados de forma similar a mapcar. Esta fucnión resulta útil cuando estamos ante una expresión que no hay que negar, y simplemente hay que reducir el rango de la negación en todos los operandos sin tocar el operador.

4.2.4 Traducción a FNC: comentar código

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.4: Comente el codigo adjunto
;;
;; Dada una FBF, que no contiene los conectores <=>, => en la
;; que la negacion aparece unicamente en literales negativos
```

```
;; evalua a una FBF 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)
  (if (null lst)
      ;; Si lst vacia, devuelve '((elt))
      (list (list elt))
    ;; Si no: ((elt x1) (elt x2) ... (elt xn)) para los xi en lst
    (mapcar #'(lambda (x) (cons elt x)) lst)))

; Dada una lista de listas de literales, devuelve una lista con todas las posibles listas que
; tienen un elemento de cada lista. En las listas internas pone como conector el conector
;externo, y en la lista externa pone como conector el opuesto del conector el externo,
(defun exchange-NF (nf)
  (if (or (null nf) (literal-p nf))
      nf ;; Si nf esta vacia o es un literal, no tenemos que hacer nada
    (let ((connector (first nf)))
      ;; Si no: cambia el conector externo por el su opuesto,
      (cons (exchange-and-or connector)
            (mapcar #'(lambda (x)
                        ;; y todos los internos por el externo,
                        (cons connector x))
               ;; en todas las posibles listas q contienen un elto de cada lista
               (exchange-NF-aux (rest nf)))))))

; Dada una lista de listas, devuelve todas las listas posibles
; resultado de combiar un elemento de cada lista
(defun exchange-NF-aux (nf)
  (if (null nf)
      NIL
    (let ((lst (first nf)))
      (mapcan #'(lambda (x)
                  ;; Combina cada x de la primera lista con todas las listas
                  ;; que salen de combinar entre si las demas listas
                  (combine-elt-lst
                   x
                   (exchange-NF-aux (rest nf))))
        ;; Si 1er elto literal, aplicamos al elto. Si es conector, aplicamos a lo demas.
        (if (literal-p lst) (list lst) (rest lst))))))

;; Dada una lista de sublistas conectadas entre si por un conector X, 'deshace' todas
;; sublistas conectadas con X, eliminando el conector y llevando todos los literales de la
;; sublista a la lista principal, i.e. (^ (v cosas) (^ cosas2)) --> (^ (v cosas) cosas2)
(defun simplify (connector lst-wffs )
  (if (literal-p lst-wffs)
```

```
      ;; Si la lista es un literal, hacemos (literal)
      lst-wffs
    (mapcan #'(lambda (x)
                (cond
                 ;; Si el elto de la sublista es literal, hacemos (literal)
                 ((literal-p x) (list x))
                 ;; Si la sublista esta conectada por el conector que simplificamos,
                 ;; aplicamos la simplificacion a todos sus elementos menos al conector
                 ((equal connector (first x))
                  (mapcan
                      #'(lambda (y) (simplify connector (list y)))
                    (rest x)))
                 ;; Si el conector no era el buscado, dejamos a la sublista tal cual
                 (t (list x))))
      ;mapcan concatena todas las listas anteriores
      lst-wffs)))

(defun cnf (wff)
  (cond
   ((cnf-p wff) wff) ;; Si ya es cnf, nada que hacer
   ((literal-p wff) ;; Si ya es un literal, (^ (v lit))
    (list +and+ (list +or+ wff)))
   ((let ((connector (first wff)))
      (cond
       ((equal +and+ connector)
        ;; Si es (^ () ()) : aplicamos cnf a todas las sublistas, simplificamos los ^
        ;; y aniadimos el ^ del principio
        (cons +and+ (simplify +and+ (mapcar #'cnf (rest wff)))))
       ((equal +or+ connector)
        ;; Si es (v () ()) : simplificamos los v, aniadimos el v del ppio y hacemos
        ;; el intercambio de los ^ por v
        (cnf (exchange-NF (cons +or+ (simplify +or+ (rest wff)))))))))))
```

4.2.5 Eliminación de conectores

PSEUDOCÓDIGO

Entrada: FBF en FNC con conectores and y or
Salida: FBF en FNC sin esos conectores, en forma de lista de listas
Procesamiento:
1. Miramos el primer elemento de la lista de entrada.
2. Si es un operador n-ario, vuelvo al paso 1 con la lista sin el primer elemento (rest argumento-de-entrada)
3. Si es una lista, es una expresión, y devolvemos una lista compuesta por el resultado de ejecutar esta funcion en el primer elemento seguido por la conotinuación de la recursión sobre el resto de la lista de entrada como en el paso 2 (este paso probablemente se entienda mejor en el código)
4. Si es cualquier otro elemento, como un literal positivo, devuelvo una lista compuesta por este primer elemento y el resultado de continuar la recursión como en el paso 2.

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)
   (unless (null cnf)
   (let ((firstEl (first cnf)))
   (cond ((n-ary-connector-p firstEl)  ;; Si el argumento de entrada
                                       ;; es una expresion n-aria
                                       ;; eliminamos conectores de todos
                                       ;; sus operandos
          (eliminate-connectors (rest cnf)))
      ((listp firstEl)                 ;; Si es una expresion elimino sus conectores
                                       ;; (este caso es por la recursion)
          (cons (eliminate-connectors firstEl)
                (eliminate-connectors (rest cnf))))
      (t (cons firstEl (eliminate-connectors (rest cnf))))))))
```

4.2.6 Conversión de infijo a fnc sin conectores

PSEUDOCÓDIGO

Entrada: expresión FBF
Salida: FBF en FNC equivalente sin conectores
Procesamiento:
1. Convertimos la expresión a prefijo
2. Eliminamos el bicondicional
3. Eliminamos el condicional
4. Reducimos el rango de la negación al máximo posible
5. Convertimos a FNC
6. Eliminamos los 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)
  ;; Aplicamos las funciones en el orden definido en el enunciado
  (eliminate-connectors
    (cnf
      (reduce-scope-of-negation
        (eliminate-conditional
          (eliminate-biconditional
            (infix-to-prefix wff)))))))
```

4.3 Simplificación de FBFs en FNC

4.3.1 Eliminación de literales repetidos

PSEUDOCÓDIGO

Entrada: Lista de literales
Salida: Devuelve una lista con los mismos literales pero sin repetición
Procesamiento:
1. A menos que el argumento de entrada sea nil, busco el primer elemento de la lista en el resto de la lista para comprobar si está repetido. 2. Si lo está devuelvo el resultado de continuar la recursión sobre el resto de la lista, deshechando el primer elemento. 3. Si no está, devuelvo una lista que contenga el primer elemento y el resultado de continuar la recursión sobre el resto de la lista.

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


;;;;
;; Funcion auxiliar que comprueba si dos literales son iguales
;;;;
(defun equal-literals (elt1 elt2)
    (or (eql elt1 elt2)
        (and (negative-literal-p elt1) (negative-literal-p elt2)
            (eql (second elt1) (second elt2)))))


;;;;
;; Funcion auxiliar que recibe una lista y un elemento.
;; Busca el elemento en la lista y devuelve t si lo encuentra.
;;;;
(defun search-literal-p (lst elt)
    (unless (null lst)
        (or (equal-literals (first lst) elt)
            (search-literal-p (rest lst) elt))))


(defun eliminate-repeated-literals (k)
    (unless (null k)
        (if (search-literal-p (rest k) (first k)) ;; Busco el primer
                                                  ;; elemento de la lista en el resto.
```

```
        (eliminate-repeated-literals (rest k)) ;; Si esta, continuo la
                                               ;; recursion sin aniadirlo.
                                               ;; Si no esta repetido, lo aniado
                                    ;; a la lista que devolvere al final.
        (cons (first k) (eliminate-repeated-literals (rest k))))))
```

COMENTARIOS

- Hemos utilizado dos funciones auxiliares:
  - equal-literals: devuelve t si recibe dos literales iguales, ya sean positivos o negativos.
  - search-literal-p: recorre recursivamente la lista que recibe como primer argumento, y comprueba si dentro de ella está el literal recibido como segundo argumento. Devuelve t en tal caso y nil en caso contrario

4.3.2 Eliminación de cláusulas repetidas

PSEUDOCÓDIGO

Entrada: FBF en FNC, lista de cláusulas con conjunción y disyunción implícitas
Salida: FNC equivalente sin clausulas repetidas
Procesamiento:
1. Eliminamos de forma recursiva los literales repetidos de cada una de las cláusulas.
2. Buscamos el primer elemento de la lista de clausulas en el resto
3. Si está continuamos la recusión sobre el resto de la lista y nos olvidamos del primer elemento
4. Si no está añadimos el primer elemento a la lista devuelta por la continuación de la recursión sobre el resto de nuestra lista de entrada.

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

;;
;; Funcion que elimina un elemento una sola vez de la lista
;; Ejemplo: (elim-elem '(a b a) 'a) -> (B A)
;;

(defun elim-elem (lst elem)
   (unless (null lst)
```

```
        (if (equal-literals (first lst) elem)
            (rest lst)
            (cons (first lst) (elim-elem (rest lst) elem)))))

;;
;; Funcion que determina si dos clausulas son iguales,
;; es decir, si tienen los mismos literales repetidos el
;; mismo numero de veces
;;

(defun equal-clauses (cl1 cl2)
    (if (null cl1)    ;; Si una es nula (o si hemos llegado
                      ;; al final de la recursion)
        (null cl2)    ;; la otra tambien debe ser nula
                      ;; (o haber llegado a la vez al final)
          (let ((firstEl (first cl1)))
            (and (search-literal-p cl2 firstEl) ;; Busco el primer elemento
                 (equal-clauses (rest cl1)
                                (elim-elem cl2 firstEl)))))) ;; Sigo la recursion, pero
                                                             ;; eliminando el elemento encontrado.

;;
;; Funcion que busca la clausula cl1 en la lista lst
;; comparandolas con la funcion equal-clauses.
;; Devuelve t si lo encuentra y false si no.
;; Se podria sustituir por una utilizacion de la funcion
;; ya incluida en lisp llamada 'member'
;;

(defun search-clause-p (cl1 lst)
    (unless (null lst)
        (or (equal-clauses cl1 (first lst))
            (search-clause-p cl1 (rest lst)))))

;;
;; Version recursiva de la funcion principal
;; que recibe las clausulas con los literales
;; repetidos eliminados
;;

(defun elim-repeated-clauses-rec (cnf)
    (unless (null cnf)
        (if (search-clause-p (first cnf) (rest cnf))
            (elim-repeated-clauses-rec (rest cnf))
            (cons (first cnf)
                  (elim-repeated-clauses-rec (rest cnf))))))

;;
;; Funcion principal que primero elimina los literales
;; repetidos y luego llama a la version recursiva de si misma
```

```
;;

(defun elim-repeated-literals-from-clauses (cnf)
    (unless (null cnf)
            (cons (eliminate-repeated-literals (first cnf))
            (elim-repeated-literals-from-clauses (rest cnf)))))

(defun eliminate-repeated-clauses (cnf)
    (elim-repeated-clauses-rec (elim-repeated-literals-from-clauses cnf)))
```

COMENTARIOS

- En esta función hemo utilizado muchas funciones auxiliares:
  - elim-elem: recibe como argumentos una lista de literales y un elemento, y devuelve la lista con el elemento eliminado (si está) una sola vez, es decir, si el elemento aparece repetido, solo lo eleimina una vez.
  - equal-clauses: comprueba de forma recursiva si dos cláusulas son iguales, es decr si tienen los mismos literales el mismo número de veces. Para ellos va eliminando de una de las listas cada elemento según lo va encontrando, utilizando elim-elem (no es destructiva)
  - search-clause-p: recibe una lista de cláusulas y una cláusula, y recorre la lista recursivamente buscando la cláusula. Devuelve t si la cláusula está en la lista y devuelve nil en caso contrario.
  - elim-repeated-clauses-rec: versión recursiva de la función principal. Realmente, la única razón por la que esta fucnión es necesaria es porque la función principal llama a elim-repeated-literals-from-clauses antes de continuar la recursión, y esto solo tiene sentido hacerlo una vez por eficiencia. - elim-repeated-literals-from-clauses: recibe una lista de cláusulas y elimina los literales repetidos de todas ellas, utilizando las funciones del apartado anterior (simplemente recorre la lista de forma recursiva eliminando los literales de cada elemento)

4.3.3 Determinación de si una cláusula subsume a otra

PSEUDOCÓDIGO

```
Entrada:  K1, K2 claúsulas
Salida:  K1 si K1 subsume K2.  Else, NIL
Procesamiento:
Si K1 contenido en K2:
 devuelve K1
Else:
 devuelve NIL
```

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)
  (if (subsetp K1 K2 :test 'equal)
    K1
    NIL))

;;
;;  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
(subsume '((~ a)) '((~ a) b (~ c)))
;; ((~ A))
```

COMENTARIOS

- Según nuestra definición, K1 subsume a K1. Por tanto buscamos K1 $\subseteq$ que K2

4.3.4 Eliminación de cláusulas subsumidas

PSEUDOCÓDIGO

```
Entrada:  FBF en CNF
Salida:    FBF equivalente sin cláusulas subsumidas
Procesamiento:
Para cada cláusula en cnf:
 Si ninguna K en resto de cnf te subsume y
 Ninguna K en la lista no-subsumidas te subsume:
  Añade cláusula a no-subsumidas y continúa procesando.
 Si no:
  Continúa procesando hasta que no queden cláusulas en cnf por comprobar.
```

CÓDIGO

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

;; Introduce elem en cnf1 si elem no es subsumido por cnf1 o cnf2.
;; Repite el proceso con el elemento siguiente a elem; el primero de cnf2
(defun rec-elim-subsum(cnf1 elem cnf2)
  (cond

    ((null cnf2)
     ;; Caso base: hay que parar la recursion: devolvemos cnf1 con o sin elem
     (if (noone-subsumes elem cnf1)
         (my-cons elem cnf1)
         cnf1))
    ((and (noone-subsumes elem cnf1) (noone-subsumes elem cnf2))
     ;; Si nadie subsume a elemn, aniadimos elem a cnf1 y repetimos con 1st-cnf2 y rest-cnf2
     (rec-elim-subsum (my-cons elem cnf1) (first cnf2) (rest cnf2)))
    (t
     ;; Si alguien subsume a elem, no aniadimos elem a cnf1 y repetimos con 1st-cnf2 y rest-cnf2
     (rec-elim-subsum cnf1 (first cnf2) (rest cnf2)))))

(defun eliminate-subsumed-clauses(cnf)
  (if
    (equal cnf '() )
    cnf
    (rec-elim-subsum () (first cnf) (rest cnf))))
;;
;;  EJEMPLOS:
;;
(eliminate-subsumed-clauses
 '((a b c) (b c) (a (~ c) b) ((~ a) b) (a b (~ a)) (c b a)))
;;; ((A (~ C) B) ((~ A) B) (B C)) ;; el orden no es importante
(eliminate-subsumed-clauses
 '((a b c) (b c) (a (~ c) b) (b) ((~ a) b) (a b (~ a)) (c b a)))
;;; ((B))
(eliminate-subsumed-clauses
 '((a b c) (b c) (a (~ c) b) ((~ a)) ((~ a) b) (a b (~ a)) (c b a)))
;;; ((A (~ C) B) ((~ A)) (B C))
(eliminate-subsumed-clauses '((a)))
;;; ((A))
(eliminate-subsumed-clauses '())
;;; NIL
(eliminate-subsumed-clauses '(()))
;; (NIL)
```

COMENTARIOS

- Para este apartado se han implementado otras funciones auxiliares que no hemos añadido para no alargar innecesariamente la memoria. La función auxiliar principal sí está aquí incluida, las demás pueden encontrarse correctamente comentadas en el código.

4.3.5 Detección de tautologías

PSEUDOCÓDIGO

Entrada: K cláusula
Salida: T si es tautología, Nil else
Procesamiento:
Para cada literal de K:
Si tu negación está en K, K es tautología

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)
  (cond
    ((null K) NIL) ;Caso base: no quedan eltos en clausula para comprobar
    ((member
        (reduce-scope-of-negation
           (list +not+ (first K))) K :test 'equal) T) ;Si (x (~x)) in K, es tautologia
    (t (tautology-p (rest K)))))
;;
;;  EJEMPLOS:
;;
(tautology-p '((~ B) A C (~ A) D)) ;;; T
(tautology-p '((~ B) A C D))    ;;; NIL
```

4.3.6 Eliminación de tautologías

PSEUDOCÓDIGO

Entrada: cnf
Salida: cnf sin cláusulas tautología
Procesamiento:
Para cada cláusula de cnf:
 Si no es tautología, añadela al resultado.

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)
  ;; Concateno listas con todas las clausulas que no son tautology
  (mapcan #'(lambda (x) (unless (tautology-p x) (list x))) cnf))

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

4.3.7 Simplificación de FBF a FNC

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.7
;; simplifica FBF en FNC
;;       * elimina literales repetidos en cada una de las clausulas
;;       * elimina clausulas repetidas
;;       * elimina tautologias
;;       * elimina clausulass subsumidas
;;
;; RECIBE  : cnf  FBF en FNC
```

```
;; EVALUA A : FNC equivalente sin clausulas repetidas,
;;           sin literales repetidos en las clausulas
;;           y sin clausulas subsumidas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun simplify-cnf (cnf)
  (eliminate-subsumed-clauses
    (eliminate-tautologies
      (eliminate-repeated-clauses
        (mapcar #'(lambda(x) (eliminate-repeated-literals x) ) 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
```

COMENTARIOS

- No hemos añadido pseudocódigo ni comentarios en el código porque creemos que la función as autoexplicativa, y simplemente consiste en aplicar secuencialmente las simplificaciones programadas en los apartados anteriores.

4.4 Construcción de RES

4.4.1 Cálculo del conjunto $\alpha_\lambda^{(0)}$

PSEUDOCÓDIGO

```
Entrada:cnf , lambda literal positvo
Salida:  cláusulas neutras para lambda
Procesamiento:
Para cada cláusula de cnf:
 Si es neutra para lambda:
  Aniade a lista-neutras
```

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

```
;; Devuelve T if (lambda not in K) AND (~lambda not in K)
(defun not-contains-lambda(lambda K)
   (not (or (member lambda K :test 'equal)
            (member (list +not+ lambda) K :test 'equal))))
;; Devuelve el subconjunto de conj que no contienen ni lambda ni ~lambda
(defun aux-extract-neutral (subconj conj lambda)
  (cond
    ((null conj)
     ;; Si conj vacio, he terminado. Devuelvo subconj con el filtro
     subconj)
    ((not-contains-lambda lambda (first conj))
     ;; Si 1a clausula de conj es neutra para lambda, aniado la clausula a subconj y repito
     (aux-extract-neutral (adjoin (first conj) subconj :test 'equal) (rest conj) lambda))
    (t
      ;; Si no, no la aniado a subconj y repito
      (aux-extract-neutral subconj (rest conj) lambda))))

(defun extract-neutral-clauses (lambda cnf)
  (aux-extract-neutral () cnf lambda))
;;
;;  EJEMPLOS:
;;
(extract-neutral-clauses 'p
                         '((p (~ q) r) (p q) (r (~ s) q) (a b p) (a (~ p) c) ((~ r) s)))
;; (((~ R) S) (R (~ S) Q))
(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)))
;; ((A (~ P) C) (A B P) (P Q))
(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 Cálculo de conjunto $\alpha_{\lambda}^{(+)}$

PSEUDOCÓDIGO

Entrada: cnf
Salida: cláusulas lambda-positivas
Procesamiento:
Para cada cláusula en cnf:
 Si es lambda positiva:
  Añade a lista-positivas

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

;; Extrae las clausulas de conj que pasan la funcion 'equal
(defun extract-clauses (subconj conj elto)
  (cond
    ((null conj)
     ;; Si no quedan clausulas q mirar en conj, ya he terminado: subconj contiene el filtro
     subconj)
    ((member elto (first conj) :test 'equal)
     ;; si 'elto pertenece a la 1a clausula de conj, aniado clausula a subconj y repito
     (extract-clauses (adjoin (first conj) subconj :test 'equal) (rest conj) elto))
    (t
      ;; si no, no aniado a subconj y repito
      (extract-clauses subconj (rest conj) elto))))

(defun extract-positive-clauses (lambda cnf)
  (extract-clauses () cnf lambda))
; Llamo a extract-clauses para que filtre clausulas de cnf que contengan 'lambda


;;
;;  EJEMPLOS:
;;
(extract-positive-clauses 'p
                          '((p (~ q) r) (p q) (r (~ s) q) (a b p) (a (~ p) c) ((~ r) s)))

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

COMENTARIOS

- Se adjunta el código de la función extract-clauses, que nos sirve para extraer de una cnf simplificada aquellas cláusulas que contentan al elemento elto

4.4.3 Cálculo de conjunto $\alpha_{\lambda}^{(-)}$

PSEUDOCÓDIGO

```
Entrada:  cnf simplificada, lambda literal positivo
Salida:  conjunto de cláusulas de cnf que contienen   lambda
Procesamiento:  Para cada cláusula en cnf:
 Si contiene a   lambda:
  Añade a lista-negativas
```

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)
  (extract-clauses () cnf (list +not+ lambda)))
; Llamo a extract-clauses para que filtre clausulas de cnf que contengan '(~ 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
```

COMENTARIOS

- Esta función emplea la función auxiliar del apartado anterior, para poder extraer las cláusulas que contienen lambda.

4.4.4 Reolvente de dos cláusulas

PSEUDOCÓDIGO

```
Entrada:lambda lit positivo, K1 K2 cláusulas
Salida:  Cláusula después de aplicar resolución
Procesamiento:
1.  Comprueba si puedes resolver con lambda en K1 y K2.
2.  Si puedes, une ambas cláusulas.
3.  Retira del resultado los literales lambda y   lambda
```

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

;;This test evals TRUE if you can resolve on K1 and K2
(defun resolvable (lambda K1 K2)
  (let ((notlambda (list +not+ lambda)))
     ;; Comprueba si lambda en K1 y no lambda en K2
    (or (and (member lambda K1 :test 'equal)
             (member notlambda K2 :test 'equal))
        ;; o no lambda en K1 y lambda en K2
       (and (member notlambda K1 :test 'equal)
            (member lambda K2 :test 'equal)))))


(defun resolve-on (lambda K1 K2)
  (unless (not (resolvable lambda K1 K2))
    ;; A la union de conjuntos K1 y K2, restamos el conjunto (lambda ~lambda)
    (list
      (set-difference (union K1 K2 :test 'equal)     ;; {K1} U {K2} \
                    (list lambda (list +not+ lambda)) ;; {(lambda ~lambda)}
                    :test '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
```

COMENTARIOS

- Hemos añadido una función auxiliar que comprueba, dadas λ y 2 cláusulas, si podemos usar resolución con ellas. Esto es, si podemos encontrar el literal λ en una de ellas y $\neg\lambda$ en la otra.

4.4.5 Cláusulas RES para CNF

PSEUDOCÓDIGO

Entrada: lambda literal positivo, fnc
Salida: clausulas resultado de la resolucion
Procesamiento:
1. Extrae las $\alpha_{\lambda}^{(+)}$
2. Extrae las $\alpha_{\lambda}^{(-)}$
3. Para todo (K en $\alpha_{\lambda}^{(+)}$, K2 en $\alpha_{\lambda}^{(-)}$):
   Añade a lista resolucion(K1, K2, λ
4. Añade a lista $\alpha_{\lambda}^{(0)}$

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

;; Devuelve una lista con el resultado de aplicar res sobre K1 y todas las Ki de cnf
(defun resolve(lambda K1 cnf)
  (unless (null cnf)
    (union (resolve-on lambda K1 (first cnf))
           (resolve lambda K1 (rest cnf))
           :test 'equal)))

;; Devuelve una lista con el resultad de aplicar res sobre todos los posibles
;; pares de clausulas (K1, K2) con K1 en cnf1 y K2 en cnf2
(defun resolve-pairs (lambda cnf1 cnf2)
  (unless (null cnf1)
    (union (resolve lambda (first cnf1) cnf2)
           (resolve-pairs lambda (rest cnf1) cnf2)
           :test 'equal)))

(defun build-RES (lambda cnf)
  (union (extract-neutral-clauses lambda cnf)
         (resolve-pairs lambda
                        (extract-negative-clauses lambda cnf)
                        (extract-positive-clauses lambda cnf))
         :test 'equal-clauses))
;;
;;  EJEMPLOS:
;;
(build-RES 'p NIL)
;; NIL
(build-RES 'P '((A (~ P) B) (A P) (A B)))
;; ((B 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) A B) (#1=(~ P)) (P A B) (P #1#) (P))
(build-RES 'p '((p q) (c q) (a b q) (p (~ q)) (p (~ q))))
;; ((A B Q) (C Q))
```

COMENTARIOS

- Para este apartado hemos añadido un par de funciones auxiliares que simplifican el código. Aunque aquí no aparecen las pruebas, se incluye una batería de ejemplos en el código.
- La primera hace, dado λ, todas las posibles resoluciones entre cláusulas $\alpha_\lambda^{(+)}$ y $\alpha_\lambda^{(-)}$.
- La segunda se encarga de parte de la funcionalidad de la primera: hace todas las posibles resoluciones entre una cláusula y una FBF, llamando de manera recursiva a `resolve-on`

4.5 Calcular $RES_\lambda(\alpha)$ para una FNC α

PSEUDOCÓDIGO

Entrada: FBF en CNF
Salida: T si CNF SAT, NIL else
Procesamiento:
LLamada con (cnf, $\lambda_1 ... \lambda_n$)
1. Si unsat(cnf) -> NIL
2. Si no hay λ_i para resolver -> T
3. Si no:
   cnf <= resolución sobre λ_1
   simplifica(cnf)
   repite llamada con (cnf, $\lambda_2 ... \lambda_n$)

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

;; Acumula en posit todos los literales positivos de la clausula K
(defun rec-positive-in-clause (clause)
  (cond
    ((null clause)
     ;; No quedan literales en clause que mirar. Hemos terminado
     NIL)
    ((positive-literal-p (first clause))
     ;; El 1er lit de clause es positivo: la union del 1er lit y los positivos en rest-clause
     (union (list (first clause))
            (rec-positive-in-clause (rest clause))
            :test 'equal))
    (t
```

```
      ;; El 1er lit de clause es negativo los positivos de rest-clause
      (rec-positive-in-clause (rest clause)))))

;; Devuelve un conjunto todos los literales positivos de cnf
(defun rec-positive-in-cnf (cnf)
  (unless
    (null cnf)
    (union (rec-positive-in-clause (first cnf))
           (rec-positive-in-cnf (rest cnf))
           :test 'equal)))

;; Devuelve T si cnf es UNSAT.
(defun unsat (cnf)
  (if
    (member NIL cnf :test 'equal)
    t
    NIL))

(defun rec-RES-SAT (lambdas cnf)
  (cond
    ;; Si cnf vacia -> T
    ((null cnf) T)
    ;; Si cnf UNSAT -> NIL
    ((unsat cnf) NIL)
    ;; Si no hay mas literales sobre los que resolver -> True
    ((null lambdas) T)
    ;; Else: aplicamos res sobre cnf con el primer lambda, repetimos para los demas lambdas
    (t (rec-RES-SAT (rest lambdas)
                    (simplify-cnf (build-RES (first lambdas)
                                             cnf))))))
(defun RES-SAT-p (cnf)
  (rec-RES-SAT (rec-positive-in-cnf cnf)
               cnf))

;;
;;  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
```

COMENTARIOS

- De nuevo, hemos incluido algunas funciones auxiliares aunque no su batería de ejemplos, que puede encontrarse en el código.
- Hay dos funciones auxiliares que permiten conjuntamente extraer todos los literales positivos en una FBF en FNC: una de ellas recorre todas las cláusulas, y la otra se encarga de buscar literales dentro de una cláusula.
- La otra función nos permite evaluar si una FBF en FNC simplificada es UNSAT, buscando NIL entre sus componentes.

4.6

PSEUDOCÓDIGO

```
Entrada:  wff, w FBFs infijo
Salida:    T si w consecuencia lógica de wff, else NIL
Procesamiento:
1.  wff <= wff en prefijo y simplificada
2.  w <= w en prefijo y simplificada
3.  union <= union ambas bases de conocimiento
4.  Si unsat(union), devuelve T. Else, devuelve NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.6:
;; Resolucion basada en RES-SAT-p
;;
;; RECIBE  : wff - FBF en formato infijo
;;           w   - FBF en formato infijo
;;
;; EVALUA A : T si w es consecuencia logica de wff
;;           NIL en caso de que no sea consecuencia logica.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun logical-consequence-RES-SAT-p (wff w)
  (not ;; Si la union es UNSAT, devolvemos True
    (RES-SAT-p ;; resolvemos sobre la union de wff y not w
      (union (simplify-cnf
               (wff-infix-to-cnf wff))
             (simplify-cnf
```

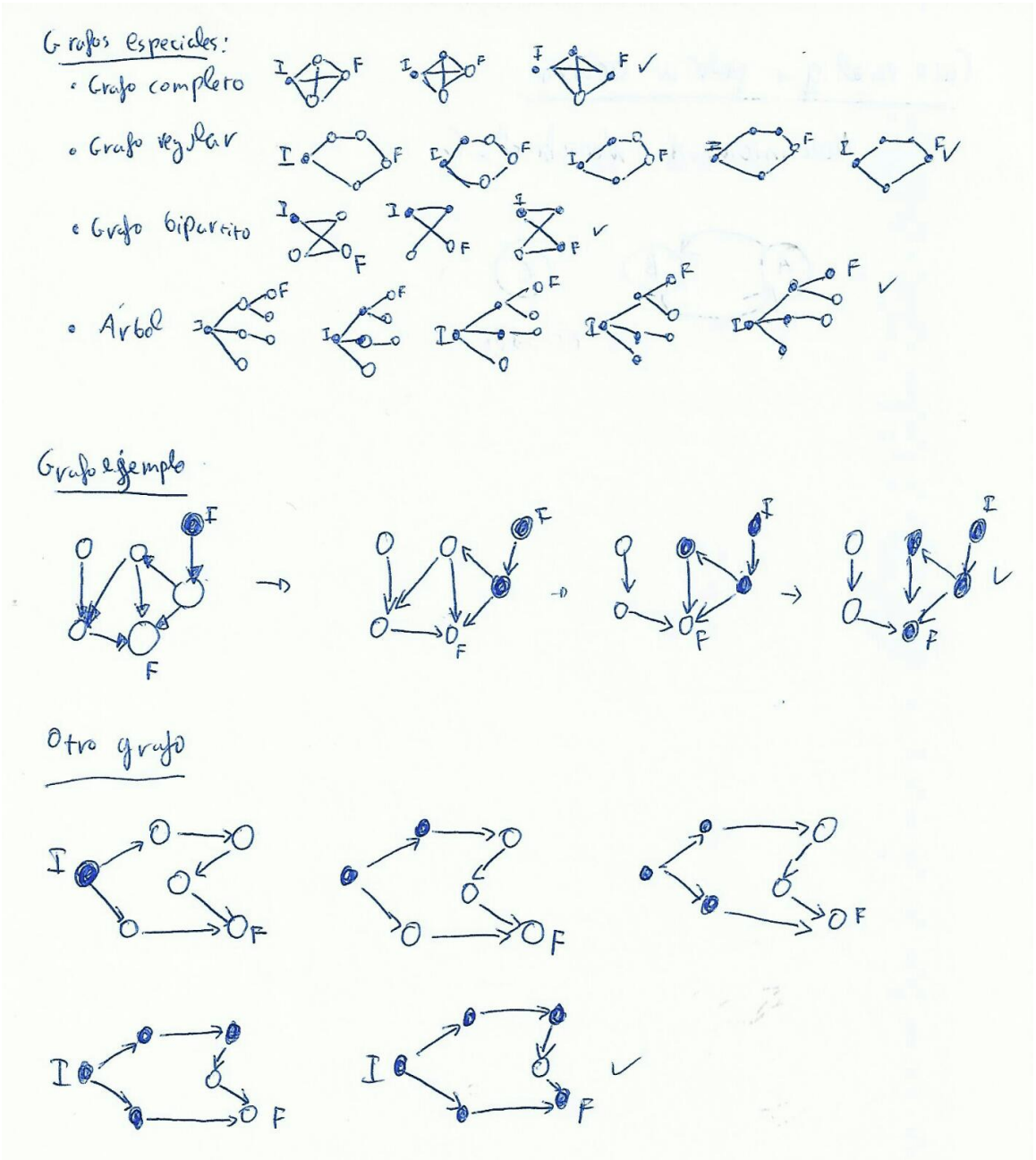
```
            (wff-infix-to-cnf
              (list +not+ w)))
           :test 'equal-clauses))))

;;
;;  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
```

COMENTARIOS

- No hemos añadido todos los ejemplos que pueden encontrarse en el código

5 Búsqueda en anchura

5.1



5.2

PSEUDOCÓDIGO

```
Entrada:  g grafo, root nodo raiz
Procesamiento:
BFS(G, root):
 inicializar(G)
 visita(root); dist-a-root(root) = 0; encola(root)
 mientras cola no vacia:
   extrae v de cola
   para cada hijo(v), si no visitado:
     visita(hijo), actualiza dist-a-root(hijo), encola(hijo)
```

5.3

5.4

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Breadth-first-search in graphs
;;;
(defun bfs (end queue net)
   (if (null queue) '() ;; No quedan listas en cola: hemos terminado
      (let* ((path (first queue))
             (node (first path)))
        (if (eql node end)
             ;; Si hemos llegado al destino: invertimos camino y esa es nuestra solucion
            (reverse path)
            ;; Else, aniadimos a la parte restante de la cola los nuevos caminos
            ;; (que tienen a los nodos adyacentes como inicio) y repetimos
            (bfs end
               (append (rest queue)
                   (new-paths path node net))
               net)))))
;; Para todos los adyacentes a node creo un nuevo camino
;; uniendo el nodo adyacente con el camino que ya conociamos del padre
(defun new-paths (path node net)
   (mapcar #'(lambda(n)
            (cons n path))
          (rest (assoc node net))))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.5

BFS resuelve el problema de encontrar el camino más corto porque la diferencia de coste entre un camino explorado y el siguiente es una función siempre creciente (por la construcción del algoritmo, no se va a encontrar un camino de coste 2, luego uno de coste 4 y luego uno de coste 3). Por tanto, cuando se encuentre un camino

a la meta siempre seráel más corto. dos los enlaces tienen la misma longitud (no hay costes diferentes asociados a cada enlace) y por tanto BFS es óptimo.

Nota: para garantizar el camino más corto pedimos que todos los enlaces tengan el mismo coste

5.6

Este es el resultado de hacer un rastreo de la funcion `shortest-path` y las funciones implicadas `new-paths` y `bfs`.

```
    (shortest-path 'a 'f '((a d) (b d f) (c e) (d f) (e b f) (f)))
1.  Trace:  (SHORTEST-PATH 'A 'F '((A D) (B D F) (C E) (D F) (E B F) (F)))
2.  Trace:  (BFS 'F '((A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
3.  Trace:  (NEW-PATHS '(A) 'A '((A D) (B D F) (C E) (D F) (E B F) (F)))
3.  Trace:  NEW-PATHS ==> ((D A))
3.  Trace:  (BFS 'F '((D A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
4.  Trace:  (NEW-PATHS '(D A) 'D '((A D) (B D F) (C E) (D F) (E B F) (F)))
4.  Trace:  NEW-PATHS ==> ((F D A))
4.  Trace:  (BFS 'F '((F D A)) '((A D) (B D F) (C E) (D F) (E B F) (F)))
4.  Trace:  BFS ==> (A D F)
3.  Trace:  BFS ==> (A D F)
2.  Trace:  BFS ==> (A D F)
1.  Trace:  SHORTEST-PATH ==> (A D F)
```

En él se muestran las 3 llamadas que se hacen por recursión a la funcion `BFS`. En las 2 primeras no se ha llegado todavía al nodo F, y por tanto generan sendas llamadas a la función `new-paths`. En la tercera llamada, el nodo F ya ha sido encontrado y se encuentra en el primer camino, y por eso es devuelto el `reverse` del path `(F D A)` en las tres ejecuciones de `BFS`.

Acerca de las llamadas a `new-paths`: en la primera, el path que se construye es el formado por ir de `A` a su único adyacente `D`. En la segunda, el path que se construye es el resultado de "unir" el path `(D A)` con el nodo `F` adyacente a `D`, dando como resultado `(D A F)`.

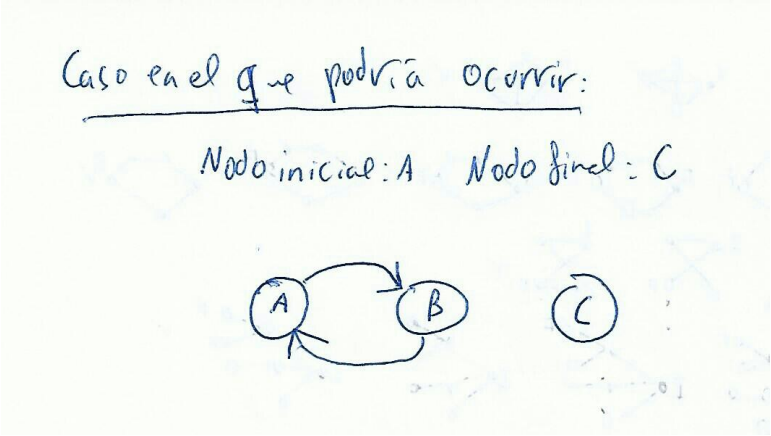
5.7

La net (lista de adyacencia) asociada a este grafo es:

```
( (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) )
```

y la llamada que necesitamos para este apartado es: `(shortest-path 'f 'c '( (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) ))`, que devuelve como resultado `(F B A C)`

5.8 Mejora al código anterior



El fallo del código anterior ocurría cuando había un bucle en un grafo sin solución. Esto ocurría por ejemplo en el grafo '((A B) (B A) (C)) tal y como se ilustra en la figura. Por ello, la única modificación que ha sido necesaria, ha sido en la función llamada new-paths, encargada de explorar el nodo y añadir los nuevos nodos a explorar a la cola. La modificación ha consistido en que ahora, en lugar de añadir a la cola los nodos sin filtro, comprueba que no hemos pasado por ellos antes de añadirlo, es decir, que no están en la variable 'path'.

El resto de funciones son idéticas excepto por sus nombres y por las funciones a las que llaman cada una.

```
;;;;;;;;;;;;;;;; Ejercicio 5.8 ;;;;;;;;;;;;;;;;;;;;
;; La funcion es practicamente identica a la del ejercicio
;; 5.3, con la salvedad de que al aniadir nuevos nodos a la cola
;; llama a la funcion new-paths-no-repetition que antes de aniadir un
;; camino nuevo, comprueba que no tiene nodos repetidos.
;; De esta manera nunca hay caminos que formen un bucle infinito.
;;

;;;;
;; Funcion auxiliar que mira si la lista recibida como
;; primer argumento contiene el elemento recibido como segundo
;; argumento
;;;;
(defun list-contains (lst elt)
  (unless (null lst)
    (or (eql (first lst) elt)
      (list-contains (rest lst) elt))))


;;;;
;; Funcion auxiliar que dado un camino seguido hasta ahora,
;; el nodo actual, y el grafo, devuelve una lista de los
;; posibles siguientes caminos a seguir desde el nodo
```

```
;; actual, siempre y cuando estos no esten repetidos.
;;;;
(defun new-paths-no-repetition (path node net)
     (mapcan #'(lambda(n)
                 ;; Comprobamos que el camino seguido
                 ;; hasta ahora no contiene el siguiente nodo
                 ;; para evitar caminos con nodos repetidos
               (and (not (list-contains path n))
                  (list (cons n path))))
               (rest (assoc node net))))


;;;;
;; Funcion de busqueda en anchura sin caminos con
;; elementos repetidos.
;;;;
(defun bfs-improved (end queue net)
   (if (null queue) '() ;; No quedan listas en cola: hemos terminado
      (let* ((path (first queue))
             (node (first path)))
         (if (eql node end)
              ;; Si hemos llegado al destino: invertimos camino y esa es nuestra solucion
             (reverse path)
             ;; Else, aniadimos a la parte restante de la cola los nuevos caminos
             ;; (que tienen a los nodos adyacentes como inicio) y repetimos
             (bfs-improved end
                (append (rest queue)
                    (new-paths-no-repetition path node net))
                net)))))

;;;;
;; Funcion mejorada de la busqueda de mejor camino entre dos nodos
;; mediante bfs-improved.
;;;;
(defun shortest-path-improved (start end net)
   (bfs-improved end (list (list start)) net))
```