Universidad Autónoma

Inteligencia Artificial

Práctica 1

Memoria

Autores:
Adrián Fernández
Santiago González-Carvajal

Pareja 7
Grupo 2302

6 de marzo de 2018

Índice

1. Similitud Coseno

1.1. Función similitud coseno

Batería de pruebas

```
» (sc-rec '(1 0) '(1 0)) --> 1.0
» (sc-rec '(1 0) '(0 1)) --> 0.0
» (sc-rec '(1 0) '(1 1)) --> 0.70710677
» (sc-mapcar '(1 0) '(1 0)) --> 1.0
» (sc-mapcar '(1 0) '(0 1)) --> 0.0
» (sc-mapcar '(1 0) '(1 1)) --> 0.70710677

» (sc-rec '(1 2 3) '(3 2 1)) --> 0.7142857
» (sc-mapcar '(1 2 3) '(3 2 1)) --> 0.7142857

» (sc-rec '(3 4) '(1 0)) --> 0.6
» (sc-rec '(3 4) '(3 0)) --> 0.6
» (sc-mapcar '(3 4) '(1 0)) --> 0.6
» (sc-mapcar '(3 4) '(3 0)) --> 0.6
```

Pseudocódigo

```
Entrada: x (vector de números reales), y (vector de números reales)
Salida: sc (valor del coseno del ángulo que forman los vectores x, y)

Procesamiento:
sim-cos (x y):
    denom <- sqrt(pesc(x x))*sqrt(pesc(y y))
    si denom es 0.0
        devuelve NIL
    en caso contrario
        devuelve pesc(x y)/denom
```

Código

```
(defun sc-rec (x y)
  (if(or (null x) (null y))
      nil
```

```
    (let ((denom (* (sqrt (pesc-rec x x)) (sqrt (pesc-rec y y)))))
      (if(eql denom 0.0) ;;Si se anula el denominador
          nil
        (/ (pesc-rec x y) denom)))))

(defun sc-mapcar (x y)
  (if(or (null x) (null y))
      NIL
    (let ((denom (* (sqrt (pesc-mapcar x x)) (sqrt (pesc-mapcar y y)))))
      (if(eql denom 0.0) ;;Si se anula el denominador
          nil
        (/ (pesc-mapcar x y) denom)))))
```

Comentarios sobre implementación

Para realizar ambas implementaciones, hemos creado 2 funciones que implementan el producto escalar. Una de forma recursiva y otra utilizando mapcar. El procesamiento de la función similitud coseno sería el mismo, sustituyendo pesc por una de las implementaciones para cada caso.

1.2. Función de categorías

Batería de pruebas

```
» (sc-conf '(1 0) '((1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  1)
--> NIL
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  1)
-- > ((1 0))
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  0.95)
-- > ((1 0) (1 0.2))
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  0.9)
-- > ((1 0) (1 0.2) (1 0.4))
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  0.8)
--> ((1 0) (1 0.2) (1 0.4) (1 0.6))
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  0.75)
--> ((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8))
» (sc-conf '(1 0) '((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))  0.7)
--> ((1 0) (1 0.2) (1 0.4) (1 0.6) (1 0.8) (1 1))
```

Pseudocódigo

```
Entrada: x (vector de números reales), vs (lista de vectores reales),
conf (número real)
Salida: vconf (vectores cuya similitud con respecto a la categoría es
superior al nivel de confianza, ordenados)

Procedimiento:
sc-conf (x vs conf):
    para cada v en vs:
        si sim(v) > conf
        add(lst similitud(x v))
    devolver ordenar(lst)
```

Código

```
(defun sc-conf (x vs conf)
  (if(or (or (null x) (null vs)) (or (> conf 1) (< conf 0)))
      nil
    (sort
     (copy-list
      (remove-if
       #'(lambda (y) (< (sc-mapcar x y) conf)) vs))
     #'>:key #'(lambda (y) (sc-mapcar x y)))))
```

Comentarios sobre implementación

Utilizar un lenguaje funcional para implementar esta función es especialmente útil, ya que requiere aplicar la función similitud coseno a una lista de vectores y filtrar los vectores con el valor de esa función. Esto se logra fácilmente mediante mapcar y expresiones lambda.

1.3. Clasificador por similitud coseno

Pseudocódigo

```
Entrada: cats (lista vectores cuyo primer elemento es un
identificador), texts(lista vectores cuyo primer elemento
es un identificador), f (función)
```

```
Salida: lsol (lista de pares cuyo  segundo elemento es
la mayor similitud coseno del conjunto de vectores cats con
el vector de texts que ocupa la posición del par y el primero
es el identificador de dicho vector)

Proceso:
sc-classifier (cats texts f):
    para cada t en texts:
        (id, valor) <-- maximo(f(t cats))
        add(lst (id valor))
    devuelve lst
```

Código

```
(defun ord-conf (x vs func)
  (if(or (or (null x) (null vs)) (null func))
      nil
    (first
     (sort
      (copy-list
       (mapcar #'(lambda (x y) (list (first y) x))
         (mapcar #'(lambda (y) (funcall func (rest x) (rest y))) vs)
         vs))
      #'> :key #'second)))) ;;Ordenamos de mayor a menor segun la similitud cosenc

(defun sc-classifier (cats texts func)
  (if(or (or (null cats) (null texts)) (null func))
      nil
    (mapcar #'(lambda (y) (ord-conf y cats func)) texts)))
```

Comentarios sobre implementación

La implementación es parecida a la de la función anterior, pero la función es más general.

1.4. Pruebas llamando a sc-classifier

Batería de pruebas

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

» (time
   (sc-classifier
    '((1 43 23 12) (2 33 54 24) (3 4 3 1) (4 2 1 2) (5 2 1 2)
    (6 2 1 2) (7 2 1 2))
    '((1 3 22 134) (2 43 26 58) (3 23 21 83) (4 12 32 1 4))
    #'sc-rec)) --> ((4 0.72641367) (4 0.99035937) (4 0.87609464)
    (2 0.9219237))
                  time: 0.007000 sec
» (time
   (sc-classifier
    '((1 4 3 1 93 1 23 9) (2 2 1 2 2321 21 21 3))
    '((1 3 22 134 122 41 48 4) (2 43 26 58 3 4 5 6))
    #'sc-rec)) --> ((1 0.6841774) (1 0.10203057))
                  time: 0.001000 sec
» (time
   (sc-classifier
    '((1 4 3 1 93 1 23 9 43 34) (2 2 1 2 2321 21 21 3 12 43))
    '((1 3 22 134 122 41 48 4 12 34) (2 43 26 58 3 4 5 6 95 23))
    #'sc-rec)) --> ((1 0.6613261) (1 0.4072973))
                  time: 0.001000 sec

» (time
   (sc-classifier
    '((1 43 23 12) (2 33 54 24) (3 4 3 1) (4 2 1 2) (5 2 1 2)
    (6 2 1 2) (7 2 1 2))
    '((1 3 22 134) (2 43 26 58) (3 23 21 83) (4 12 32 1 4))
    #'sc-mapcar)) --> ((4 0.72641367) (4 0.99035937) (4 0.87609464)
    (2 0.9219237))
                     time: 0.008000 sec
```

```
» (time
   (sc-classifier
    '((1 4 3 1 93 1 23 9) (2 2 1 2 2321 21 21 3))
    '((1 3 22 134 122 41 48 4) (2 43 26 58 3 4 5 6))
    #'sc-mapcar)) --> ((1 0.6841774) (1 0.10203057))
                      time: 0.000000 sec
» (time
   (sc-classifier
    '((1 4 3 1 93 1 23 9 234 12) (2 2 1 2 2321 21 21 3 21 34))
    '((1 3 22 134 122 41 48 44 23 12) (2 43 26 58 3 4 5 6 12 34))
    #'sc-mapcar)) --> ((1 0.6613261) (1 0.4072973))
                      time: 0.001000 sec
```

Comentarios sobre las pruebas realizadas

Recursión es más eficiente para vectores pequeños, ya que mapcar paraleliza el proceso, y para procesos pequeños se pierde más tiempo paralelizando que ejecutando. Sin embargo mapcar es más rápido para vectores más grandes y mediante recursión se puede llenar la pila de recursión si los vectores son lo suficientemente grandes.

2. Raíces de una función

2.1. Función bisect

Batería de pruebas

```
» (bisect #'(lambda(x) (sin (* 6.26 x))) 0.1 0.7 0.001) ;;---> 0.5020995
» (bisect #'(lambda(x) (sin (* 6.26 x))) 0.0 0.7 0.001) ;;---> NIL
» (bisect #'(lambda(x) (sin (* 6.28 x))) 1.1 1.5 0.001) ;;---> NIL
» (bisect #'(lambda(x) (sin (* 6.28 x))) 1.1 2.1 0.001) ;;---> NIL
```

Pseudocódigo

```
Entrada: f (función), a (número real), b (número real mayor que a),
tol (número real)
Salida: c (solución de f en [a b] con error máximo tol)
```

```
Procesamiento:
bisect (f a b tol):
    si f(a) es 0
        devuelve a
    si f(b) es 0
        devuelve b
    si dist(a b) menor tol
        devuelve mid(a b)
    en caso contrario
        si f(a)*f(mid(a b)) menor 0
            devuelve bisect(f a mid(a b) tol)
        en caso contrario
            devuelve bisect(f mid(a b) b tol)
```

Código

```
(defun mid (a b)
  (float (/ (+ a b) 2)))

(defun dist (a b)
  (abs (- a b)))

(defun bisect_rec (f a b tol)
  (let ((middle (mid a b)))
    (cond ((= (funcall f a) 0.0)
           a)
          ((= (funcall f b) 0.0)
           b)
          ((< (dist a b) tol)
           middle)
          (t
           (if (< (* (funcall f a) (funcall f middle)) 0.0)
               (bisect_rec f a middle tol)
             (bisect_rec f middle b tol))))))

(defun bisect (f a b tol)
  (if (or (> a b) (>= (* (funcall f a) (funcall f b)) 0.0) (< tol 0.0))
      NIL
```

```
    (bisect_rec f a b tol)))
```

Comentarios sobre implementación

Puesto que se exige un número real a la salida, pero la condición de salida se mide con la norma del intervalo, hemos decidido que el output de la función sea el punto medio del primer intervalo que cumple dicha condición. También cabe destacar que en el enunciado se pide explícitamente que bisect devuelva NIL si $f(a)f(b) >= 0$, pero sería más correcto comprobar $f(a)f(b) > 0$.

2.2. Función allroot

Batería de pruebas

```
» (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) --> (0.50027466 1.0005188 1.5007629 2.001007)
» (allroot #'(lambda(x) (cos (* 6.28 x))) '(0.00 0.50 1.00 1.50 2.00)
0.0001) --> (0.2501526 0.7503967 1.2506409 1.750885)
» (allroot #'(lambda(x) (cos (* 6.28 x))) '(0.00 0.50 1.00 1.50 2.00 2.50)
0.0001) --> (0.2501526 0.7503967 1.2506409 1.750885 2.2511292)
```

Pseudocódigo

```
Entrada: f (función), lst (lista de números reales),
tol (número real)
Salida: lsol (lista con las soluciones de f encontradas
mediante bisect entre
los puntos de lst)

Procesamiento:
allroot (f lst tol):
    si lst vacía
        devuelve NIL
    si rest(lst) vacía
        devuelve NIL
    en caso contrario
```

```
devuelve concatenar(bisect(f first(lst) first(rest(lst)) tol)
                    allroot(f rest(lst) tol))
```

Código

```
(defun allroot (f lst tol)
  (mapcan #'(lambda (x y)
              (let ((root (bisect f x y tol)))
                (when root
                  (list root))))
    lst
    (rest lst)))
```

Comentarios sobre implementación

Se implementa fácilmente con una llamada recursiva y la función bisect.

2.3. Función allind

Batería de pruebas

```
» (allind #'(lambda(x) (sin (* 6.28 x))) 0.1 2.25 1 0.0001)
--> NIL
» (allind #'(lambda(x) (sin (* 6.28 x))) 0.1 2.25 2 0.0001)
--> (0.50027084 1.0005027 1.5007347 2.0010324)
» (allind #'(lambda(x) (cos (* 6.28 x))) 0.0 2.14 3 0.0001)
--> (0.25009555 0.7503519 1.2506084 1.7508645)
» (allind #'(lambda(x) (cos (* 6.28 x))) 0.0 2.15 4 0.0001)
--> (0.2501488 0.75038075 1.2506126 1.7509103)
```

Pseudocódigo

```
Entrada: f (función), a (número real), b (número real mayor que a),
N (número natural), tol (número real)
Salida: lsol (lista con las soluciones de f encontradas en la
partición de [a b] en 2^N intervalos)

Procesamiento:
allind (f a b N tol):
```

```
    si N es 1
        devuelve allroot(f list(a mid(a b) b) tol)
    en caso contrario
        devuelve concatenar(allind(f a mid(a b) N-1 tol)
                            allind(f mid(a b) b N-1 tol))
```

Código

```
(defun mid_lst (a b)
  (list a (mid a b)))

(defun allind-rec (f a b N tol)
  (if (= N 1)
      (mid_lst a b)
    (let ((middle (mid a b)))
      (append
       (allind-rec f a middle (- N 1) tol)
       (allind-rec f middle b (- N 1) tol)))))

(defun allind (f a b N tol)
  (cond ((or (null a) (null b) (> a b) (< N 1) (< tol 0.0))
         NIL)
        (t
         (allroot f (append (allind-rec f a b N tol) (list b)) tol))))
```

Comentarios sobre implementación

Hemos decidido implementar esta función basándonos en allroot, ya que se adapta perfectamente a la funcionalidad exigida y no requiere mayor esfuerzo. Para conseguir la lista de 2^N elementos basta con partir el intervalo por la mitad de forma recursiva N veces.

3. Combinación de listas

3.1. Combinar elemento con lista

Batería de pruebas

```
» (combine-elt-lst 'a nil) --> NIL
» (combine-elt-lst 'a '(1 2 3)) --> ((A 1) (A 2) (A 3))
» (combine-elt-lst '4 '(1 2 3)) --> ((4 1) (4 2) (4 3))
» (combine-elt-lst nil '(1 2 3)) --> NIL
```

Pseudocódigo

```
Entrada: elt (elemento a combinar), lst (lista con la que
combinar el elemento)
Salida: lsol (lista de listas con todas las combinaciones
posibles del elemento con los elementos
de la lista)

Procesamiento:
combine-elt-lst (elt lst):
    para cada el en lst:
        lis <-- lista(elt el)
        add(lsol lis)
    devuelve lsol
```

Código

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

Comentarios sobre implementación

La única decisión de diseño que hemos realizado ha sido la de usar (list) en vez de (cons), y esto ha sido simplemente, con el objetivo de conseguir la salida pedida en el enunciado de la práctica ya que (cons) añadía un punto al par (A . B), mientras que (list) no lo hacía (A B).

3.2. Combinar lista con lista

Batería de pruebas

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

Pseudocódigo

```
Entrada: lst1 (lista a combinar), lst2 (lista con la que combinar)
Salida: lsol (lista de listas con todas las combinaciones posibles
de ambas listas)

Procesamiento:
combine-lst-lst (lst1 lst2):
    para cada el en lst1:
        lis <-- combine-elt-lst(el lst2)
        add(lsol lis)
    devuelve lsol
```

Código

```
(defun combine-lst-lst (lst1 lst2)
  (if(or (null lst1) (null lst2))
     nil
    (mapcan #'(lambda (x) (combine-elt-lst x lst2)) lst1)))
```

Comentarios sobre implementación

Hemos decidido aprovechar la implementación del apartado anterior, y por lo tanto, hemos separado el problema en N combinaciones de elemento con lista donde N es el número de elementos de la primera lista.

3.3. Función de disposiciones

Batería de pruebas

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

» (combine-list-of-lsts '((a b c) (+ -) (1 2 3 4)))
--> ((A + 1) (A + 2) (A + 3) (A + 4) (A - 1) (A - 2) (A - 3) (A - 4)
     (B + 1) (B + 2) (B + 3) (B + 4) (B - 1) (B - 2) (B - 3) (B - 4)
     (C + 1) (C + 2) (C + 3) (C + 4) (C - 1) (C - 2) (C - 3) (C - 4))

» (combine-list-of-lsts '((a b c) (+ -) (1 2 3 4) (p o) (i r s)))
-->((A + 1 P I) (A + 1 P R) (A + 1 P S)
    (A + 1 O I) (A + 1 O R) (A + 1 O S)
    (A + 2 P I) (A + 2 P R) (A + 2 P S)
    (A + 2 O I) ...)
```

Pseudocódigo

```
Entrada: lstolsts (lista de listas a combinar)
Salida: lsol (lista con todas las combinaciones posibles de las listas)

combine-list-of-lsts (lstolsts):
    si lstolsts vacía
        devuelve (lista NIL)
    en caso contrario
        para cada el en (primera lista de lstolsts):
            lis-aux <-- (combine-list-of-lsts(rest lstolsts))
            lis-res <-- combine-elt-lst(el lis-aux)
            add(lsol lista-res)
        devuelve lsol
```

Código

```
(defun combine-elt-lst-c (elt lst)
```

```
  (if(or (null elt) (null lst))
      nil
  (mapcar #'(lambda (x) (cons elt x)) lst)))

(defun combine-list-of-lsts (lstolsts)
  (if (null lstolsts)
      (list NIL)
    (mapcan
        #'(lambda (x) (combine-elt-lst-c
                       x
                       (combine-list-of-lsts (rest lstolsts))))
      (first lstolsts))))
```

Comentarios sobre implementación

Hemos implementado una primera versión, que funcionaba pero era muy larga (y poco elegante). Pero, después, hemos conseguido llegar a una solución mucho más elegante, para la que hemos tenido que ultilizar una función çombine-elt-lst-c"que es como la que hemos tenido que implementar en el apartado 3.1 salvo por el retorno (en vez de hacer (list), empleamos (cons) al combinar el elemento con la lista). A parte de eso, lo más difícil ha sido darnos cuenta del caso base.

4. Inferencia en lógica proposicional

4.1. Predicados en LISP, FBFs, cláusulas y FNCs

Determinar si una expresión es un literal positivo

Batería de pruebas:

```
» (positive-literal-p 'p)           --> T
» (positive-literal-p T)            --> NIL
» (positive-literal-p NIL)          --> NIL
» (positive-literal-p '~)           --> NIL
» (positive-literal-p '=>)          --> NIL
» (positive-literal-p '(p))         --> NIL
» (positive-literal-p '(~ p))       --> NIL
» (positive-literal-p '(~ (v p q))) --> NIL
```

Pseudocódigo:

```
Entrada: x (expresión a evaluar)
Salida: T si es un literal positivo, NIL en caso contrario

Procesamiento:
positive-literal-p (x):
    ret <-- and(not(list(x)), not(truth-value-p(x)), not(connector-p(x)))
    devuelve ret
```

Código

```
(defun positive-literal-p (x)
  (and (not (truth-value-p x))
       (not (connector-p x))
       (not (listp x))))
```

Determinar si una expresión es un literal negativo

Batería de pruebas:

```
» (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: x (expresión a evaluar)
Salida: T si es un literal negativo, NIL en caso contrario
```

```
Procesamiento:
negative-literal-p (x):
    si x es list
        first <-- unary-connector-p(first(x))
        second <-- positive-literal-p(second(x))
        third <-- null(third(x))
        ret <-- and(first, second, third)
        devuelve ret
```

Código

```
(defun negative-literal-p (x)
  (when (listp x) (and
                   (unary-connector-p (first x))
                   (positive-literal-p (second x))
                   (null (third x)))))
```

Determinar si una expresión es un literal

Batería de pruebas:

```
» (literal-p 'p)           --> T
» (literal-p '(~ p))       --> T
» (literal-p '(p))         --> NIL
» (literal-p '(~ (v p q))) --> NIL
```

Pseudocódigo:

```
Entrada: x (expresión a evaluar)
Salida: T si es un literal, NIL en caso contrario

Procesamiento:
literal-p (x):
    ret <-- or(positive-literal-p(x), negative-literal-p(x))
    devuelve ret
```

Código

```
(defun literal-p (x)
  (or (positive-literal-p x)
      (negative-literal-p x)))
```

Determinar si una expresión está en formato infijo

Batería de pruebas:

```
» (wff-infix-p 'a)                              --> T
» (wff-infix-p '(^))                            --> T  por convencion
» (wff-infix-p '(v))                            --> T  por convencion
» (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 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 =>))                         --> NIL
» (wff-infix-p '( B ^ C v D ))                  --> NIL
```

Pseudocódigo:

```
Entrada: x (expresión a evaluar)
Salida: T si está en formato infijo, NIL en caso contrario

Procesamiento:
wff-infix-p (x):
    si literal-p(x)
        devuelve T
    si listp(x)
        si unary-connector-p(first(x))
            devuelve wff-infix-p(second(x))
        si binary-connector-p(second(x))
            si null(fourth(x))
                devuelve and(wff-infix-p(first(x)),
                             wff-infix-p(third(x)))
            en caso contrario
                devuelve NIL
        si n-ary-connector-p(second(x))
            si null(fourth(x))
                devuelve and(wff-infix-p(first(x)),
```

```
                            wff-infix-p(third(x)))
            en caso contrario
                devuelve and(wff-infix-p(first(x)),
                            wff-infix-p(rest(rest(x))),
                            eql(second(x), fourth(x))
        en caso contrario
            devuelve NIL
    en caso contrario
        devuelve NIL
```

Código

```
(defun wff-infix-p (x)
  (unless (null x)
    (or (literal-p x)
        (and (listp x)
             (let ((first (first x))
                   (second (first (rest x)))
                   (rest_1 (rest (rest x))))
               (cond
                ((unary-connector-p first)
                 (and (null rest_1)
                      (wff-infix-p second)))
                ((binary-connector-p second)
                 (let ((rest_2 (rest rest_1)))
                   (and (null rest_2)
                        (wff-infix-p first)
                        (wff-infix-p (first rest_1)))))
                ((n-ary-connector-p second)
                 (let ((rest_2 (rest rest_1)))
                   (or (and (null rest_2)
                            (wff-infix-p first)
                            (wff-infix-p (first rest_1)))
                       (and (wff-infix-p first)
                            (wff-infix-p rest_1)
                            (eql second (first rest_2))))))
                ((n-ary-connector-p first) (null (second x)))
                (t NIL)))))))
```

Transformar una expresión infija a formato prefijo

Batería de pruebas:

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

Pseudocódigo:

```
Entrada: x (expresión a transformar)
Salida: wff (expresión original en formato prefijo)

Procesamiento:
infix-to-prefix (x):
    si literal-p(x)
        devuelve x
    en caso contrario
        si unary-connector-p(first(x))
            devuelve list(first(x),
                          infix-to-prefix(rest(x)))
        si binary-connector-p(second(x))
            devuelve list(second(x),
                          infix-to-prefix(first(x)),
                          infix-to-prefix(third(x)))
        si n-ary-connector(second(x))
            si null(fourth(x))
                devuelve list(second(x),
                              infix-to-prefix(first(x)),
                              infix-to-prefix(third(x))))
            en caso contrario
                rest <-- rest(infix-to-prefix(rest(rest(x))))
```

```
            devuelve append(list(second(x)),
                            list(infix-to-prefix(first(x))),
                            rest)
      en caso contrario
          devuelve NIL
```

Código

```
(defun infix-to-prefix (wff)
  (when (wff-infix-p wff)
    (if (literal-p wff)
        wff
      (let ((first (first wff))
            (second (second wff))
            (third (third wff)))
        (cond
         ((unary-connector-p first)
          (list first
                (infix-to-prefix second)))
         ((binary-connector-p second)
          (list second
                (infix-to-prefix first)
                (infix-to-prefix third)))
         ((and (n-ary-connector-p first) (null second)) wff)
         ((n-ary-connector-p second)
          (cond
            ((null (fourth wff))
             (list second
                   (infix-to-prefix first)
                   (infix-to-prefix third)))
            (t
             (let ((rest (infix-to-prefix (rest (rest wff)))))
               (append (list second)
                       (list (infix-to-prefix first))
                       (rest rest))))))
          (t NIL))))))
```

Determinar si una FBF es cláusula

Batería de pruebas:

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

Pseudocódigo:

```
Entrada: x (expresión a evaluar)
Salida: T si es una cláusula, NIL en caso contrario

Procesamiento:
clause-aux (x):
    si null(x)
        devuelve T
    en caso contrario
        devuelve and(literal-p(first(x)),
                     clause-aux(rest(x)))

clause-p (x):
    si and(wff-prefix-p(x), not(literal-p(x)))
        devuelve and(is-or(first(x))
                     clause-aux(rest(x)))
    en caso contrario
        devuelve NIL
```

Código

```
(defun clause-aux (lst)
  (if (null lst)
      T
    (and
     (literal-p (first lst))
     (clause-aux (rest lst)))))
```

```
(defun clause-p (wff)
  (when (wff-prefix-p wff)
    (when (not (literal-p wff))
      (and
       (eq (first wff) +or+)
       (clause-aux (rest wff))))))
```

Determinar si una FBF en prefijo está en FNC

Batería de pruebas:

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

Pseudocódigo:

```
Entrada: x (FBF en formato prefijo)
Salida: T si está en FCN, NIL en caso contrario

Procesamiento:
cnf-aux (x):
    si null(x)
        devuelve T
    en caso contrario
        devuelve and(clause-p(first(x)),
                     cnf-aux(rest(x)))

cnf-p (x):
    si and(wff-prefix-p(x), not(literal-p(x)))
        devuelve and(is-and(first(x))
                     clause-aux(rest(x)))
    en caso contrario
        devuelve NIL
```

Código

```
(defun cnf-aux (lst)
  (if (null lst)
      T
    (and
     (clause-p (first lst))
     (cnf-aux (rest lst)))))


(defun cnf-p (wff)
  (when (wff-prefix-p wff)
    (when (not (literal-p wff))
        (and
         (eq (first wff) +and+)
         (cnf-aux (rest wff))))))
```

4.2. Algoritmo de transformación de una FBF a FNC

Eliminar conectores condicionales en una FBF

Batería de pruebas:

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

Pseudocódigo:

```
Entrada: x (FBF en formato prefijo)
Salida: wff (expresión original sin conectores condicionales)

Procesamiento:
eliminate-conditional (x):
    si or(null(x), literal-p(x))
        devuelve x
```

```
    en caso contrario
        si is-cond(first(x))
            wff1 <-- eliminate-conditional(second(x))
            wff2 <-- eliminate-conditional(third(x))
            devuelve list('or',
                          not(wff1),
                          wff2)
        en caso contrario
            devuelve cons(first(x),
                          eliminate-conditional(rest(x)))
```

Código

```
(defun eliminate-conditional (wff)
  (if (or (null wff) (literal-p wff))
      wff
    (let ((connector (first wff)))
      (if (eq connector +cond+)
          (let ((wff1 (eliminate-conditional (second wff)))
                (wff2 (eliminate-conditional (third  wff))))
            (list +or+ ;; Descompone (A => B) en (~A v B)
                  (list +not+ wff1)
                  wff2))
        (cons connector ;; Si no hay =>
              (mapcar #'eliminate-conditional (rest wff)))))))
```

Reducir ámbito de negación en una FBF

Batería de pruebas:

```
» (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))
```

Pseudocódigo:

```
Entrada: x (FBF en formato prefijo)
Salida: wff (expresión original con negación sólo en literales)
```

```
Procesamiento:
reduce-scope-of-negation (x):
    si null(x)
        devuelve NIL
    si literal-p(x)
        devuelve x
    si unary-connector-p(first(x))
        devuelve invert(second(x))
    en caso contrario
        devuelve cons(first(x),
                      reduce-scope-of-negation(rest(x)))
```

Código

```
(defun invert (lst)
  (cond
   ((null lst) ;; Condicion de parada de conectores n-arios
    NIL)
   ((positive-literal-p lst) ;; Caso base con literal positivo
    (list +not+ lst))
   ((negative-literal-p lst) ;; Caso base con literal negativo
    (second lst))
   ((n-ary-connector-p lst) ;; Caso con conector n-ario
    (exchange-and-or lst))
   ((unary-connector-p (first lst)) ;; Expresion negada
    (second lst))
   (t
    (cons (invert (first lst)) (invert (rest lst))))))

(defun reduce-scope-of-negation-aux (lst)
  (cond ((null lst)
         NIL)
        ((literal-p lst)
         lst)
        ((unary-connector-p (first lst))
         (invert (second lst)))
        (t
```

```
        (cons (first lst) (mapcar
                             #'reduce-scope-of-negation-aux
                           (rest lst))))))

(defun reduce-scope-of-negation (wff)
  (when (wff-prefix-p wff)
    (reduce-scope-of-negation-aux wff)))
```

Eliminar conectores conjunción y disyunción en una FBF en FNC

Batería de pruebas:

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

Pseudocódigo:

```
Entrada: x (FNC en formato prefijo)
Salida: cnf (expresión original sin conectores and y or)

Procesamiento:
eliminate-connectors (x):
    si null(x)
        devuelve NIL
    si literal-p(x)
        devuelve x
    si n-ary-connector-p(first(x))
        devuelve eliminate-connectors(rest(x))
    en caso contrario
        devuelve cons(eliminate-connectors-aux(first(x)),
                      eliminate-connectors-aux(rest(x)))
```

Código

```
(defun eliminate-connectors-aux (lst)
  (cond
   ((null lst)
```

```
    NIL)
   ((literal-p lst)
    lst)
   ((n-ary-connector-p (first lst))
    (eliminate-connectors-aux (rest lst)))
   (t
    (cons (eliminate-connectors-aux (first lst))
          (eliminate-connectors-aux (rest lst))))))

(defun eliminate-connectors (cnf)
  (when (wff-prefix-p cnf)
    (eliminate-connectors-aux cnf)))
```

Transformar una FBF en notación infija a FNC con conjunciones y disyunciones implícitas

Batería de pruebas:

```
» (wff-infix-to-cnf 'a)
--> ((A))
» (wff-infix-to-cnf '(~ a))
--> (((~ A)))
» (wff-infix-to-cnf  '( (~ p) v q v (~ r) v (~ s)))
--> (((~ P) Q (~ R) (~ S)))
» (wff-infix-to-cnf  '((p v (a => (b ^ (~ c) ^ d))) ^ ((p <=> (~ q)) ^ p)))
--> ((P (~ A) B) (P (~ A) (~ C)) (P (~ A) D) ((~ P) (~ Q)) (Q P) (P))
```

Pseudocódigo:

```
Entrada: x (FBF en formato infijo)
Salida: cnf (expresión original en FNC sin conectores)

Procesamiento:
wff-infix-to-cnf (x):
    si wff-infix-p(x)
        aux1 <-- eliminate-biconditional(infix-to-prefix(x))
        aux2 <-- reduce-scope-of-negation(eliminate-conditional(aux1))
        devuelve eliminate-connectors(cnf(aux2))
    en caso contrario
```

```
        devuelve NIL
```

Código

```
(defun wff-infix-to-cnf (wff)
  (when (wff-infix-p wff)
    (eliminate-connectors
     (cnf
      (reduce-scope-of-negation
       (eliminate-conditional
        (eliminate-biconditional (infix-to-prefix wff))))))))
```

4.3. Simplificación de FBFs en FNC

Eliminar literales repetidos en una cláusula

Batería de pruebas:

```
» (eliminate-repeated-literals '(a b (~ c) (~ a) a c (~ c) c a))
--> (B (~ A) (~ C) C A)
```

Pseudocódigo:

```
Entrada: x (FBF sin conectores)
Salida: clause (cláusula original sin literales repetidos)

Procesamiento:
eliminate-repeated-literals (x):
    si literal-p(first(x))
        si member(first(x), rest(x))
            devuelve eliminate-repeated-literals(rest(x))
        en caso contrario
            devuelve cons(first(x),
                          eliminate-repeated-literals(rest(x)))
    en caso contrario
        devuelve NIL
```

Código

```
(defun clause-lst-p (lst)
  (if (null lst)
      T
    (when (list lst)
      (when (literal-p (first lst))
        (if (null (rest lst))
            T
          (clause-aux (rest lst)))))))

(defun eliminate-repeated-literals-aux (lst)
  (let ((first (first lst))
          (rest (rest lst)))
      (when (literal-p first)
        (if (member first rest :test #'equal)
            (eliminate-repeated-literals-aux rest)
          (cons first
                (eliminate-repeated-literals-aux rest))))))

(defun eliminate-repeated-literals (k)
  (when (clause-lst-p k)
    (eliminate-repeated-literals-aux k)))
```

Eliminar cláusulas repetidas en una FNC

Batería de pruebas:

```
» (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))
```

Pseudocódigo:

```
Entrada: x (FNC sin conectores)
Salida: cnf (FNC original sin cláusulas repetidas)

Procesamiento:
eliminate-repeated-clauses (x):
    si null(x)
        return NIL
```

```
    si member(first(x), rest(x))
        devuelve eliminate-repeated-clauses(rest(x))
    en caso contrario
        devuelve cons(first(x),
        eliminate-repeated-clauses(rest(x)))
```

Código

```
(defun cnf-lst-p (lst)
  (when (list lst)
    (when (clause-lst-p (first lst))
      (if (null (rest lst))
          T
        (cnf-lst-p (rest lst))))))

(defun contain-clause-aux (c1 c2)
  (if (null c1)
      T
    (let ((elmnt (find (first c1) c2 :test #'equal)))
      (if elmnt
          (contain-clause-aux (rest c1) c2)
        NIL))))

(defun contain-clause (c1 c2)
  (if (null c1)
      (list NIL)
    (when (and (clause-lst-p c1) (clause-lst-p c2))
      (if (contain-clause-aux c1 c2)
          T
        NIL))))

(defun equal-clause (c1 c2)
  (when (and (clause-lst-p c1) (clause-lst-p c2))
    (let ((c1-aux (eliminate-repeated-literals c1))
          (c2-aux (eliminate-repeated-literals c2)))
      (and (contain-clause c1-aux c2-aux)
           (contain-clause c2-aux c1-aux)))))
```

```
(defun eliminate-repeated-clauses-aux (lst)
  (if (null lst)
      NIL
    (when (clause-lst-p (first lst))
      (let ((first (eliminate-repeated-literals (first lst)))
            (rest (rest lst)))
        (if (member first rest :test #'equal-clause)
            (eliminate-repeated-clauses-aux rest)
          (cons first
                (eliminate-repeated-clauses-aux rest)))))))

(defun eliminate-repeated-clauses (cnf)
  (when (cnf-lst-p cnf)
    (eliminate-repeated-clauses-aux cnf)))
```

Determinar si una cláusula subsume a otra

Batería de pruebas:

```
» (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)))
```

Pseudocódigo:

```
Entrada: K1 (cláusula), K2 (cláusula)
Salida: K1 si K1 subsume a K2, NIL en caso contrario

Procesamiento:
subsume-aux (K1, K2):
    si null(K1)
        devuelve T
    en caso contrario
        elmnt <-- find(first(K1), K2)
        si not(null(elmnt)
            devuelve subsume-aux(rest(K1), K2)
        en caso contrario
            devuelve NIL
```

```
subsume (K1, K2):
    si subsume-aux(K1, K2)
        devuelve K1
    en caso contrario
        devuelve NIL
```

Código

```
(defun subsume-aux (K1 K2)
  (if (null K1)
      T
    (let ((elmnt (find (first K1) K2 :test #'equal)))
      (if elmnt
          (subsume-aux (rest K1) K2)
        NIL))))

(defun subsume (K1 K2)
  (when (and (clause-lst-p K1) (clause-lst-p K2))
    (if (subsume-aux K1 K2)
        (list K1)
      NIL)))
```

Eliminar cláusulas subsumidas en una FNC

Batería de pruebas:

```
» (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))
```

Pseudocódigo:

```
Entrada: x (expresión en FNC)
Salida: cnf (expresión equivalente sin cláusulas subsumidas)

Procesamiento:
eliminate-subsumed-clauses-aux (aux, cnf):
    si null(aux)
        devuelve NIL
    en caso contrario
        para cada clause en cnf:
            si subsume(clause, first(aux))
                devuelve eliminate-subsumed-clauses-aux(rest(aux), cnf)
            en caso contrario
                devuelve cons(first(aux),
                              eliminate-subsumed-clauses-aux(rest(aux), cnf)

eliminate-subsumed-clauses (x):
    devuelve eliminate-subsumed-clauses-aux(x, x)
```

Código

```
(defun subsume-lst (lst clause)
  (when lst
    (if (equal (first lst) 'NIL)
        T
      (or (and (not (eq (first lst) clause))
               (not (null (subsume (first lst) clause))))
        (subsume-lst (rest lst) clause)))))

(defun eliminate-subsumed-clauses-aux (aux cnf)
  (if (null aux)
      NIL
    (if (subsume-lst cnf (first aux))
        (eliminate-subsumed-clauses-aux (rest aux) cnf)
      (cons (first aux)
            (eliminate-subsumed-clauses-aux (rest aux) cnf)))))

(defun eliminate-subsumed-clauses (cnf)
  (when (cnf-lst-p cnf)
```

```
    (eliminate-subsumed-clauses-aux cnf cnf)))
```

Determinar si una cláusula es tautología

Batería de pruebas:

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

Pseudocódigo:

```
Entrada: x (cláusula)
Salida: T si es tautología, NIL en caso contrario

Procesamiento:
tautology-p (x):
    si null(x)
        devuelve NIL
    en caso contrario
        conj <-- conjugado(first(x))
        devuelve or(member(conj, rest(x)),
                    tautology-p(rest(x)))
```

Código

```
(defun conjugado (lit1 lit2)
  (when (and (literal-p lit1) (literal-p lit2))
    (or (and (positive-literal-p lit1)
             (negative-literal-p lit2)
             (equal lit1 (second lit2)))
        (and (negative-literal-p lit1)
             (positive-literal-p lit2)
             (equal (second lit1) lit2)))))

(defun tautology-p-aux (lst)
  (unless (null lst)
    (or (member (first lst) (rest lst) :test #'conjugado)
        (not (null (tautology-p-aux (rest lst)))))))
```

```
(defun tautology-p (K)
  (when (clause-lst-p K)
    (tautology-p-aux K)))
```

Eliminar tautologías

Batería de pruebas:

```
» (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
```

Pseudocódigo:

```
Entrada: x (expresión en FNC)
Salida: cnf (expresión equivalente sin tautologías)

Procesamiento:
eliminate-tautologies (x):
    si null(x)
        devuelve NIL
    en caso contrario
        si tautology-p(first(x))
            devuelve eliminate-tautologies(rest(x))
        en caso contrario
            devuelve cons(first(x),
                          eliminate-tautologies(rest(x)))
```

Código

```
(defun eliminate-tautologies-aux (lst)
  (if (null lst)
      NIL
    (if (tautology-p (first lst))
        (eliminate-tautologies-aux (rest lst))
      (cons (first lst)
            (eliminate-tautologies-aux (rest lst))))))
```

```
(defun eliminate-tautologies (cnf)
  (when (cnf-lst-p cnf)
    (eliminate-tautologies-aux cnf)))
```

Transformación de FBF a FNC

Batería de pruebas:

```
» (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)) el orden no es importante
```

Pseudocódigo:

```
Entrada: x (FBF)
Salida: cnf (expresión equivalente en FNC)

Procesamiento:
simplify-cnf (x):
    aux <-- eliminate-repeated-clauses(eliminate-repeated-literals(x))
    devuelve eliminate-subsumed-clauses(eliminate-tautologies(aux))
```

Código

```
(defun simplify-cnf-aux (lst)
  (eliminate-subsumed-clauses
   (eliminate-tautologies
    (eliminate-repeated-clauses
     (mapcar #'eliminate-repeated-literals lst)))))

(defun simplify-cnf (cnf)
  (when (cnf-lst-p cnf)
    (simplify-cnf-aux cnf)))
```

4.4. Construcción de RES

Extraer cláusulas neutras

Batería de pruebas:

```
» (extract-neutral-clauses 'r NIL)
--> NIL
» (extract-neutral-clauses 'r '(NIL))
--> (NIL)
» (extract-neutral-clauses 'p
                           '((p (~ q) r) (r (~ s) q) (a (~ p) c) ((~ r) s)))
--> ((R (~ S) Q) ((~ R) S))
» (extract-neutral-clauses 'p
                           '((p (~ q) r) (r (~ s) p q) (a (~ p) c) ((~ r) p s)))
--> NIL
```

Pseudocódigo:

```
Entrada: lambda (literal positivo), cnf (expresión en FNC)
Salida: cnf-lambda (cláusulas de cnf que no contienen lambda ni ~lambda)

Procesamiento:
extract-neutral-clauses (lambda, cnf):
    si null(cnf)
        devuelve NIL
    conj <-- conjugado(lambda)
    si or(member(lambda, first(cnf)), member(conj, first(cnf)))
        devuelve extract-neutral-clauses(lambda, rest(cnf))
    en caso contrario
        devuelve cons(first(cnf),
                      extract-neutral-clauses(lambda, rest(cnf)))
```

Código

```
(defun equal-literal (lit1 lit2)
  (or (equal lit1 lit2) (conjugado lit1 lit2)))

(defun extract-neutral-clauses-aux (lit lst)
  (if (null lst)
      NIL
    (when (clause-lst-p (first lst))
      (if (member lit (first lst) :test #'equal-literal)
          (extract-neutral-clauses-aux lit (rest lst))
```

```
        (cons (first lst)
              (extract-neutral-clauses-aux lit (rest lst)))))))

(defun extract-neutral-clauses (lambda cnf)
  (when (and (positive-literal-p lambda) (cnf-lst-p cnf))
    (extract-neutral-clauses-aux lambda cnf)))
```

Extraer cláusulas positivas

Batería de pruebas:

```
» (extract-positive-clauses 'r NIL)
--> NIL
» (extract-positive-clauses 'r '(NIL))
--> NIL
» (extract-positive-clauses 'p
                            '((p (~ q) r) (p q) (r (~ s) q) (a b p) (r)))

--> ((P (~ Q) R) (P Q) (A B P))

» (extract-positive-clauses 'p
                            '(((~ p) (~ q) r) ((~ p) q) (a b (~ p)) ((~ p))))
--> NIL
```

Pseudocódigo:

```
Entrada: lambda (literal positivo), cnf (expresión en FNC)
Salida: cnf-lambda (cláusulas de cnf que no contienen lambda)

Procesamiento:
extract-positive-clauses (lambda, cnf):
    si null(cnf)
        devuelve NIL
    si member(lambda, first(cnf))
        devuelve cons(first(cnf),
                      extract-positive-clauses(lambda, rest(cnf)))
    en caso contrario
        devuelve extract-positive-clauses(lambda, rest(cnf))
```

Código

```
(defun extract-lit-clauses-aux (lit lst)
  (if (null lst)
      NIL
    (when (clause-lst-p (first lst))
      (if (member lit (first lst) :test #'equal)
          (cons (first lst) (extract-lit-clauses-aux lit (rest lst)))
      (extract-lit-clauses-aux lit (rest lst))))))

(defun extract-positive-clauses (lambda cnf)
  (when (and (positive-literal-p lambda) (cnf-lst-p cnf))
    (extract-lit-clauses-aux lambda cnf)))
```

Extraer cláusulas negativas

Batería de pruebas:

```
» (extract-negative-clauses 'r NIL)
--> NIL
» (extract-negative-clauses 'r '(NIL))
--> NIL
» (extract-negative-clauses 'p
                            '((p (~ q) r) (p q) (r (~ s) q) (a (~ p) c)))
--> ((A (~ P) C))
» (extract-negative-clauses 'p
                            '(( p (~ q) r) ( p q) (r (~ s) p q)))
--> NIL
```

Pseudocódigo:

```
Entrada: lambda (literal positivo), cnf (expresión en FNC)
Salida: cnf-lambda (cláusulas de cnf que no contienen ~lambda)

Procesamiento:
extract-negative-clauses (lambda, cnf):
    si null(cnf)
        devuelve NIL
    conj <-- conjugado(lambda)
```

```
    si member(conj, first(cnf))
        devuelve cons(first(cnf),
                      extract-negative-clauses(lambda, rest(cnf)))
    en caso contrario
        devuelve extract-negative-clauses(lambda, rest(cnf))
```

Código

```
(defun extract-negative-clauses (lambda cnf)
  (when (and (positive-literal-p lambda) (cnf-lst-p cnf))
    (extract-lit-clauses-aux (list +not+ lambda) cnf)))
```

Resolver entre dos cláusulas

Batería de pruebas:

```
» (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
```

Pseudocódigo:

```
Entrada: lambda (literal positivo), K1 (cláusula), K2 (cláusula)
Salida: res (cláusula resolución de K1 y K2 con lambda)

Procesamiento:
resolve-on (lambda, K1, K2):
    si or(null(K1), null(K2))
        devuelve NIL
    en caso contrario
        pos1 <-- extract-positive-clauses(lambda, list(K1))
        neg1 <-- extract-negative-clauses(lambda, list(K2))
```

```
        pos2 <-- extract-positive-clauses(lambda, list(K2))
        neg2 <-- extract-negative-clauses(lambda, list(K1))
        si and(not(null(pos1)), not(null(neg1)))
            devuelve union(pos1, neg1)
        si and(not(null(pos2)), not(null(neg2)))
             devuelve union(pos2, neg2)
        en caso contrario
            devuelve NIL
```

Código

```
(defun union-aux (LL1 LL2 lambda flag)
  (let ((L1 (copy-list (first LL1)))
        (L2 (copy-list (first LL2)))
        (lambda-conj (invert lambda)))
  (if (= flag 1)
      (list (union (remove lambda L1) (remove lambda-conj L2 :test #'equal)))
    (list (union (remove lambda-conj L1 :test #'equal) (remove lambda L2))))))

(defun resolve-on (lambda K1 K2)
  (if (or (null K1) (null K2))
      '()
    (let ((pos1 (extract-positive-clauses lambda (list K1)))
          (neg1 (extract-negative-clauses lambda (list K2)))
          (pos2 (extract-positive-clauses lambda (list K2)))
          (neg2 (extract-negative-clauses lambda (list K1))))
      (cond
       ((and pos1 neg1)
        (union-aux pos1 neg1 lambda 1)) ;res on lambda:lambda-conj
       ((and pos2 neg2)
        (union-aux neg2 pos2 lambda 2)) ;res on lambda-conj:lambda
       (T NIL))))) ;Solo llega aqui si no es posible la resolucion
```

Construir el conjunto RES de una FNC

Batería de pruebas:

```
» (build-RES 'p NIL)
--> NIL
```

```
» (build-RES 'p '(NIL))
--> (NIL)
» (build-RES 'P '((A  (~ P) B) (A P) (A B)))
--> ((A B))
» (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))
```

Pseudocódigo:

```
Entrada: lambda (literal positivo), cnf (FBF en FNC simplificada)
Salida: res (expresión con todas las resoluciones sobre lambda)

Procesamiento:
build-RES (lambda, cnf):
    si null(cnf)
        devuelve NIL
    en caso contrario
        pos <-- extract-positive-clauses(lambda, cnf)
        neg <-- extract-negative-clauses(lambda, cnf)
        res <-- extract-neutral-clauses(lambda, cnf)
        para cada x en pos:
            para cada y en neg:
                append(res, resolve-on(lambda, x, y))
        devuelve res
```

Código

```
(defun build-RES (lambda cnf)
  (if (null cnf)
      NIL
    (let ((pos (extract-positive-clauses lambda cnf))
          (neg (extract-negative-clauses lambda cnf))
          (neu (extract-neutral-clauses lambda cnf)))
      (eliminate-repeated-clauses
```

```
        (append
         neu
         (mapcan #'(lambda(x)
                     (mapcan #'(lambda (y) (resolve-on lambda x y)) neg))
           pos))))))
```

4.5. Algoritmo para determinar si una FNC es SAT

Batería de pruebas

```
SAT Examples:
» (RES-SAT-p nil)            --> T
» (RES-SAT-p '((p) ((~ q)))) --> T
» (RES-SAT-p
   '((a b d) ((~ p) q) ((~ c) a b) ((~ b) (~ p) d) (c d (~ a))))   --> T
» (RES-SAT-p
   '(((~ p) (~ q) (~ r)) (q r) ((~ q) p) ((~ q)) ((~ p) (~ q) r))) --> T

UNSAT Examples:
» (RES-SAT-p '(nil))         --> NIL
» (RES-SAT-p '((S) nil))     --> NIL
» (RES-SAT-p '((p) ((~ p)))) --> NIL
» (RES-SAT-p
   '(((~ p) (~ q) (~ r)) (q r) ((~ q) p) (p) (q) ((~ r)) ((~ p) (~ q) r)))
--> NIL
```

Pseudocódigo

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

Procesamiento:
RES-SAT-p (cnf):
   si null(cnf)
       devuelve T
   si member(nil, cnf)
       devuelve NIL
   para cada atom en atoms(cnf):
```

```
        append(list, build-RES(atom, cnf))
    si member('(NIL)', list)
        devuelve NIL
    new-knowlg <-- simplify-cnf(list)
    si null(new-knowlg)
        devuelve NIL
    si same-list(cnf, new-knowlg)
        devuelve T
    en caso contrario
        devuelve RES-SAT-p(union(new-knowlg, cnf))
```

Código

```
(defun check-same-cl (K1 K2) ;Comprueba si dos clausulas son equivalentes
  (if (and (null K1) (null K2))
      T
    (if (member (first K1) K2 :test #'equal)
        (check-same-cl (rest K1)
                       (remove
                        (first K1)
                        (copy-list K2)
                        :count 1 :test #'equal))
      NIL)))

(defun check-same-list (l1 l2) ;Comprueba si dos listas tienen las mismas
  (if (null l2)                ;cláusulas, o si la segunda está contenida
      T                        ;en la primera
    (if (member (first l2) l1 :test #'check-same-cl)
        (check-same-list (remove
                          (first l2)
                          (copy-list l1)
                          :count 1 :test #'check-same-cl)
                         (rest l2))
      NIL)))

(defun extract-atoms-aux (list)
  (if (null list)
      NIL
```

```
    (if (positive-literal-p (first list))
        (cons (first list) (extract-atoms-aux (rest list)))
      (extract-atoms-aux (rest list)))))

(defun extract-atoms (lol)
  (if (null lol)
      NIL
    (eliminate-repeated-literals (mapcan #'extract-atoms-aux lol))))

(defun  RES-SAT-p (cnf)
  (if (null cnf)
      T
    (if (member nil cnf)
        NIL
      (let ((lista (mapcar #'(lambda (x)
                               (build-RES x cnf))
                           (extract-atoms cnf))))
        (if (member '(nil) lista :test #'equal) ;Si hemos derivado
            NIL                                 ;la cláusula vacía
          (let ((l-new-knowlg (simplify-cnf (apply #'append lista))))
            (if (null l-new-knowlg) ;Cláusula vacía
                NIL
              (if (check-same-list cnf l-new-knowlg) ;Si no hemos adquirido
                  T                                  ;nuevo conocimiento
                (RES-SAT-p (union ;Si hemos adquirido nuevo conocimiento
                            l-new-knowlg
                            cnf
                            :test #'check-same-cl)))))))))
```

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

Batería de pruebas

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

Pseudocódigo

```
Entrada: wff (FBF en formato infijo), w (FBF en formato infijo)
Salida: T si w es consecuencia lógica de wff, NIL en caso contrario

Procesamiento:
logical-consequence-RES-SAT-p (wff, w):
    si or(null(wff), null(w))
        devuelve NIL
    aux1 <-- simplify-cnf(wff-infix-to-cnf(prefix-to-infix
                          (reduce-scope-of-negation
                           (not(eliminate-conditional
                            (eliminate-biconditional
                             (infix-to-prefix(w))))))))
    aux2 <-- simplify-cnf(wff-infix-to-cnf (wff))
    fnc <-- append(aux1, aux2)
    si null(RES-SAT-p(fnc))
        devuelve T
    en caso contrario
        devuelve NIL
```

Código

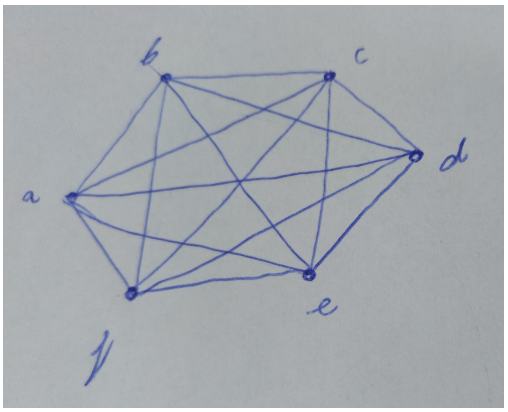
```
(defun logical-consequence-RES-SAT-p (wff w)
  (if (or (null wff) (eq w nil))
      NIL
    (let ((fnc (append           ;Incluimos la negación de la meta
```

```
            (simplify-cnf    ;en la base de conocimiento
             (wff-infix-to-cnf
              (prefix-to-infix
               (reduce-scope-of-negation
                (list +not+ (eliminate-conditional
                             (eliminate-biconditional
                              (infix-to-prefix w))))))))
            (simplify-cnf
             (wff-infix-to-cnf wff)))))
(if (eq (RES-SAT-p fnc) nil) ;Si derivamos la cláusula vacía con
    T                        ;w negado, es consec. lógica
  NIL))))                    ;si no, no
```

5. Búsqueda en anchura

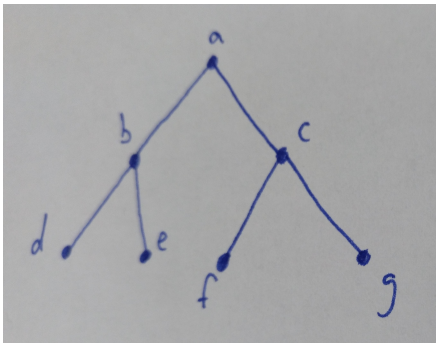
5.1. Ilustrar el funcionamiento del algoritmo

Ejemplo 1:



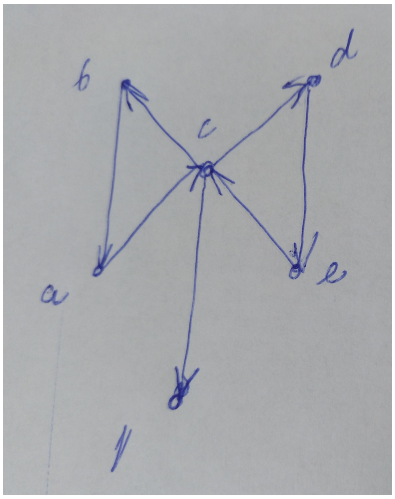

```
--------------------------------
Origen: a
Meta: f
Procedimiento:
  Explora a, descubre b, c, d,
  e y f.
  Explora b, c, d, y e.
  Devuelve a->f.
--------------------------------
```

Ejemplo 2:




```
--------------------------------
Origen: a
Meta: f
Procedimiento:
  Explora a, descubre b y c.
  Explora b, descubre d y e.
  Explora c, descubre f y g.
  Explora d y e.
  Devuelve a->c->f.
--------------------------------
```

Ejemplo 3:




```
--------------------------------
Origen: b
Meta: f
Procedimiento:
  Explora b, descubre a.
  Explora a, descubre c.
  Explora c, descubre d y f.
  Explora d, descubre e.
  Devuelve b->a->c->f.
--------------------------------
```

5.2. Pseudocódigo correspondiente al algoritmo BFS

```
Entrada : G (grafo), s (nodo origen), d (nodo destino)
Salida: camino (si lo hay) más corto entre s y d

Procedimiento:
BFS (G, s, d):
    para cada u en vertices(G):
        estado[u] <-- NO_VISITADO
    estado[s] <-- VISITADO
```

```
push(camino[s], s)        ; Siendo camino[x] una lista
push(Q, s)                ; Siendo Q una cola
mientras vacia(Q) == F:
    u <-- pop(Q)
    si u == d
        devuelve camino[u]
    para cada v en adyacentes(u, G):
       si estado[v] == NO_VISITADO
           estado[v] <-- VISITADO
           camino[v] <-- camino[u]
           push(camino[v], v)
           push(Q, v)
```

5.3. Comentarios sobre el código

Comentarios sobre bfs:

- Si no hay elementos en queue, no es posible encontrar la meta.
- Cada elemento de queue es un camino desde un nodo al origen.
- Cuando encontramos un camino en queue que vaya de la meta al origen, devolvemos el camino inverso.
- Cuando el nodo no es meta, encolamos queue con los caminos de los nodos adyacentes y repetimos el proceso.

Comentarios sobre new-paths:

- Devuelve los caminos de los nodos adyacentes a node.

5.4. ¿Por qué encuentra el camino más corto entre dos nodos?

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

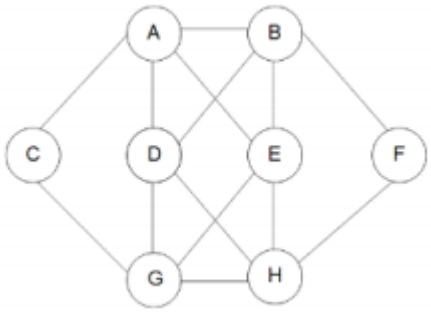
Encuentra el camino más corto porque parte de un camino que sólo contiene al nodo origen. Esto garantiza que se exploren primero los nodos a distancia N-1 del origen que los de distancia N. Así, la primera vez que expandamos la meta estaremos seguros de que el camino que nos ha llevado hasta ella es óptimo.

5.5. Ilustrar el funcionamiento del código

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

Traza:
(bfs 'f '((a)) '((a d) (b d f) (c e) (d f) (e b f) (f)))
   (new-paths '(a) 'a '((a d) (b d f) (c e) (d f) (e b f) (f))) -> ((d a))
(bfs 'f '((d a)) '((a d) (b d f) (c e) (d f) (e b f) (f)))
   (new-paths '(d a) 'd '((a d) (b d f) (c e) (d f) (e b f) (f))) -> ((f d a))
(bfs 'f '((f d a)) '((a d) (b d f) (c e) (d f) (e b f) (f)))
   (eql node end) -> T, devuelve (reverse path)=(a d f)
```

5.6. Encontrar el camino más corto entre los nodos F y C




```
» (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)))
--> (F B A C)
```

5.7. Corregir el código

Falla porque no etiqueta los nodos como visitados, por ejemplo en ((a d) (b d f) (c e) (d f) (e b f) (f a)), la llamada (shortest-path 'a 'b '((a d) (b d f) (c e) (d f) (e b f) (f a)))

Error: Stack overflow (signal 1000) [condition type: SYNCHRONOUS-OPERATING-SYSTEM-SIGNAL]

Código mejorado:

```
(defun bfs-improved (end queue net)
  (if (null queue) '()
    (let* ((path (first queue))
           (node (first path)))
      (if (eql node end)
          (reverse path)
        (if (member node (rest path)) ;De esta manera evitamos los bucles
            '()
          (bfs-improved end
                        (append (rest queue)
                                (new-paths path node net))
                        net))))))

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