PRACTICA 1: LISP

Blanca Abella Miravet
María Barroso Honrubia
Pareja 6

Inteligencia Artificial
Grupo 2301

1. Similitud Coseno

1.1

Código de funciones utilizadas para ambas funciones:

```
;;; no-vacia (lista)
;;; Funcion que comprueba que un vector no es vacío
;;; INPUT: lista: vector, representado como una lista
;;; OUTPUT: T si la lista no es vacia, nil en caso contrario

(defun no-vacia (lista)
	(not (eql nil lista)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; no-cero (lista)
;;; Funcion que comprueba que una vector no es el vector cero
;;; INPUT: lista: vector, representado como una lista
;;; OUTPUT: t si la lista no es el vector 0, nil en caso contrario

(defun no-cero (lista)
	(not (every #'zerop lista)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; is-ok1
;;; Funcion que comprueba que los argumentos que recibe sc-rec son correctos (no son vectores vacíos ni son el vector cero)
;;; INPUT: x: primer vector a comprobar parametros
;;; y: segundo vector a comprobar parametros
;;; OUTPUT: t si los argumentos son correctos, nil en caso contrario

(defun is-ok1 (x y)
	(and (no-vacia x) (no-vacia y) (no-cero x) (no-cero y)))
```

- **Recursiva**

  Batería de ejemplos:

```
;;; (sc-rec nil '(1 2 3)) ;; --> nil
;;; (sc-rec '(0 0) '(7 3)) ;; --> nil
;;; (sc-rec '(1 2 3) '(4 1 7)) ;; --> 0.88823473
```

Pseudocódigo:

Entrada: x y (vectores positivos de igual longitud)
Salida: n (similitud coseno de x y)
Procesamiento:
	Si x o y es vector vacío o vector cero
		retorna nil

en caso contrario

evalua a $\frac{prod\text{−}esc\text{−}rec\ (x\ y)}{(modulo\text{−}rec\ (x))\cdot(\ modulo\text{−}rec\ (y))}$

Código:

```
;;; prod-esc-rec (x y)
;;; Funcion que 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 de los dos vectores

(defun prod-esc-rec (x y)
	(if (null (rest x))  ; caso base
	(* (first x) (first y)) ; multiplica los primeros elementos de los vectores x y
	(+ (* (first x) (first y)) (prod-esc-rec (rest x) (rest y))))) ; sumatorio de todas estas
multiplicaciones
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; modulo-rec (x)
;;; Función que calcula el módulo de un vector
;;; INPUT: x: vector, representado como una lista
;;; OUTPUT: raiz cuadrada del producto escalar de x x (definicion de modulo de x)

(defun modulo-rec (x)
	(sqrt (prod-esc-rec x x))) ; raíz cuadrada del producto escalar de x x
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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 vacías o que son
;;; (0 0...0) es NIL.
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;; OUTPUT: similitud coseno entre x e y

(defun sc-rec (x y)
	(if (null (is-ok1 x y))  ; condición de error
	nil
	(/ (prod-esc-rec x y) (* (modulo-rec x) (modulo-rec y))))) ; aplica prod-esc-rec (x
y)/(modulo-rec (x))*(modulo-rec (y))
```

- **Iterativa**

Batería de ejemplos

```
;;; (sc-mapcar nil '(1 2 3)) ;; --> nil
;;; (sc-mapcar '(0 0) '(7 3)) ;; --> nil
;;; (sc-mapcar '(1 2 3) '(4 1 7)) ; --> 0.88823473
```

Pseudocódigo:

Entrada: x y (vectores positivos de igual longitud)
Salida: n (similitud coseno de x y)
Si x o y es vector vacío o vector cero
    retorna nil
    n = <x,y> / (|x| * |y|)

Código:

```
;;; sumatorio (x)
;;; Funcion que calcula el sumatorio de los numeros de un vector
;;; INPUT: x: vector, representado como una lista
;;; OUTPUT: sumatorio de todos los elementos de x

(defun sumatorio (x)
   (reduce #'+ x)) ; aplica la funcion suma a los elementos de x
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; prod-escalar (x y)
;;; Funcion que calcula el producto escalar de dos vectores
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;; OUTPUT: sumatorio de la lista formada por el producto de los elementos de x e y

(defun prod-escalar (x y)
   (sumatorio (mapcar #'(lambda (z w) (* z w)) x y))) ; sumatorio del producto de los
elementos de dos vectores
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; modulo (x)
;;; Funcion que calcula el modulo de un vector
;;; INPUT: x, vector, representado como una lista
;;; OUTPUT: raiz cuadrada del producto escalar de x y x (definicion del modulo de x)

(defun modulo (x)
   (sqrt (prod-escalar x x))) ; aplica la raiz cuadrada del producto escalar de x x
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-mapcar (x y)
;;; Calcula la similitud coseno de un vector usando mapcar
;;;
```

```
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: similitud coseno entre x e y

(defun sc-mapcar (x y)
   (if (null (is-ok1 x y)) ; condicion de error
         nil
         (/ (prod-escalar x y) (* (modulo x) (modulo y))))) ; aplica la definion de la similitud
coseno
```

1.2

Batería de ejemplos:

```
;;; (sc-conf '(1 43 23 12) '((1 3 22 134) nil) 0.2) ;; --> nil
;;; (sc-conf '(2 33 54 24) '((1 3 22 134) (1 3 22 134)) 0.2) ;; --> ((1 3 22 134) (1 3 22
134))
;;; (sc-conf '(1 2 3) '((4 2 1) (1 2 3)) 0.5) ;; --> ((1 2 3) (4 2 1))
;;; (sc-conf '(1 43 23 12) nil 0.2) ;; --> nil
;;; (sc-conf '(0 0 0 0) '((1 43 23 12)) 0.2) ;; --> nil
;;; (sc-conf '(1 2 3 4) '((1 43 23 12)) 4) ;; --> nil
```

Pseudocódigo:

```
Entrada: cat vs conf
Salida: n (vectores similares a una categoria)
Procesamiento:
Si cat o vs o y son vector vacio, el vector cero o conf no se encuentra entre [0,1]
        devuelve nil
        para todo x de vs, si (sc-rec cat x < cnf)
                eliminar x de vs
        n = ordenar vs por sc-rec
```

Código:

```
;;; limpia-lista (cat vs conf)
;;; Funcion que elimina de lista de lista aquellos vectores cuya similitud sea menor al
nivel de confianza
;;; INPUT: cat: vector que representa a una categoría, representado como una lista
;;; vs: vector de vectores
;;; conf: Nivel de confianza
;;; OUTPUT: nueva lista vs que no contiene aquellos elementos cuya similitud coseno
con los vectores de cat sea menor que el nivel de confianza conf

(defun limpia-lista (cat vs conf)
```

```
   (remove-if #'(lambda (y) (< (abs (sc-rec cat y)) conf)) vs)) ; elimina de vs aquellos
vectores que al aplicarle similitud coseno con los vectores de vs sea menor que el nivel
de confianza establecido
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; is-ok2 (cat vs conf)
;;; Función que comprueba que todos los vectores son distintos al vacio y al cero, y que
el parámetro conf se encuentra entre [0,1]
;;; INPUT: cat: vector que representa a una categoría, representado como una lista
;;; vs: vector de vectores
;;; conf: Nivel de confianza
;;; OUTPUT: t si los argumentos son correctos, nil en caso contrario

(defun is-ok2 (cat vs conf)
   (if (and (no-vacia cat) (no-cero cat) (>= conf 0) (<= conf 1)) ; si vs y conf son
correctos
         (mapcar #'(lambda(x) (and (no-vacía x) (no-cero x))) vs) ; crea una lista con t o
nil en función de si los vectores son correctos o no
         (list nil)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf (cat vs conf)
;;; Devuelve aquellos vectores similares a una categoria
;;; INPUT: cat: vector que representa a una categoría, representado como una lista
;;; vs: vector de vectores
;;; conf: Nivel de confianza
;;; OUTPUT: Vectores cuya similitud con respecto a la categoría es superior al
;;; nivel de confianza, ordenados

(defun sc-conf (cat vs conf)
   (if (some #'null (is-ok2 cat vs conf))
         nil
         (sort (limpia-lista cat vs conf) #'(lambda(y z) (> (sc-rec cat y) (sc-rec cat z))))))
```

1.3

Batería de ejemplos:

```
;;; (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))
;;; (sc-classifier nil texts #'sc-mapcar) ;; --> nil
;;; (sc-classifier '((0 0)) '((1 2)) #'sc-mapcar) ;; --> nil
;;; (sc-classifier '(nil) '((1 2)) #'sc-mapcar) ;; --> nil
```

Pseudocódigo:

Entrada: cats texts func
Salida: n (Pares (id, similitud))
Procesamiento:
Si cats o texts son vacias o cero
    devuelve nil
Para todo x de texts
    aux = (id, similitud coseno)
    aux = ordenar aux por similitud coseno
    n += (primer par de aux), donde U denota la union

Código:

```
;;; is-ok3 (cats texts)
;;; Función que comprueba si los argumentos son correctos
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;; texts: vector de vectores, representado como una lista de listas
;;; OUTPUT: lista con los valores de verdad de los argumentos

(defun is-ok3 (cats texts)
   (if (or (null cats) (null texts))
         (list nil)
         (mapcar #'(lambda(x y) (and (no-vacia x) (no-cero x) (no-vacia y) (no-cero y)))
cats texts))) ; comprueba que se cumpla todas las condiciones

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; encuentra-categoria (x cat f)
;;; Función que devuelve la categoría a la que pertenece un vector
;;; INPUT: x: vector representado como una lista
;;; cat: vector de vectores, representado como una lista de listas
;;; f: función para evaluar la similitud coseno
;;; OUTPUT: par de identificador categoría e valor de la similitud coseno del vector con
la categoría a la que pertenece

(defun encuentra-categoria (x cat f)
   (first (sort (mapcar #'(lambda (y) ; primer par de idenificador categoría tras haberlo
ordenado
   (cons (first y) (funcall f (rest y) (rest x)))) cat) ; creacion de lista con identificador y
similitud coseno
   #'(lambda(x y) (> x y)) :key #'rest))) ; se ordena segun el segundo argumento
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-classifier (cats texts func)
```

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

(defun sc-classifier (cats texts func)
   (if (some #'null (is-ok3 cats texts))
         nil
         (mapcar #'(lambda (x) (encuentra-categoria x cats func)) texts)))
```

1.4

```
;;; (setf cats '((1 43 23 12) (2 33 54 24)))
;;; (setf texts '((1 3 22 134) (2 43 26 58)))
;;; (setf cats1 '((1 43 23 12 32 55 3 5 76 12 96 32) (2 33 54 24 54 31 54 76 37 66 14
27)))
;;; (setf texts1 '((1 3 22 134 32 41 39 4 6 10 11 40) (2 43 26 58 13 24 35 46 23 14 5 6)))
(time (sc-classifier cats texts #'sc-rec))
;; --> 0.000414 sec (99.52 %)
(time (sc-classifier cats texts #'sc-mapcar))
;; --> 0.000296 sec (99.66)
(time (sc-classifier cats1 texts1 #'sc-rec))
;; --> 1 msec
(time (sc-classifier cats1 texts1 #'sc-mapcar))
;; → 0 msec
```

Para valores pequeños es más eficiente emplear el método iterativo, sin embargo, para valores mayores sc-rec es mucho más eficiente como puede comprobarse

2. Raíces de una función.

2.1

Batería de 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)  ;;---> 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 a b tol
Salida: n bisectriz de a b en f con tolerancia tol
Procesamiento:
si f(a) o f(b) es 0 devuelve a o b respectivamente
calculamos el valor medio de a y b (m)
si b-a <tol
devolvemos m
si m*f(a) es es < 0
bisec (a ,m)
si m*f(b) es es < 0
bisec(m,b)

Código:

```
;;; prod-funcion (f a b)
;;; Calcula el producto de f(a) y f(b)
;;; INPUT: f: funcion a evaluar
;;;        a: numero en el que evaluar f
;;;        b: numero en el que evaluar f
;;; OUTPUT: producto de f(a) y f(b)

(defun prod-funcion (f a b)
   (* (funcall f a) (funcall f b)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; valor-medio (a b)
;;; Calcula el punto medio de (a b)
;;; INPUT: a: extremo inferior del intervalo
;;;        b: extremo superior del intervalo
;;; OUTPUT: punto medio de (a b)   (a+b)/2

(defun valor-medio (a b)
   (/ (+ b a) 2))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; bisect (f a b tol)
;; Finds a root of f between the points a and b using bisection.
;; 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, or NIL if no root is found

(defun bisect (f a b tol)
   (if (or (>= (prod-funcion f a b) 0) (> a b))
         nil
         (if (< (- b a) tol)
                  (valor-medio a b)
                  (if (<= (prod-funcion f b (valor-medio a b)) 0)
                           (bisect f (valor-medio a b) b tol)
                           (bisect f a (valor-medio a b) tol)))))
```

2.2

Batería de ejemplos:

```
;;; (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)
```

Pseudocódigo:

if not null biscec(primer y segundo) de lista y el tercero es null
biscec(primer y segundo)
sino, concatena biscec(primer y segundo) con la recursión a partir del primero

Código:

```
;; allroot (f lst 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.
;; OUTPUT: A list o real values containing the roots of the function in
;; the given sub-intervals

(defun allroot (f lst tol)
   (if (not (null (bisect f (first lst) (second lst) tol)))
         (if (null (third lst))
                  (list(bisect f (first lst) (second lst) tol))
                  (cons (bisect f (first lst) (second lst) tol) (allroot f (rest lst) tol)))
         (allroot f (rest lst) tol)))
```

2.3

Batería de ejemplos:

```
;;; (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)
```

Pseudocódigo:

Entrada: f a b N tol
Salida: n (lista con todas las raices desde a hasta b)
Procesamiento:
dividir en intervalos iguales
allroot(intervalo) y agrupamos para todos.

Código:

```
;;; potencia (n)
;;; Calcula el resultado de elevar 2 a la potencia n
;;; INPUT: n: potencia a la que se eleva 2
;;; OUTPUT: 2 elevado a n

(defun potencia (n)
   (if (= n 0)
         1
         (* 2 (potencia (- n 1)))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; tam (a b i)
;;; Calcula el tamaño de los partes en que dividimos un intervalo
;;; INPUT: a: extremo inferior del intervalo
;;;        b: extremo suoerior del intervalo
;;;        n: numero de subintervalos en los que hay que dividir (a b)
;;; OUTPUT: tamaño de los subintervalos

(defun tam (a b n)
   (/(- b a) (potencia n)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; intervalo (a x tam)
;;; Calcula el siguiente extremo de los subintervalos
;;; INPUT: a: extremo inferior del intervalo original
;;;        x: numero de subintervalo a calcular
;;;        tam: tamaño que tiene que tener el subintervalo
;;; OUTPUT: extremo  del subintervalo
```

```
(defun intervalo (a x tam)
   (+ a (* x tam)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; lista (a b pot n)
;;; Agrupa en una lista los numeros que serán extremo de los intervalos
;;; en los que calcularemos la raiz de una funcion
;;; INPUT: a: extremo inferior del intervalo original
;;;        b: extremo superior del intervalo original
;;;        pot: numero de subintervalos en los que dividir (a b)
;;;        n: numero de intervalo que queremos calcular
;;; OUTPUT: extremo  del subintervalo

(defun lista (a b pot n)
   (if (equal n pot)
         (list(intervalo a n (tam a b pot)))
         (cons (intervalo a n (tam a b pot)) (lista a b pot (+ n 1)))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  allind (f a b N 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.
;;  OUTPUT: List with all the found roots.

(defun allind (f a b N tol)
   (if (some #'null (lista a b (potencia N) 0))
         nil
         (allroot f (lista a b (potencia N) 0) tol)))
```

3. Combinación de listas

3.1

Batería de ejemplos:

```
;;; (combine-elt-lst 'a nil) ;; --> NIL
```

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

Pseudocódigo:

Entrada: elt lst
Salida: n (lista de combinacion elt con elementos de lst)
Procesamiento:
Si lst es nil
    devuelve nil
Para todo x de lst
    n += (elt x)

Código:

```
;;; combine-elt-lst (elt lst)
;;; Combina un elemento dado con todos los elementos de una lista
;;; INPUT: elt: elemento que se combina con cada elemento de la lista
;;;        lst: lista de elementos a combinar
;;; OUTPUT:lista con el elemento combinado con cada elemento de la lista

(defun combine-elt-lst (elt lst)
   (if(null lst) ; condicion de error
        nil
        (mapcar #'(lambda(x) (list elt x)) lst))) ; combina elt con cada elemento de lst
```

3.2

Batería de ejemplos:

;;; (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) (A2) (B 1) (B 2) (C 1) (C 2))

Pseudocódigo:

Entrada: lst1 lst2
Salida: n (producto cartesiano de dos listas)
Procesamiento:
Si lst1=null o lst2=null
    devuelve nil
Para todo x de lst1
    n += combine-elt-lst(x lst2)

Código:

```
;;; combine-lst-lst (lst1 lst2)
;;; Calcula el producto cartesiano de dos listas
;;; INPUT: lst1: primera lista
;;;        lst2: segunda lista
;;; OUTPUT: producto cartesiano de las listas

(defun combine-lst-lst (lst1 lst2)
        (if(or (null lst1) (null lst2)) ; condicion de error
                nil
                (mapcan #'(lambda (x) (combine-elt-lst x lst2))lst1))) ; combina las dos
listas como producto cartesiano
```

3.3

Batería de ejemplos:

```
;;; (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))
```

Pseudocódigo:

```
Entrada: lstolsts
Salida: n (conjunto de posibles combinaciones de los elementos pertenecientes  N)
Procesamiento:
Si existe x perteneciente a lstolsts / x = null
        devuelve nil
Si no hay mas lst pertenecientes a lstolsts
        para todo x de lst
                n += (list x)
Si no
        para todo x de combine-lst-lst (first lstolsts) (combine-list-of-lsts (rest lstolsts))
                n += ((first x) (second x))
```

Código:

```
;;; combine-list-of-lsts (lstolsts)
;;; Calcula todas las posibles disposiciones de elementos pertenecientes a N listas de
;;; forma que en cada disposición aparezca únicamente un elemento de cada lista
;;; INPUT: lstolsts: lista de listas
```

```
;;; OUTPUT: lista con las posibles disposiciones.

(defun combine-list-of-lsts (lstolsts)
        (if (some #'null lstolsts)
                nil
        (if (null (rest lstolsts))
                (mapcar #'(lambda (x) (append (list x))) (first lstolsts)); caso base
                (mapcar #'(lambda (x) (cons (first x) (second x))) (combine-lst-lst (first
        lstolsts) (combine-list-of-lsts (rest lstolsts)))))))
```

4. Inferencia en lógica proposicional

Código necesario para cada una de estas funciones

```
;; Definicion de simbolos que representan valores de verdad,
;; conectores y predicados para evaluar si una expresion LISP
;; es un valor de verdad o un conector

(defconstant +bicond+ '<=>) ;; BICONDICIONAL
(defconstant +cond+   '=>) ;; CONDICIONAL
(defconstant +and+  '^) ;; AND
(defconstant +or+    'v) ;; OR
(defconstant +not+   '~) ;; NOT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;Funcion que determina si un elemento tiene valor de verdad
;;
;;RECIBE x elemento
;;EVALUA A : t si es verdad o la lista vacia, nil en caso contrario

(defun truth-value-p (x)
  (or (eql x T) (eql x NIL)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;determina si un elemnto es un conector unario (not)
;;
;;RECIBE : elemento x
;;EVALUA A : T si es el simbolo definido como not, NIL en caso contrario

(defun unary-connector-p (x)
  (eql x +not+))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;determina si un elemnto es un conector binario(cond, bicond)
;;
```

```
;;RECIBE : elemento x
;;EVALUA A : T si es el simbolo definido como cond o bicond, NIL en caso contrario

(defun binary-connector-p (x)
  (or (eql x +bicond+)
      (eql x +cond+)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;determina si un elemnto es un conector n-ario(and, or)
;;
;;RECIBE : elemento x
;;EVALUA A : T si es el simbolo definido como and o or, NIL en caso contrario

(defun n-ary-connector-p (x)
  (or (eql x +and+)
      (eql x +or+)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;determina si un elemnto es un conector de cualquier tipo
;;
;;RECIBE : elemento x
;;EVALUA A : T si es un conector, NIL en caso contrario

(defun connector-p (x)
  (or (unary-connector-p  x)
      (binary-connector-p x)
      (n-ary-connector-p  x)))
```

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

4.1.1

Batería de ejemplos:

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

Pseudocódigo:

Entrada: x
Salida: T si la expresion es literal positivo, nil en caso contrario
Procesamiento:

```
si x = atom && x !=true-value && x != conector
        true
nil
```

Código:

```
;; 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)  ;;un literal es positivo si es un atomo, no tiene valor de verdad y no es conector
   (and (atom x)
        (not (truth-value-p x))
        (not (connector-p x))))
```

4.1.2

Batería de ejemplos:

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

Pseudocódigo:

Entrada: x
Salida: T si es literal negativo, nil en caso contrario
Procesamiento:

si x = lista && (first x) = conector && (second x) = positive-literal-p
true
nil

Código:

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

(defun negative-literal-p (x)
   (and (listp x)                              ;;sabemos que es un literal negativo si es una
lista cuyo primer elemento es un conector unario y el resto un literal positivo
        (unary-connector-p (first x))
        (positive-literal-p (first (rest x)))))
```

4.1.3

Batería de ejemplos:

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

Pseudocódigo:

si x = positive-literal-p || x = negative-literal-p
t
nil

Código:

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

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

4.1.4

Batería de ejemplos:

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

Código:

```
;; EJERCICIO 4.1.4
;; Predicado para determinar si una expresion esta en formato prefijo
;;
;; RECIBE   : expresion x
;; EVALUA A : T si x esta en formato prefijo,
;;        NIL en caso contrario.

(defun wff-infix-p (x)
  (unless (null x)              ;; NIL no es FBF en formato prefijo (por convencion)
        (or (literal-p x)       ;; Un literal es FBF en formato prefijo
        (and (listp x)          ;; En caso de que no sea un literal debe ser una lista
```

```
        (let ((op1 (car x))
                (exp1 (cadr x))
                (list_exp2 (cddr x)))
                (cond
                ((unary-connector-p op1)    ;; Si el primer elemento es un connector unario
                (and (null list_exp2)
                (wff-infix-p exp1)))    ;; evaluamos el segundo elemento
                ((n-ary-connector-p op1)
                (null (rest x)))
                ((binary-connector-p exp1)    ;; Si el primer elemento es un conector
binario
                (and (wff-infix-p op1)          ;; si el primer elemento esta en formato infijo
                (null (cdr list_exp2))
                (wff-infix-p (car list_exp2))));;evaluamos el tercer elemento
                ((n-ary-connector-p exp1) ;; Si el primer elemento es un conector enario
                (and (wff-infix-p op1)
                (nop-verify exp1 (cdr x))))
                (t NIL)))))))

(defun nop-verify (op exp)
  (or (null exp)
        (and (equal op (car exp))
        (wff-infix-p (cadr exp))
        (nop-verify op (cddr exp)))))
```

Batería de ejemplos:

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

Código:

```
;; Convierte FBF en formato prefijo a FBF en formato infijo
```

```
;;
;; RECIBE   : FBF en formato prefijo
;; EVALUA A : FBF en formato infijo

(defun prefix-to-infix (wff)
  (when (wff-prefix-p wff)     ;;solo si wff esta en formati prefijo
        (if (literal-p wff)                ;; si es un literal, lo devuelve
        wff
        (let ((connector        (first wff))
        (elements-wff (rest wff)))
        (cond
        ((unary-connector-p connector)     ;; si es un conector unario, repite el proceso a
partir del segundo elemento
        (list connector (prefix-to-infix (second wff))))
        ((binary-connector-p connector)     ;; si es binario devuelve una lista con la
expresion del segundo, el conector y la expresion del tercero en infijo
        (list (prefix-to-infix (second wff))
                connector
                (prefix-to-infix (third wff))))
        ((n-ary-connector-p connector) ;; si es enario
        (cond
        ((null elements-wff)           ;;; conjuncion o disyuncion vacias.
        wff)                    ;;; por convencion, se acepta como fbf en formato infijo
        ((null (cdr elements-wff))  ;;; conjuncion o disyuncion con un unico elemento
        (prefix-to-infix (car elements-wff)))
        (t (cons (prefix-to-infix (first elements-wff))
                (mapcan #'(lambda(x) (list connector (prefix-to-infix x)))
                (rest elements-wff))))))
        (t NIL))))))) ;; no deberia llegar a este paso nunca
```

4.1.5

Batería de 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))
```

<u>Código</u>:

```
;; 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

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

4.1.6

Batería de ejemplos:

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

(clause-p '(^ a b q (~ r) s)) ; NIL
(clause-p '(v (^ a b) q (~ r) s)) ; NIL
(clause-p '(~ (v p q)))   ; NIL

Pseudocódigo:

Entrada: wff
Salida : T o NIL
Procesamiento:
(if (first) == or and allrest == literal) T
NIL

Código:

```
;;determina si todos los elementos de una lista son literales
;;
;;RECIBE : wff fbf pormato prefijo sin el primer elemento
;;EVALUA A : T si es una lista vacia o son todos literales, nil en caso contrario

(defun all-literal (wff)
   (if (null wff)
          t
          (and (literal-p (first wff)) (all-literal (rest wff)))))   ;;de manera recursiva determina
si son todos literales con un and de todos los elementos

;; Predicado para determinar si una FBF es una clausula
;;
;; RECIBE   : FBF en formato prefijo
;; EVALUA A : T si FBF es una clausula, NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun clause-p (wff)
   (unless (null wff)
          (and t (eql (first wff) +or+) (all-literal (rest wff)))))   ;;para que sea clausula el
pimer elemento debe ser un or y el resto literales
```

4.1.7

Batería de ejemplos:

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

```
(cnf-p '(v p q (~ r) s))                          ; NIL
(cnf-p '(^ (v a b) q (v (~ r) s) a b))            ; NIL
(cnf-p '(^ p))                          ; NIL
(cnf-p '(v ))                           ; NIL
(cnf-p NIL)                                       ; NIL
(cnf-p '((~ p)))                        ; NIL
(cnf-p '(p))                            ; NIL
(cnf-p '(^ (p)))                        ; NIL
(cnf-p '((p)))                          ; NIL
(cnf-p '(^ a b q (r) s))                          ; NIL
(cnf-p '(^ (v a  (v b c)) (v q r) (v (~ r) s) a b)) ; NIL
(cnf-p '(^ (v a (^ b c)) (^ q r) (v (~ r) s) a b))  ; NIL
(cnf-p '(~ (v p q)))                              ; NIL
(cnf-p '(v p q (r) s))                            ; NIL
```

Pseudocódigo:

Entrada: wff

Salida : T o NIL

Procesamiento:

(if (first) == and and allrest ==clausula) T

Código:

```
;; determina si todos los elementos son clausulas
;;
;;RECIBE   : una ffb en formato prefijo sin el primer elemento
;;EVALUA A : T si es una lista vacia o todos sus elementos son clausulas, NIL en caso
contrario
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun all-clause (wff)
   (if (null wff)
         t
         (if (listp (first wff))
                  (and (clause-p (first wff)) (all-clause (rest wff))) ;;determina de manera
recursiva si todos los elementos son clausulas usando un and de todos ellos
                  nil)))

;; Predicado para determinar si una FBF esta en FNC
;;
;; RECIBE   : FFB en formato prefijo
;; EVALUA A : T si FBF esta en FNC con conectores,
;;        NIL en caso contrario.
```

```
(defun cnf-p (wff)
   (unless (null wff)
        (and t (eql (first wff) +and+) (all-clause (rest wff))))) ;; el primer elemento debe
ser un and y el resto clausulas
```

4.2. Algoritmo de transformación de una FBF a FNC

4.2.1

Batería de ejemplos:

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

Pseudocódigo:

```
Entrada: wff
Salida: wff sin conector <=>
Procesamiento:
        si wff = null || wff = literal-p
                wff
        conector = first(wff)
        si conector =  <=>
                wff1 = eliminate-biconditional (second (wff))
                wff2 = eliminate-biconditional (third (wff))
                wff = (^ (=> wff1 wff2) (=> wff2 wff1))
        wff = (conector eliminate-biconditional(x) para todo x de (rest(wff)))
```

Código:

```
;; 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 wff es nil o es literal, devolvemos wff tal cual
        wff
        (let ((connector (first wff))) ;; declaramos como connector al primer elemento de
wff
        (if (eq connector +bicond+) ;; si connector es bicondicional
```

(let ((wff1 (eliminate-biconditional (second wff))) ;; declaramos wff1 al resultado de la llamada recursiva con el segundo elemento de wff
(wff2 (eliminate-biconditional (third wff)))) ;; declaramos wff2 al resultado de la llamada recursiva con el tercer elemento de wff
(list +and+ ;; aplicamos definicion de bicondicional
(list +cond+ wff1 wff2) ;; (wff1 => wff2) en formato prefijo
(list +cond+ wff2 wff1))) ;; (wff2 => wff1) en formato prefijo
(cons connector
(mapcar #'eliminate-biconditional (rest wff))))))) ;; si no es bicondicional, analizamos el resto de wff

4.2.2

Batería de ejemplos:

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

Pseudocódigo:

Entrada: wff
Salida: wff (FBF sin conector =>)
Procesamiento:
si wff = null || wff = literal-p
wff
conector = first(wff)
si conector = '=>'
wff1 = eliminate-conditional (second (wff))
wff2 = eliminate-conditional (third (wff))
wff = (v (¬ wff1) wff2)
wff = (conector eliminate-conditional(x) para todo x de (rest(wff)))

Código:

```
;; Dada una FBF, que contiene conectores => evalua a
;; una FBF equivalente que no contiene el connector =>
;;
;; RECIBE   : wff en formato prefijo sin el connector <=>
;; EVALUA A : wff equivalente en formato prefijo
;;        sin el connector =>

(defun eliminate-conditional (wff)
```

```
   (if (or (null wff) (literal-p wff)) ;; si wff es nil o literal, devolvemos wff tal cual
                wff
        (let ((connector (first wff))) ;; declaramos conector = primer elemento de wff
         (if (eq connector +cond+) ;; si es conector condicional
                (let ((wff1 (eliminate-conditional (second wff))) ;; wff1 = llamada recursiva
con el segundo elemento de wff
               (wff2 (eliminate-conditional (third  wff)))) ;; wff2 = llamada recursiva con el
tercer argumento
        (list +or+ ;; aplicamos definicion de condicional
                (list +not+ wff1) wff2)) ;; ((¬ wff1) v wff2)
        (cons connector
        (mapcar #'eliminate-conditional (rest wff)))))))  ;; si no es condicional, analizamos
el resto de wff
```

4.2.3

Batería de ejemplos:

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

Pseudocódigo:

```
Entrada: wff
Salida: wff sin conector negacion
Procesamiento:
        si wff = null || wff = literal-p
                wff
        conector = first(wff)
        si conector = '¬'
                si first(second(wff)) = '¬'
                        reduce-scope-of-negation (cadr (cadr wff))
                wff = (conector reduce-scope-of-negation (list +not+ x) para todo x de (rest
(second wff))
```

Código:

```
;; Cambia un conector and por or y viceversa
;;
;; RECIBE   : connector - conector unario
;; EVALUA A : nuevo conector contrario

(defun exchange-and-or (connector)
  (cond
   ((eq connector +and+) +or+) ;; si el conector es and, cambiamos a or
```

```
  ((eq connector +or+) +and+) ;; si el conector es or, cambiamos a and
  (t connector)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; funcion que niega un literal
;;
;; RECIBE   : x - literal
;; EVALUA A : literal negado
;; Funcion que no usada en este ejercicio pero si util mas adelante

(defun negar-literal (x)
   (if (positive-literal-p x) ;; si el literal es postivo lo negamos
         (list +not+ x)
         (second x))) ;; negar un literal negativo, es devolver el positivo
;; Dada una FBF, que no contiene los conectores <=>, =>
;; evalua a una FNF equivalente en la que la negacion
;; aparece unicamente en literales negativos
;;
;; RECIBE   : FBF en formato prefijo sin conector <=>, =>
;; EVALUA A : FBF equivalente en formato prefijo en la que
;;       la negacion  aparece unicamente en literales
;;       negativos.

(defun reduce-scope-of-negation (wff)
  (if (or (null wff) (literal-p wff)) ;; si wff es nil o literal, devolvemos wff tal cual
       wff
       (let ((connector (first wff))) ;; declaramos conector = primer elemento de wff
       (if (eq connector +not+) ; Comprobamos si el operador es el not
       (if (eq (first (first (rest wff))) +not+) ; en caso de doble negacion se deja igual
       (reduce-scope-of-negation (cadr (cadr wff)))
       (cons (exchange-and-or (first (first (rest wff)))) ; Aplicamos la ley de De Morgan
              (mapcar #'(lambda(x) (reduce-scope-of-negation (list +not+ x))) (rest
(second wff)))))
       (cons connector (mapcar #'reduce-scope-of-negation (rest wff))))))) ; Si el
operador no es not, analizamos el resto de wff
```

4.2.4

Batería de ejemplos:

```
(cnf NIL)               ; NIL
(cnf 'a)                ; (^ (V A))
(cnf '(~ a))            ; (^ (V (~ A)))
(cnf '(V (~ P) (~ P))) ; (^ (V (~ P) (~ P)))
(cnf '(V A))            ; (^ (V A))
```

```
(cnf '(^ (v p (~ q)) (v k r (^ m  n))))
;;;   (^ (V P (~ Q)) (V K R M) (V K R N))
```

Pseudocódigo:

Entrada: wff
Salida: wff en formato FNC
Procesamiento:

```
    si wff = literal-p
        wff = (^ (v wff))
    si wff esta en formato FNC
        conector = first(wff)
        si conector = '^'
            (^ (lista sin co))
```

Código:

```
;;combina cada elemento de una lista con otro creando una lista de pares
;;
;;RECIBE   : elt -elemento a combinar, lst lista que se combina
;;EVALUA A : devuelve una lista de pares de elt y cada elemento de lst
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun combine-elt-lst (elt lst)
  (if (null lst)
        (list (list elt))
        (mapcar #'(lambda (x) (cons elt x)) lst)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;cambia los and por or en uma cnf
;;
;;RECIBE  : nf- lista de elementos
;;EVALUA A: una lista igual pero con los elementos and y oir cambiados segun
corresponda

(defun exchange-NF (nf)
  (if (or (null nf) (literal-p nf))
        nf
        (let ((connector (first nf)))
        (cons (exchange-and-or connector)
        (mapcar #'(lambda (x)
                        (cons connector x))
                (exchange-NF-aux (rest nf)))))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;funcion auxiliaren la que se basa exchange-NF para hacer el cambio
;;
;;RECIBE  : nf- lista de elementos
```

```
;;EVALUA A: lista con los elementos combinados.

(defun exchange-NF-aux (nf)
  (if (null nf)
        NIL
        (let ((lst (first nf)))
        (mapcan #'(lambda (x)
                (combine-elt-lst
                x
                (exchange-NF-aux (rest nf))))
        (if (literal-p lst) (list lst) (rest lst))))))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;devuelve una clausula sin los conectores de manera que estos quedan supuestos
;;
;;RECIBE  : una lista de elementos (Clausula)
;;EVALUA A: una lista sin conectores en los que se suponen or's

(defun simplify (connector lst-wffs )
  (if (literal-p lst-wffs)
        lst-wffs
        (mapcan #'(lambda (x)
                (cond
                ((literal-p x) (list x))
                ((equal connector (first x))
                (mapcan
                #'(lambda (y) (simplify connector (list y)))
                (rest x)))
                (t (list x))))
        lst-wffs)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Dada una FBF, que no contiene los conectores <=>, => en la
;; que la negacion aparece unicamente en literales negativos
;; evalua a una FNC equivalente en FNC con conectores ^, v
;;
;; RECIBE   : FBF en formato prefijo sin conector <=>, =>,
;;        en la que la negacion aparece unicamente
;;        en literales negativos
;; EVALUA A : FBF equivalente en formato prefijo FNC
;;        con conectores ^, v

(defun cnf (wff)
  (cond
```

```
        ((literal-p wff)
        (list +and+ (list +or+ wff)))
  ((cnf-p wff) wff)
  ((let ((connector (first wff)))
        (cond
        ((equal +and+ connector)
        (cons +and+ (simplify +and+ (mapcar #'cnf (rest wff)))))
        ((equal +or+ connector)
        (cnf (exchange-NF (cons +or+ (simplify +or+ (rest wff))))))))))
```

4.2.5

Batería de ejemplos:

```
(eliminate-connectors 'nil)
;; NIL
(eliminate-connectors (cnf '(^ (v p  (~ q))  (v k  r  (^ m  n)))))
;; ((P (~ Q)) (K R M) (K R N))
(eliminate-connectors
 (cnf '(^ (v (~ a) b c) (~ e) (^ e f (~ g) h) (v m n) (^ r s q) (v u q) (^ x y))))
;; (((~ A) B C) ((~ E)) (E) (F) ((~ G)) (H) (M N) (R) (S) (Q) (U Q) (X) (Y))
(eliminate-connectors (cnf '(v p  q  (^ r  m)  (^ n  q)  s )))
;; ((P Q R N S) (P Q R Q S) (P Q M N S) (P Q M Q S))
(eliminate-connectors (print (cnf '(^ (v p  (~ q)) (~ a) (v k  r  (^ m  n))))))
;; ((P (~ Q)) ((~ A)) (K R M) (K R N))
(eliminate-connectors '(^)) ; NIL
(eliminate-connectors '(v)) ; NIL
(eliminate-connectors '(^ (v p (~ q)) (v) (v k r))) ; ((P (~ Q)) NIL (K R))
(eliminate-connectors '(^ (v a b))) ; ((A B))
```

Pseudocódigo:

```
Entrada: cnf
Salida: cnf sin conectores
Procesamiento:
        si cnf = nul || rest(cnf) = nul
                nil
        aplica rest a (rest x) para cada x de cnf
```

Código:

```
;; Dada una FBF en  FNC
;; evalua a lista de listas sin conectores
;; que representa una conjuncion de disyunciones de literales
;;
;; RECIBE   : FBF en FNC con conectores ^, v
```

```
;; EVALUA A : FBF en FNC (con conectores ^, v eliminaos)

(defun eliminate-connectors (cnf)
   (if (or (null (rest cnf)) (null cnf)) ;; Si cnf es nil o cnf es un unico elemento, devolvemos
nil
         nil
         (mapcar #'rest (rest cnf)))) ;; elimina los conectores ^ y v de cnf
```

4.2.6

Batería de ejemplos:

```
(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) ^ e))
;; ((P (~ A) B) (P (~ A) (~ C)) (P (~ A) D) ((~ P) (~ Q)) (Q P) (P) (E))
```

Pseudocódigo:

```
Entrada: wff
Salida: wff en formato FNC
Procesamiento:
        (eliminate-connectors
                (cnf
                        (reduce-scope-of-negation
                                (eliminate-conditional
                                        (eliminate-biconditional
                                                (infix-to-prefix wff)))))))
```

Código:

```
;; Dada una FBF en formato infijo
;; evalua a lista de listas sin conectores
;; que representa la FNC equivalente
;;
;; RECIBE   : FBF
;; EVALUA A : FBF en FNC (con conectores ^, v eliminados)

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

4.3 Simplificación de FBFs en FNC

4.3.1

Batería de ejemplos:

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

Pseudocódigo:

```
Entrada:k
Salida: k sin elementos repetidos
Procesamiento:
        si k == literal-p
                k
        si k esta en rest k (esta repetido)
                eliminate-repeated-literals( rest k)
         sino,  guardo k y eliminate-repeated-literals( rest k)
```

Código:

```
;; EJERCICIO 4.3.1
;; eliminacion de literales repetidos una clausula
;;
;; RECIBE   : K - clausula (lista de literales, disyuncion implicita)
;; EVALUA A : clausula equivalente sin literales repetidos

(defun eliminate-repeated-literals (k)
   (cond ((null k) nil)
            ((literal-p k) k)
            ((member (first k) (rest k) :test #'equal) (eliminate-repeated-literals (rest k))) ;; si
el primer elemento esta repetido, eliminar repetidos del resto de la lista
            (t(cons (first k) (eliminate-repeated-literals (rest k))))) ;; si no esta, creamos
una lista con el primero y el resultado de eliminar los repetidos del resto
```

4.3.2

Batería de ejemplos:

```
(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:cnf
Salida: cnf sin clausulas repetidas
Procesamiento:
limpiar cnf
si first k == algun elem de k
eliminate-repeated-clauses(rest k)
sino, guardo k y eliminate-repeated-clauses(rest k)

Código:

```
;;elimina los elementos repetidos de las clausulas de una cnf
;;
;;RECIBE   : k- FBF en FNC (lista de clausulas, conjuncion implicita)
;:EVALUA A : FNC equivalente sin elementos repetidos en sus clausulas


(defun limpiar-c (k)
   (cond ((null k) nil)
         (t(mapcar #'(lambda(x) (eliminate-repeated-literals x)) k))))

;;determina si dos clausulas son iguales
;;
;;RECIBE   : l1 y l2 listas de literales (clausulas)
;;EVALUA A  : T si ambas son iguales, NIL si son diferentes


(defun equals(l1 l2)
   (if (= (length (intersection l1 l2 :test 'equal)) (length l1) (length l2))
          T
          NIL))

;;determina si una a una si las clausulas son iguales que alguna de las clausulas
siguientes de la lista.
;;
;;RECIBE  : una lista de clausulas
;;EVALUA A: FNC equivalente sin clausulas repetidas

(defun erc-aux(list)
   (cond ((null list) nil)
                 ((every #'null (mapcar #'(lambda(x) (equals (first list) x))(rest list)))
                  (cons(first list)(erc-aux (rest list)))) ;; si la primera clausula es distinta de
todas las demas se añade a la lista que devolvera la funcion
```

(t(erc-aux (rest list))))) ;; si la primera clausula si que esta repetida, evaluamos la funcion a partir de ella.

```
;; eliminacion de clausulas repetidas en una FNC
;;
;; RECIBE   : cnf - FBF en FNC (lista de clausulas, conjuncion implicita)
;; EVALUA A : FNC equivalente sin clausulas repetidas

(defun eliminate-repeated-clauses (cnf)
   (erc-aux(limpiar-c cnf))) ;; llamamos a la funcion que elimina las clausulas repetidas sobre una lista de clausulas sin elementos repetidos
```

4.3.3

Batería de 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
```

Pseudocódigo:

```
Entrada:k1 k2
Salida: k1 si T, NIL si k1 no subsume
Procesamiento:
        for x elemento in k1
                if x member k2 --para todo x
                        k1
                sino NIL
```

Código:

```
;;determina si un elemento subsume a una clausula
;;
;;RECIBE  : e-elemento, L clausula
;;EVALUA A : e si el elemento subsume a la clausula, NIL si no la subsume


(defun subsume-el (e L)
        (if (member e L :test #'equal) e NIL))


;;determina si cada elemento de la primera clausula, subsume a la segunda.
;;
;;RECIBE   : K1, K2  clausulas
;;EVALUA A : lista con elementos y nil en funcion de si subsumen o no


(defun subsume-tot (K1 K2)
        (mapcar #'(lambda(x) (subsume-el x K2))K1))

;; Predicado que determina si una clausula subsume otra
;;
;; RECIBE   : K1, K2 clausulas
;; EVALUA a : (list K1) si K1 subsume a K2
;;           NIL en caso contrario
(defun subsume (K1 K2)
        (if (some #'null (subsume-tot K1 K2)) ;; si algun elemento de la primera clausula
no esta en la segunda, no subsume.
                nil
                (list K1))) ;; si todos estan, si subsume
```

4.3.4

Batería de 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))
```

Pseudocódigo:

```
Entrada:cnf
Salida:cnf sin clausulas repetidas
Procesamiento:
        for x y clausula in k1
                if x is subsumed by y --para todo x, y
                        remove x
```

Código:

```
;;determina si la primera clausula es subsumida por la segunda
;;
;;RECIBE      : x y clausulas
;; EVALUA A  : NIL si son iguales o x no es subsumida por y, x si x es subsumida por y

(defun is-subsumed (x y)
   (cond ((equals y x) NIL)
          ((subsume y x) x )))

;;agrupa en un lista si una clausula de L es subsumido por la clausula e
;;
;;RECIBE      : e clausula que subsume
;;EVALUA A  : una lista cuyos elementos son el resultado de aplicar is-subsumed de
cada elemento de L con e

(defun aux1 (e L)
   (mapcar #'(lambda(x) (is-subsumed x e))L))

;;agrupa en una lista las listas generadas en aux1 para cada elemento de list
;;
;;RECIBE      : list -lista de clausulas
;;EVALUA A  : lista cuyos elementos son los elementos de las listas de aux1 para cada
elemento de list
(defun aux2 (list)
   (mapcan #'(lambda(x) (aux1 x list)) list))
```

```
;;elimina los elementos de la lista de aux2 que son nil, de manera que nos quedamos
con clausulas no vacias
;;
;;RECIBE   : cnf lista de clausulas
;;EVALUA A : lista de las clausulas subsumidas de cnf

(defun esc-aux (cnf)
  (remove-if #'null (aux2 cnf)))

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

(defun eliminate-subsumed-clauses (cnf)
   (set-difference cnf (esc-aux cnf)))
```

4.3.5

Batería de ejemplos:

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

Pseudocódigo:

```
Entrada:k
Salida: T si hay tautologia NIL si no hay
Procesamiento:
        for x in k
                if !x member de k
                        T
                NIL
```

Código:

```
;;determina para cada elemento de la lista, esta tambien su contrario
;;
;;RECIBE      : una lista (clausula)
;;EVALUA A  : una lista de T y Nil

(defun both (L)
   (mapcar #'(lambda(x) (member (negar-literal x) L :test #'equal))L))
```

```
;;Predicado que determina si una clausula es tautologia
;;
;; RECIBE   : K (clausula)
;; EVALUA a : T si K es tautologia
;;         NIL en caso contrario

(defun tautology-p (K)
   (if(every #'null (both K))
         NIL  ;; si para ningun elemento, esta tambien su contrario, entonces no es
tatutologia
         T))  ;; si para algun elemento esta tambien su contrario, es tautologia
```

4.3.6

Batería de 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
```

Pseudocódigo:

```
Entrada:cnf
Salida: cnf sin tautologias
Procesamiento:
         for x in k
                  if  tautology(x)
                           remove x
```

Código:

```
;; eliminacion de clausulas en una FBF en FNC que son tautologia
;;
;; RECIBE   : cnf - FBF en FNC
;; EVALUA A : FBF en FNC equivalente a cnf sin tautologias

(defun eliminate-tautologies (cnf)
         (remove-if #'(lambda(x) (tautology-p x)) cnf))
```

4.3.7

Batería de 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
```

Pseudocódigo:

Entrada:cnf
Salida: cnf limpia sin repeticiones ni tautologias ni clausulas subsumidas
Procesamiento:
    eliminate-repeated-clauses
    eliminate-tautologies
    eliminate-subsumed-clauses

Código:

```
;; simplifica FBF en FNC
;;      * elimina literales repetidos en cada una de las clausulas
;;      * elimina clausulas repetidas // esta hace lo de arriba tbn
;;      * elimina tautologias
;;      * elimina clausulass subsumidas
;;
;; RECIBE   : cnf  FBF en FNC
;; EVALUA A : FNC equivalente sin clausulas repetidas,
;;      sin literales repetidos en las clausulas
;;      y sin clausulas subsumidas

(defun simplify-cnf (cnf)
(eliminate-subsumed-clauses (eliminate-tautologies (eliminate-repeated-clauses cnf))))
```

4.4. Construcción de RES

4.4.1

Batería de ejemplos:

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

'((p (~ q) r) (p q) (r (~ s) p q) (a b p) (a (~ p) c) ((~ r) p s)))
;; NIL

Pseudocódigo:

Entrada: lambda cnf
Salida: cnf sin aquellas clausulas que contengan lambda o su negacion
Procesamiento:

```
si cnf != null
    si lambda o ¬lambda pertenece al first(wff)
        (first(cnf) extract-neutral-clauses(lambda rest(wff)))
nil
```

Código:

```
;; Comprueba si una clausula contiene al literal positivo lambda
;;
;; RECIBE : lambda - literal positivo
;;          lst - clausula
;; EVALUA A : T si lo contiene, nil en caso contrario
;;

(defun contiene-lambda (l lst)
   (if (not (equal (member l lst :test #'equal) NIL)) ;; si member no devuelve nil, entonces
lambda pertenece a lst
       t
       nil))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Comprueba si una clausula contiene al literal positivo lambda
;; o a su negacion
;;
;; RECIBE : lambda - literal positivo
;;          lst - clausula
;; EVALUA A : T si lo contiene, nil en caso contrario

(defun contiene-lambda-neutral (l lst)
   (if (or (contiene-lambda l lst) (contiene-lambda (negar-literal l) lst))
       t
       nil))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Construye el conjunto de clausulas lambda-neutras para una FNC
;;
;; RECIBE   : cnf    - FBF en FBF simplificada
;;    lambda - literal positivo
```

```
;; EVALUA A : cnf_lambda^(0) subconjunto de clausulas de cnf
;;        que no contienen el literal lambda ni ~lambda

(defun extract-neutral-clauses (lambda cnf)
   (if (not (null cnf))
        (if (contiene-lambda-neutral lambda (first cnf))
                (extract-neutral-clauses lambda (rest cnf))
                (cons (first cnf) (extract-neutral-clauses lambda (rest cnf))))
        nil))
```

4.4.2

Batería de ejemplos:

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

Pseudocódigo:

```
Entrada: lambda cnf
Salida: cnf sin aquellas clausulas que no contengan lambda
Procesamiento:
        si cnf != null
                si lambda pertenece a first(cnf)
                        (first(cnf) extract-positive-clauses lambda (rest cnf))
                extract-positive-clauses lambda (rest cnf)
        nil
```

Código:

```
;; Construye el conjunto de clausulas lambda-positivas para una FNC
;;
;; RECIBE   : cnf      - FBF en FNC simplificada
;;        lambda - literal positivo
;; EVALUA A : cnf_lambda^(+) subconjunto de clausulas de cnf
```

```
;;        que contienen el literal lambda

(defun extract-positive-clauses (lambda cnf)
  (if (not (null cnf))
          (if (contiene-lambda lambda (first cnf)) ;; si la primera clausula contiene a
lambda
                    (cons (first cnf) (extract-positive-clauses lambda (rest cnf))) ;; crea lista
con dicha clasula y evalua al resto de cnf
                    (extract-positive-clauses lambda (rest cnf))) ;; si la primera clausula no
esta contenida, entonces evalua directamente al resto de cnf
          nil)) ;; cnf es nil
```

4.4.3

Batería de 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
```

Pseudocódigo:

```
Entrada: lambda cnf
Salida: cnf sin aquellas clausulas que no contengan ¬lambda
Procesamiento:
        si cnf != null
                si ¬lambda pertenece a first(cnf)
                        (first(cnf) extract-negative-clauses lambda (rest cnf))
                extract-negative-clauses lambda (rest cnf)
        nil
```

Código:

```
;; 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)
  (if (not (null cnf))
        (if (contiene-lambda (negar-literal lambda) (first cnf)) ;; si la primera clausula
contiene a la negacion de lambda
                (cons (first cnf) (extract-negative-clauses lambda (rest cnf))) ;; crea lista
con la primera clausula, y evalua al resto de cnf
                (extract-negative-clauses lambda (rest cnf))) ;; evalua al resto de cnf
        nil)) ; cnf es nil
```

4.4.4

Batería de 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
```

Pseudocódigo:

Entrada: lambda K1 K2
Salida: lista con la resolucion de K1 y K2
Procesamiento:
- si lambda pertenece a K1 && ¬lambda pertenece a K2
  - K1 = eliminar lambda de K1
  - K2 = eliminar ¬lambda de K2
  - K1 U K2 sin elementos repetidos
- si lambda pertenece a K2 && ¬lambda pertenece a K1
  - K2 = eliminar lambda de K2
  - K1 = eliminar ¬lambda de K1
  - K1 U K2 sin elementos repetidos

nil

Código:

```
;; Elimina elemento de una lista
;;
;; RECIBE   : ele      - literal
;;         lst       - clausula
;; EVALUA A : lista sin ele si ele pertence a la lista

(defun eliminar-elemento (ele lst)
   (remove-if #'(lambda (y) (equal ele y)) lst))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Union de K1 sin lambda y K2 sin la negacion de lambda
;;
;; RECIBE   : lambda          - literal positivo
;;         K1, K2           - clausulas simplificadas
;; EVALUA A : union de RES_lambda(K1) y RES_notlambda(K2)

(defun resolve-on-aux (lambda K1 K2)
   (if (and (null (rest K1)) (null (rest K2))) ;; si K1 o K2 son listas de un solo elemento,
evalua a nil
           (list NIL)
           (eliminate-repeated-literals ;; elimina literales repetidos
                                                   (append (eliminar-elemento lambda K1) ;;
union de K1 sin lambda
                                                                   (eliminar-elemento (negar-literal
lambda) K2))))) ;; K2 sin la negacion de lambda
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; resolvente de dos clausulas
;;
;; RECIBE   : lambda          - literal positivo
;;         K1, K2           - clausulas simplificadas
;; EVALUA A : res_lambda(K1,K2)
;;                         - lista que contiene la
;;                         clausula que resulta de aplicar resolucion
;;                         sobre K1 y K2, con los literales repetidos
;;                         eliminados

(defun resolve-on (lambda K1 K2)
```

```
   (if (and (contiene-lambda lambda K1) (contiene-lambda (negar-literal lambda) K2)) ;;
lambda pertenece a K1 y ¬lambda pertenece a K2
         (resolve-on-aux lambda K1 K2) ;; res_aux(K1,K2)
         (if (and (contiene-lambda (negar-literal lambda) K1) (contiene-lambda lambda
K2)) ;; ¬lambda pertenece a K1 y lambda pertenece a K2
                  (resolve-on-aux lambda K2 K1) ;; res_aux(K1,K2)
                  nil))) ;; otro caso evalua a nil
```

4.4.5

Batería de ejemplos:

```
(build-RES 'p NIL)
;; NIL
(build-RES 'P '((A  (~ P) B) (A P) (A B)));; ((A B))
(build-RES 'P '((B  (~ P) A) (A P) (A B)));; ((B A))
(build-RES 'p '(NIL))
;; (NIL)
(build-RES 'p '((p) ((~ p))))
;; (NIL)
;; ((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 cnf
Salida: RES_lambda(cnf)
Procesamiento:
        res = resolucion lambda de (extract-positive-clauses(lambda cnf)) y
(extract-negative-clauses (lambda cnf)
        neutral = extract-neutral-clauses(lambda cnf)
        union = res U neutral, sin elementos repetidos
```

Código:

```
;; Aplica resolucion de una clausula sobre cada clausula de una lista de clausulas
;;
;; RECIBE   : lambda - literal positivo
;;                  x - clausula
;;        lst       - FBF en FNC simplificada
;;
;; EVALUA A : RES_lambda(x, y) para todo y de lst

(defun RES-aux (lambda x lst)
```

```
   (mapcar #'(lambda (y) (resolve-on lambda x y)) lst)) ;; union de aplicar resolucion de
x con cada clausula de lst
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Aplica resolucion de dos listas de clausulas
;;
;; RECIBE   : lambda - literal positivo
;;                  lst1  - FBF en FNC simplificada
;;       lst2     - FBF en FNC simplificada
;;
;; EVALUA A : RES_lambda(x, y) para todo x de lst1 y para todo y de lst2

(defun RES (lambda lst1 lst2)
   (mapcan #'(lambda(x) (RES-aux lambda x lst2)) lst1)) ;; producto escalar de la
resolucion de dos FBF en FNC simplificadas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Construye el conjunto de clausulas RES para una FNC
;;
;; RECIBE   : lambda - literal positivo
;;       cnf      - FBF en FNC simplificada
;;
;; EVALUA A : RES_lambda(cnf) con las clauses repetidas eliminadas

(defun build-RES (lambda cnf)
   (eliminate-repeated-clauses ;; elimina clausulas repetidos
         (append (extract-neutral-clauses lambda cnf) ;; subconjunto de clausulas de cnf
que no contienen ni lambda ni ¬lambda
                        (RES lambda (extract-positive-clauses lambda cnf) ;; RES_lambda
de cnf_lambda^(+) y cnf_lambda^(-)
                                        (extract-negative-clauses lambda cnf)))))
```

4.5. Algoritmo para determinar si una FNC es SAT

Batería de 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
(RES-SAT-p '((P (~ Q)) (K R))) ;;; T
;;
```

```
;; UNSAT Examples
;;
(RES-SAT-p '((P (~ Q)) NIL (K R))) ;;; NIL
(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
Salida: true si es SAT y nil si es UNSAT
Procesamiento:

Data: λ1, λ2,..., λk literales positivos diferentes que aparecen en las cláusulas de α
α 0 = α;
j = 0;
repeat
j = j + 1;
α j = RES λ j (α j−1);
simplificar(α j);
until se obtiene la cláusula vacía (α es UNSAT);
o no se pueden hacer más resoluciones (α es SAT);

Código:

```
;; Elimina los literales negativs de una clausula
;;
;; RECIBE   : lst - clausula
;; EVALUA A : lst sin literales negativos

(defun eliminar-negative (lst)
   (if (negative-literal-p lst) ;; si la lst es un literal negativo, devolvemos nil
        nil
        (remove-if #'(lambda (y) ;; elimina literal negativo de lst
                               (negative-literal-p y)) lst)))

;; lista de literales positivos de una FBF en FNC simplificada
;;
;; RECIBE   : cnf - FBF en FNC simplificada

(defun get-lambdas (cnf)
   (eliminate-repeated-literals ;; elimina literales repetidos
```

```
          (mapcan #'(lambda (x)
                    (eliminar-negative x)) cnf))) ;; aplica eliminar-negative a cada clausula de
cnf

;; comprueba si cnf es unsat aplicando la defincion
;; cnf contiene nil
;;
;; RECIBE   : cnf - FBF en FNC simplificada
;; EVALUA A : T si es unsat, nil en caso contrario

(defun unsat (cnf)
  (if (or (contiene-lambda nil cnf) (contiene-lambda (list nil) cnf)) ;; añado caso (nil) por la
implementacion de build-RES
          t
    nil))

;; aplica resolucion recursivamente sobre la lista de literales y cnf
;; cnf contiene nil o (nil)
;;
;; RECIBE   :    lambdas - lista de literales positivos
;;                          cnf - FBF en FNC simplificada
;; EVALUA A : T si es sat, nil en caso contrario

(defun RES-SAT-aux (lambdas cnf)
  (cond
          ((null cnf) T) ; sat
          ((unsat cnf) NIL) ;; unsat
          ((null lambdas) T)  ;; No se puede seguir resolviendo, sat
          (t (RES-SAT-aux (rest lambdas) (build-RES (first lambdas) cnf))))) ;; llamada
recursiva del resto de lambdas y de la resolucion del primer literal de lambdas y cnf

;; Comprueba si una FNC es SAT calculando RES para todos los
;; atomos en la FNC
;;
;; RECIBE   : cnf - FBF en FNC simplificada
;; EVALUA A :    T  si cnf es SAT
;;                  NIL  si cnf es UNSAT

(defun  RES-SAT-p (cnf)
  (RES-SAT-aux (get-lambdas cnf) cnf)) ;; aplica RES-SAT-aux sobre el conjunto de
literales positivos de cnf y cnf
```

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

Batería de 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
```

Pseudocódigo:

```
Entrada: wff w
Salida: T si es SAT, nil si es UNSAT
Procesamiento:
        wff = wff-infix-to-cnf(wff) simplificadas
        x = wff-infix-to-cnf(¬w) simplificadas
        res = resolucion(wff U x)
        si res = null
                true
        nil
```

Código:

```
;; 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)
   (if (null (RES-SAT-p (union ;; hacemos resolucion de la union
                        (simplify-cnf (wff-infix-to-cnf wff)) ;; simplficamos la transformacion
de wff a cnf
                        (simplify-cnf (wff-infix-to-cnf (list +not+ w))) ;; simplficamos la
transformacion de la ¬w a cnf
                        :test 'equals)))
         t
         nil))
```