Memoria Práctica I

Andrés Salas Peña (andres.salas@estudiante.uam.es)
Ricardo Riol González (ricardo.riol@estudiante.uam.es)

Grupo 2301 - Pareja 8

Inteligencia Artificial
Universidad Autónoma

7 de marzo de 2018

Introducción

Esta primera práctica ha sido un primer acercamiento al lenguaje de programación LISP, un lenguaje muy distinto a los que habíamos visto hasta ahora. La programación funcional, nos abre muchas posibilidades a la hora de implementar algoritmos, ya que en pocas líneas de código se obtienen resultados realmente útiles.

1. Similitud Coseno

1.1 Sc-Rec y Sc-Mapcar

Pseudocódigo

Entrada:
x: vector representado como lista
y: vector representado como lista
Salida:
Similitud coseno entre x e y
Procesamiento:
Si alguna lista es nil devuelve nil, en caso contrario calcula $\frac{x*y}{\|x\|_2\|y\|_2}$

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 1.1.1
;;; sc-rec (x y)
;;; Calcula la similitud coseno de un vector de forma recursiva
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: similitud coseno entre x e y
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun is-ok (x y)
  (cond ((or (equal nil x) (equal nil y)) nil)
        ((or (every #' zerop x) (every #' zerop y)) nil)
        (t t)))

(is-ok '(1 2 3) '(3 4 5)) ;; t
(is-ok '(1 2 3) '(3 4 -5)) ;; nil
(is-ok '(0 0 0) '(1 0 0));; nil
(is-ok '(1 0 0) '());; nil

;;Calcula el producto escalar de dos vectores representados como listas
```

```
(defun our-pesc-rec (x y)
  (if (or (equal nil x) (equal nil y)) ;; Si x o y es null devolvemos 0
      0
    ;; Calculamos sum ( x(i) * y(i))
    (+ (* (first x) (first y)) (our-pesc-rec (rest x) (rest y)))))
    con 1 < i < long x

(our-pesc-rec '(1 2 3) '( 2 -5 6))

(defun sc-rec (lista1 lista2)
  ;;Comprobamos que a lista cumple las condiciones del enunciado
  (if (equal NIL (is-ok lista1 lista2))
      NIL
    (/ (our-pesc-rec lista1 lista2) (*(sqrt (our-pesc-rec lista1 lista1))
    (sqrt (our-pesc-rec lista2 lista2))))))
;; El producto escalar de un vector consigo mismo es su norma al cuadrado

(sc-rec '(1 0) '(0 1)) ;; 0.0
(sc-rec '() '(0 1)) ;; nil
(sc-rec '(1 2 3) '(1 0 0))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 1.1.2
;;; 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 our-pesc-map (lista1 lista2)
  (reduce '+ (mapcar #'* lista1 lista2))) ;; Sum (x(i) * y(i)) com 1< i <len x

;; Utilizamos las funciones recursivas del primer apartado.
(defun sc-mapcar (lista1 lista2)
  (if (equal NIL (is-ok lista1 lista2))
      nil
    (/ (our-pesc-map lista1 lista2) (*(sqrt (our-pesc-map lista1 lista1))
    (sqrt (our-pesc-map lista2 lista2))))))

(sc-mapcar '(1 2 3) '(1 0 0))
```

Comentarios

- En la función sc-rec se realiza una implementación basada en la definición recursiva, para ello hacemos uso de dos funciones auxiliares, is-ok y our-pesc-rec:
  - La primera comprueba si se puede calcular la similitud coseno desechando si alguno de los vectores es nil o está compuesto por ceros.
  - La segunda calcula el producto escalar de los dos vectores pasados.
  - Tras esto, sc-rec solo llama a la primera para ver si se cumplen las condiciones del enunciado y a la segunda varias veces acorde con la fórmula $\frac{x*y}{\|x\|_2\|y\|_2}$
- En la función sc-rec se realiza una implementación basada en el uso del mapcar. La función principal es igual que la anterior y llama también al mismo is-ok pero llama a our-pesc-map que calcula el producto escalar basándose en el mapcar.

1.2 Sc-Conf

Pseudocódigo

Entrada:
x: vector representado como lista
vs: vector de vectores representado como lista de listas
conf: nivel de confianza
Salida:
Vectores ordenados con similitud mayor que confianza
Procesamiento:
Si la similitud de una lista es menor que la confianza pedida se rechaza
Si no, se añade a la lista y se ordenan de mayor a menor

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 1.2
;;;     sc-conf (x vs conf)
;;; Devuelve aquellos vectores similares a una categoria
;;;
;;; INPUT: x: vector, representado como una lista
;;; vs: vector de vectores, representado como una lista de listas
;;; conf: Nivel de confianza
;;;
;;; OUTPUT: Vectores cuya similitud es superior al nivel de confianza, ordenados
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun our-conf (x vs conf)
  ;; Eliminamos de vs las listas cuyo cos con x sea menor que una constante dada
```

```
  (remove-if #'(lambda (y) (< (sc-mapcar x y) conf)) vs))

(our-conf '(1 2 3) '((1 2 3) (2 3 4) (1 0 0)) 0.5)
(sc-mapcar '(1 2 3) '(1 0 0))

(defun sc-conf (x vs conf)
   ;; Ordenamos de mayor a menor el vector en funcion de su cos con la lista x.
  (sort (our-conf x vs conf) #'(lambda (z y) (> (sc-mapcar x z) (sc-mapcar x y)))))

(sc-conf '(1 2 3) '((1 2 3) (3 4 5) (1 0 0) (1 1 1)) 0.9)
```

Comentarios

- En la función sc-conf realizamos la ordenación de mayor a menor de los vectores en función de su similarity-cos, para descartar aquellos que tienen un similarity-cos menor que el nivel de confianza usamos la función auxiliar our-conf:
    - La función our-conf simplemente elimina a través de una función lambda aquellos vectores cuyo valor sea menor que el nivel de confianza.

1.3 Sc-Classifier

Pseudocódigo

Entrada:
cats: vector de vectores representado como lista de listas
vs: vector de vectores representado como lista de listas
func: función que evaluará la similitud coseno
Salida:
Pares identificadores de categoría con el resultado de similitud coseno
Procesamiento:
Calcula cada similitud coseno de cada vs con todas las cats y guarda los pares identificador similitud coseno en una lista y los ordena según su similitud coseno para después quedarse con el mayor y finalmente se añaden a la lista.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 1.3
;;; sc-classifier (cats texts func)
;;; Clasifica a los textos en categorias.
;;;
;;; INPUT: cats: vector de vectores, representado como una lista de listas
;;; vs: vector de vectores, representado como una lista de listas
;;; func: referencia a funcion para evaluar la similitud coseno
;;;
;;; OUTPUT: Pares identificador de categoria con resultado de similitud coseno
```

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

;; Forma una lista de pares de la forma (vector , cos <lista y>) con y
;; perteneciente a cats
(defun our-similarity-cos (cats lista func)
  (mapcar #'(lambda (y) (append(list(first y) (funcall func (rest lista) (rest y)))

(our-similarity-cos '((1 2 3) ( 2 3 4) (6 6 8)) '(3 3 3) #'sc-rec)

;; Ordena los pares en funcion de la segunda coordenada.
(defun our-max-similarity (cats lista func)
  (first (sort(our-similarity-cos cats lista func)
  #'(lambda (z y) (> (second z) (second y))))))

(our-max-similarity '((1 2 3) ( 2 3 5) (6 6 8)) '( 3 6 8) #'sc-rec)

;;Aplica las funciones anteriores a un conjunto de vectores.
(defun sc-classifier (cats texts func)
  (mapcar #'(lambda (z) (our-max-similarity cats z func)) texts))


(sc-classifier '((1 2 3) (2 3 5) (3 6 8)) '((1 3 5) (2 6 8)) #'sc-rec)
(sc-classifier '((1 2 3) (2 3 5) (3 6 8)) '((1 3 5) (2 3 6) (3 2 3)) #'sc-rec)
(sc-classifier '((1 2 3) (2 3 5) (3 6 8)) '((1 3 5) (2 6 8)) #'sc-mapcar)
(sc-classifier '((1 2 3) (2 3 5) (3 6 8)) '((1 3 5) (2 3 6) (3 2 3)) #'sc-mapcar)
(sc-classifier '((1 43 23 12) (2 33 54 24)) '((1 3 22 134) (2 43 26 58)) #'sc-rec)
```

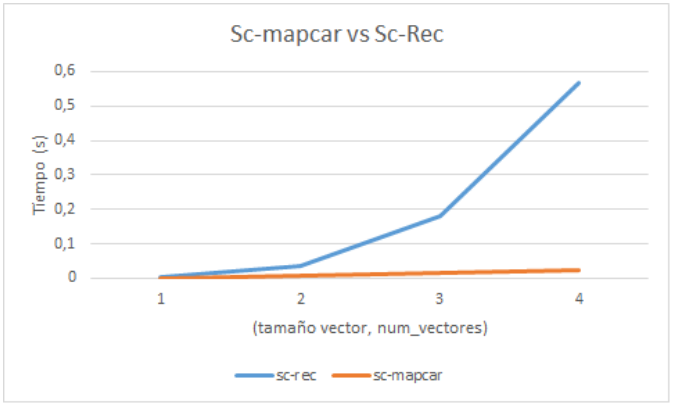
Comentarios

- Para la implementación de esta función nos hemos apoyado en un par de llamadas a otras funciones de tal forma que esta función principal solo se encarga de llamar a las otras tantas veces como vectores en texts tengamos:
    - La función our-max-similarity ordena una lista calculada en our-similarity-cos de mayor a menor según los segundos elementos de las sublistas y se queda con la primera de las sublistas.
        - La función our-similarity-cos realiza mediante un mapcar una lista de pares de la forma (vector, similitud-coseno de cada elemento de la lista)

1.4 Tiempos

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Estudio de tiempos
(time (sc-classifier '((1 2 3 4 5 6 7 3 4 4 4 4 4 4 5)
(2 3 5 5 6 7 8 2 2 2 2 2 3 4 5) (3 6 8 6 7 7 7 1 2 1 2 3 5 6 7))
```

```
'((1 3 5 2 2 2 2 6 8 4 2 1 9 9 9) (2 6 8 4 3 4 6 1 3 4 6 7 2 2 2)) #'sc-rec))
;;real time  0.002000 sec
(time (sc-classifier '((1 2 3 4 5 6 7 3 4 4 4 4 4 4 5)
(2 3 5 5 6 7 8 2 2 2 2 2 3 4 5) (3 6 8 6 7 7 7 1 2 1 2 3 5 6 7))
'((1 3 5 2 2 2 2 6 8 4 2 1 9 9 9) (2 6 8 4 3 4 6 1 3 4 6 7 2 2 2)) #'sc-mapcar))
;;real time  0.000000 sec
(time (sc-classifier (make-list 8 :initial-element
(make-list 20 :initial-element 1)) (make-list 8 :initial-element
(make-list 20 :initial-element 2)) #'sc-rec))
;;real time  0.035000 sec
(time (sc-classifier (make-list 8 :initial-element (make-list 20
:initial-element 1)) (make-list 8 :initial-element (make-list 20
:initial-element 2)) #'sc-mapcar))
;;real time  0.006000 sec
(time (sc-classifier (make-list 12 :initial-element (make-list 50
:initial-element 1)) (make-list 12 :initial-element (make-list 50
:initial-element 2)) #'sc-rec))
;;real time  0.182000 sec
(time (sc-classifier (make-list 12 :initial-element (make-list 50
:initial-element 1)) (make-list 12 :initial-element (make-list 50
:initial-element 2)) #'sc-mapcar))
; real time  0.014000 sec
(time (sc-classifier (make-list 15 :initial-element (make-list 100
:initial-element 1)) (make-list 15 :initial-element (make-list 100
:initial-element 2)) #'sc-rec))
;;real time  0.568000 sec
(time (sc-classifier (make-list 15 :initial-element (make-list 100
:initial-element 1)) (make-list 15 :initial-element (make-list 100
:initial-element 2)) #'sc-mapcar))
;; real time  0.025000 sec
```



En la gráfica anterior definimos los siguientes números que se corresponden con un par (tamaño del vector, número de vectores)
1 := (15,4); 2 := (20,8); 3 := (50,10); 4 := (100,15);
Como se puede observar en la gráfica la utilización de mapcar es más eficiente que la acudir a la recursión. Esto se debe a que la recursión es cotosa si nos es de cola.

2. Raíces de una Función

2.1 Bisect

Pseudocódigo

Entrada:
f: función cuyas raíces queremos encontrar
a: extremo inferior del intervalo en el que buscamos
b: extremo superior del intervalo en el que buscamos
tol: tolerancia que define el criterio de parada
Salida:
Devuelve $\frac{a+b}{2}$ como solución.
Procesamiento:
Subdivide en dos el intervalo y se queda con el que al aplicar $f(inf) * f(sup) < 0$ y lo hace recursivamente.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 2.1
;;; bisect (f a b tol)
;;; Encuentra una raiz de f entre los puntos a y b usando biseccion
;;;
;;; Si f(a)f(b)>0 no hay garantia de que vaya a haber una raiz en el
;;; intervalo, y la funcion devolvera NIL.
;;;
;;; INPUT: f: funcion de un solo parametro real con valores reales cuya
;;; raiz queremos encontrar
;;; a: extremo inferior del intervalo en el que queremos buscar la raiz
;;; b: b>a extremo superior del intervalo en el que queremos buscar la raiz
;;; tol: tolerancia para el criterio de parada: si b-a < tol de la funcion
;;;
;;; OUTPUT: devuelve (a+b)/2 como solucion
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun our-distance (a b) ;;Calcula la distancia entre dos reales
  (abs (- b a)))
```

```
(defun our-medium-point (a b) ;;Calcula el punto medio entre dos extremos reales
  (/ (+ a b) 2))

 ;; Aplica el algoritmo de la biseccion para encontrar soluciones de f (f(a) = 0).
 (defun bisect (f a b tol)
  (let ((c (our-medium-point a b))) ;; Definimos c = punto medio de (a , b)
    (cond ((equal (funcall f a) 0) a) ;; Si f(a) = 0 -> a
          ((equal (funcall f b) 0) b) ;; Si f(b) = 0 -> b
          ;; Si f(a) * f(b) > 0 -> nil (ambas son positvas o negativas)
          ((> (* (funcall f a) (funcall f b)) 0) nil)
          ;; Si el tamanyo de (a , b) es menor que tol -> punto medio (a , b)
          ((< (our-distance a b) tol) c)
          ((= (funcall f c) 0) c) ;; Si f(c) = 0 -> c
          ;; Aplicamos la recursividad en uno de los dos intervalos
          ((< (* (funcall f a) (funcall f c)) 0) (bisect f a c tol))
          ((< (* (funcall f b) (funcall f c)) 0) (bisect f c b tol)))))

(bisect #'(lambda (z) (* z z z)) -4 5 0.5)
(bisect #'(lambda(x) (sin (* 6.26 x))) 0.0 0.7 0.001)
(bisect #'(lambda(x) (sin (* 6.28 x))) 1.1 1.5 0.001)
(bisect #'(lambda(x) (sin (* 6.28 x))) 1.1 2.1 0.001)
```

Comentarios

- En este apartado usamos la función principal para ir desechando todos los casos de error y especiales:
  - Si $f(a) = 0$ devuelve a.
  - Si $f(b) = 0$ devuelve b.
  - Si $f(a) * f(b) > 0$ devuelve nil porque o ambas son positivas o ambas negativas.
  - Condición de parada, si el tamaño del intervalo (a b) ¡tol se devuelve el punto medio.
  - Si $f(c) = 0$ devuelve c, hemos encontrado la raíz.

  Si no se cumple ninguno de estos casos, usamos la recursividad para llamar al subintervalo que siga cumpliendo la condición de $f(a) * f(b) < 0$.
- Usamos dos funciones auxiliares: una our-distance que simplemente calcula la distancia entre los extremos de los intervalos; y otra, our-medium-point que calcula el punto medio de un intervalo.

2.2 Allroot

Pseudocódigo

Entrada:
f: función cuyas raíces queremos encontrar

lst: lista ordenada de valores reales
tol: tolerancia que define el criterio de parada

Salida:

Devuelve lista con valores que contienen las raíces en los subintervalos dados

Procesamiento:

Si la lista tiene un solo elemento o ninguno devuelve nil, si no llama al método bisectriz con cada par de valores de la lista ordenados y elimina los que no tienen solución (nil).

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 2.2
;;; allroot (f lst tol)
;;; Encuentra todas las raices localizadas entre dos valores consecutivos
;;; de una lista de valores
;;;
;;; INPUT: f: funcion de un solo parametro real con valores reales cuya
;;; raiz queremos encontrar
;;; lst: lista ordenada de valores reales (lst[i] < lst[i+1])
;;; tol: tolerancia para el criterio de parada: si b-a < tol de la funcion
;;;
;;; OUTPUT: una lista o valores reales que contienen las raices de la funcion
;;; en los subintervalos dados.
;;;
;;; Cuando sgn(f(lst[i])) != sgn(f(lst[i+1])) esta funcion busca
;;; una raiz en el correspondiente intervalo
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun allroot (f lst tol)
  ;; Caso base: Si la lista no tiene elementos o solo 1.
  (if (or (equal lst nil) (equal (rest lst) nil))
      nil
    ;;Lo utilizamos para eliminar los nil de la lista
    (mapcan #'(lambda (x) (unless (null x) (list x)))
    (append (list (bisect f (first lst) (second lst) tol))  ;; lista de soluciones
            (allroot f (rest lst) tol)))))

(allroot #'(lambda(x) (sin (* 6.28 x))) '(0.25 0.75 1.25 1.75 2.25) 0.0001)
(allroot #'(lambda(x) (sin (* 6.28 x))) '(0.25 0.9 0.75 1.25 1.75 2.25) 0.0001)
```

Comentarios

- Esta función simplemente comprueba que ningún elemento de la lista es nil y si no lo son, hacemos un mapcan para filtrar y eliminar aquellos valores nil que puede devolver la función

bisectriz y se llama a la función bisección con los dos primeros valores de la lista recursivamente.

2.3 Allind

Pseudocódigo

Entrada:
f: función cuyas raíces queremos encontrar
a: extremo inferior del intervalo en el que buscamos
b: extremo superior del intervalo en el que buscamos
N: exponente del número de intervalos en el que [a, b] será dividido (2^N)
tol: tolerancia que define el criterio de parada
Salida:
Devuelve $\frac{a+b}{2}$ como solución.
Procesamiento:
Subdividimos en 2^N intervalos y los introducimos en la lista junto con el extremo superior b y llamamos a la función del apartado anterior.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 2.3
;;; allind (f a b N tol)
;;;     Divide en un numero 2^N de invervalos y encuentra todas las raices
;;; de la funcion f en los intervalos obtenidos
;;;
;;; INPUT: f: funcion de un solo parametro real con valores reales cuya
;;; raiz queremos encontrar
;;; a: extremo inferior del intervalo en el que buscamos la raiz
;;; b: b>a extremo superior del intervalo en el que queremos buscar la raiz
;;; N: exponente del numero de intervalos en el que [a, b] va a ser dividido
;;; [a, b] es dividido en 2^N intervalos
;;; tol: tolerancia para el criterio de parada: si b-a < tol de la funcion
;;;
;;; OUTPUT: devuelve (a+b)/2 como solucion
;;;
;;; El intervalo (a, b) es dividido en intervalos (x[i], x[i+1]) con
;;; x[i] = a + i*dlt; una raiz es buscada en cada intervalo, y todas las
;;; raices encontradas se juntan en una lista que se devuelve
;;;
;;; Pista:
;;; Uno puede encontrar una manera de usar allroot para implementar esta funcion.
;;; Esto es posible por supuesto, pero hay una forma simple de hacerlo recursivo
;;; sin usar allroot.
;;;
```

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

(defun div-int (a b N) ;; Dividimos un intervalo (a , b) en 2^ns intervalos
  (let ((c (our-medium-point a b))) ;;Definimos c = punto intermedio
    (if (equal N 0) ;;Caso base: N = 0 -> 1 intervalo
        (list a)
      ;; dividimos cada intervalo por la mita hasta que n = 0
      (append (div-int a c (- N 1)) (div-int c b (- N 1))))))

;; La funcion anterior nos devuelve una lista con todos las divisiones del interval

(div-int 0 8 3)

(defun allind (f a b N tol)
  ;;Anyadimos b y calculamos todas las soluciones con la funcion del apartado 2.2
  (allroot f (append (div-int a b N) (list b)) tol))

(allind #'(lambda(x) (sin (* 6.28 x))) 0 3 2 0.01) ;
(allind #'(lambda(x) (sin (* 6.28 x))) 0.1 2.25 3 0.0001)
(allind #'(lambda(x) (sin (* 6.28 x))) 0.25 2.25 3 0.01) ;
```

Comentarios

- La función principal solo se encarga de añadir a la lista el extremo superior b y llama a la función auxiliar div-int.
- La función div-int llama a la función del punto medio del apartado anterior y va añadiendo a la lista ordenadamente cada extremo hasta llegar a completar los 2^N intervalos.

3. Combinación de Listas

3.1 Combine-elt-lst

Pseudocódigo

Entrada:
elem: elemento que combinaremos con los de la lista
lst: lista con la que se combina el elemento
Salida:
Devuelve la lista con las combinaciones.
Procesamiento:
Si la lista está vacía devolvemos nil, si no se van añadiendo combinaciones recursivamente.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 3.1
```

```
;;; combine-elt-lst (elem lst)
;;; Combina un elemento dado con todos los elementos de una lista
;;;
;;; INPUT: elt: elemento que se combinara con los de la lista
;;; lst: lista con la que se combinara el elemento
;;;
;;; OUTPUT: devuelve la lista con las combinaciones del elemento y la lista dadas.
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun combine-elem-lst (elem lst)
  ;; Con esta definicion no repetimos codigo
  (let ((k (list(list elem (first lst)))))
  (cond ((equal lst nil) nil) ;; Caso base: lst nil -> nil
        ;; Caso base: si la lista tiene un elemento -> k
        ((equal (rest lst) nil) k)
        ( t (append k (combine-elem-lst elem (rest lst)))))))

(combine-elem-lst 'a nil)
(combine-elem-lst 'a '(1))
```

Comentarios

- La función comprueba:
  - Si la lista está vacía devuelve nil.
  - Si la lista tiene solo un elemento lo devuelve.
  - Si la lista posee varios elementos añade el primero y llama recursivamente a la misma función con el resto de la lista.

3.2 Combine-lst-lst

Pseudocódigo

Entrada:
lst1: primera lista del producto cartesiano
lst2: segunda lista del producto cartesiano
Salida:
Devuelve la lista resultado del producto cartesiano.
Procesamiento:
Devuelve nil si ambas listas están vacías, si no se llama a la función anterior con cada elemento de la primera lista y la segunda lista entera.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 3.2
```

```
;;; combine-lst-lst (lst1 lst2)
;;; Calcula el producto cartesiano de dos listas
;;;
;;; INPUT: lst1: primera lista sobre la que se realizara el producto cartesiano
;;; lst2: segunda lista sobre la que se realizara el producto cartesiano
;;;
;;; OUTPUT: devuelve la lista resultado del producto cartesiano de las anteriores
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun combine-lst-lst (lst1 lst2)
  ;; Caso base: Si ambas lista son nil -> nil nil
  (if (or (equal lst1 nil) (equal lst2 nil))
    ;; Combinamos el elemento primero de la primera lista con la segunda
    ;; y aplicamos recursividad sobre la lst1
    (append (combine-elem-lst (first lst1) lst2) (combine-lst-lst (rest lst1) lst2)

(combine-lst-lst nil nil)
(combine-lst-lst '(a b c) nil)
(combine-lst-lst NIL '(a b c))
(combine-lst-lst '(a b c) '(1 2))
(combine-lst-lst '(a b c d e f) '(1 2))
```

Comentarios

- Comprueba si alguna lista es nil y si no lo son, llama a la función del apartado anterior para el primer elemento de la lista 1 y por recursividad a esta propia función pasándole como primer argumento el resto de la lista.

3.3 Combine-list-of-lsts

Pseudocódigo

Entrada:
lstolsts: todas las listas a combinar
Salida:
Devuelve la lista resultado de la combinación de todas las dadas.
Procesamiento:
Devuelve nil si la lista o alguna de sus sublistas está vacía, si la lista solo tiene un vector cada uno de sus elementos se convierten en listas independientes y s i no se devuelve la combinación de todas las listas dadas.

Código

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; EJERCICIO 3.3
;;; combine-list-of-lsts (lstolsts)
```

```
;;; Calcula todas las posibles disposiciones pertenecientes a N listas
;;; de forma que en cada disposicion aparezca solo un elemento de cada lista
;;;
;;; INPUT: lstolsts: todas las listas que se combinaran
;;;
;;; OUTPUT: devuelve una lista resultado de la combinacion de todas las dadas
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun flatten (lst)  ;; Funcion que sirve para quitar parentesis a las listas
  (cond
   ((null lst) NIL)
   ((atom (first lst))
    (cons (first lst) (flatten (rest lst))))
   (t (append (flatten (first lst)) (flatten (rest lst))))))

(defun cl (lstolsts)
  (if (equal lstolsts nil) ;; Si lstolsts es nil devolvemos (nil)
      (return-from cl nil))
  (let ((comb (cl (rest lstolsts)))) ;;Definimos (cl (rest lstoslts)) = comb
    ;;Si ha finalizado de recorrer la lista de listas devolvemos todo lo anterior
    (if (eql comb nil)
       (first lstolsts)
      (combine-lst-lst (first lstolsts) comb)))) ;; LLamada recursiva

(defun combine-list-of-lsts (lstolsts)
  (cond ((equal lstolsts nil) '(nil)) ;; Si el vector de listas es nil -> nil
        ;; Si alguna lista de vector es nil -> nil
        ((some #'(lambda (z) (equal z nil)) lstolsts) nil)
        ;; Si el vector tiene solo una lista
        ;; -> Convertimos cada elemento en una lista independiente
        ((equal (rest lstolsts) nil) (mapcar #'list (first lstolsts)))
        ;; En caso contrario, llamamos a la funcion recursciva cl
        ( t (mapcar #'(lambda (x) (flatten x)) (cl lstolsts)))))

(combine-list-of-lsts '((a b c) (+ -) (1 2 3 4) (m o r e)))
(combine-list-of-lsts '(() (+ -) (1 2 3 4)))
(combine-list-of-lsts '((a b c) () (1 2 3 4)))
(combine-list-of-lsts '((a b c) (1 2 3 4) ()))
(combine-list-of-lsts '((1 2 3 4)))
(combine-list-of-lsts '((a b c) (+ -) (1 2 3 4)))
```

Comentarios

- La función principal contempla todos los casos posibles:
  - Si el vector de listas es nil devuelve nil.

- Si alguna lista de vectores es nil devuelve nil.
- Si el vector tiene una sola lista, convierte cada elemento suyo en una lista.
- Si no está en ningún caso anterior llama a la función auxiliar recursiva cl y le aplica otra función auxiliar flatten.

- La función cl comprueba si la lista pasada es nil y si no recorre la lista de listas de manera recursiva hasta que termina y devuelve todas las combinaciones.
- La función flatten se usa para quitar los paréntesis a las listas que devuelve la función recursiva anterior, con el objetivo de aplanar y simplificar las expresión.

4. Inferencia en Lógica Proposicional

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

4.1.1 Positive-literal-p

Pseudocódigo

Entrada:
lstolsts: x un átomo
Salida:
Devuelve true si x es un literal, nil en caso contrario.
Procesamiento:
Comprueba que x es un átomo y no es un conector, true o nil.

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

;; 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
```

Comentarios

La función comprueba que x que se cumplen a la vez dos condiciones a través d un and. El hecho de que x sea un átomo y no sea un valor de verdad (true o nil).

4.1.2 Negative-literal-p

Pseudocódigo

Entrada:
Literal del que se quiere saber si es negativo
Salida:
True en caso que de sea un literal negativo, nil en caso contrario.
Procesamiento:
Comprobamos que el argumento es una lista y tiene dos elementos: el primero es un conector unario y el segundo un literal positivo.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.2
;; Predicado para determinar si una expresion
;; es un literal negativo
;;
;; RECIBE    : expresion x
;; EVALUA A  : T si la expresion es un literal negativo,
;;             NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun negative-literal-p (x)
  (and
   (listp x)                                  ;; Vemos que es una lista
   (null (cddr x))                            ;; comprobamos que tiene dos elementos
   (unary-connector-p (first x))              ;; El primero es un concetor unario
   (positive-literal-p (first (rest x)))))    ;;El segundo un literal positivo

;; 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
```

4.1.3 Literal-p

Pseudocódigo

Entrada:
Literal del que se quiere saber si es un literal
Salida:
True en caso que de sea un literal, nil en caso contrario.
Procesamiento:
Comprobamos que el argumento sea un literal positivo o literal negativo.

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

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

4.1.4 wff-infix-p

Pseudocódigo

Entrada:
x expresion

Salida:
T si x esta en formato prefijo,
Procesamiento:
Si n no es nil, devolvemos or (n es un literal)
Si n es una expresión negativa devolvemos (infijo rest)
Si n es una expresión binaria devolvemos and (infjo op1) (infijo op2)
Si n es expresión binaria devolvemos (infijo op1) and (nop-verify rest))

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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)
    (or (literal-p x)
        (and (listp x)
             (let ((op1 (car x))
                   (exp1 (cadr x))
                   (list_exp2 (cddr x)))
               (cond
                ((unary-connector-p op1)
                 (and (null list_exp2)
                      (wff-infix-p exp1)))
                ((n-ary-connector-p op1)
                 (null (rest x)))
                ((binary-connector-p exp1)
                 (and (wff-infix-p op1)
                      (null (cdr list_exp2))
                      (wff-infix-p (car list_exp2))))
                ((n-ary-connector-p exp1)
                 (and (wff-infix-p op1)
                      (nop-verify exp1 (cdr x))))
                 (t NIL)))))))

;; Verifica si una expresion de la forma op <wff> ... op <wff>
(defun nop-verify (op exp)
  (or (null exp)
      (and (equal op (car exp))
           (wff-infix-p (cadr exp))
           (nop-verify op (cddr exp)))))

;;
```

```
;; 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
```

Comentarios

La función va recorriendo los posibles escenarios que presenta la expresión:

- Si la expresion es nil, devuelve nil
- Si es un literal, ya está en formato infijo, lo que implica true
- Si es una literal negativo, se comprueba que la expresión es de la forma (conector expresión-infijo). Si ocurre esto devolvemos true, y nil en caso contrario.
- Si es una expresión con un conector binario, se comprueba que tenga la estructura (op1 concetor op2).
- Si es un conector n-ario lo que se hace es que se comprueba que la expresión sea de la forma (op1 concetor op2 concetor ... opn) (que lo hace la función nop-verify)

4.1.5 defun infix-to-prefix

Pseudocódigo

Entrada:
x expresion infijo
Salida:
x en expresión prefijo
Procesamiento:
cuando exp infijo,
Si exp es literal devuelve exp
Si es conector unario devolvemos (inf-pref resto exp)
Si es conector binario devolvemos devolvemos (concetor inf-pref op1 inf-pref op2)
Si es conector n-ario devolvemos (conector inf-pref (exp sin conectores))

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.5
;;
;; Convierte FBF en formato infijo a FBF en formato prefijo
;;
;; RECIBE   : FBF en formato infijo
;; EVALUA A : FBF en formato prefijo
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun infix-to-prefix (wff)
  (when (wff-infix-p wff)
    (if (literal-p wff)
        wff
      (let ((op1 (first wff))
            (rst_ele (rest wff))
            (conector (first (rest wff))))
        (cond
         ((unary-connector-p op1)
          (list op1 (infix-to-prefix (second wff))))
         ((binary-connector-p conector)
          (list conector (infix-to-prefix op1) (infix-to-prefix (third wff))))
         ((n-ary-connector-p conector)
          (cond
           ((null rst_ele) wff)
           ((null (cdr rst_ele))
            (infix-to-prefix op1))
           (t (cons conector (mapcar #'(lambda (x) (infix-to-prefix x))
            (list-def wff conector) )))))
         (t nil))))))

(defun list-def (wff conector)
  (remove-if #'(lambda (x) (equal conector x)) wff))
```

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


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

(infix-to-prefix '(~ ((~ p) v q v (~ r) v (~ s))))
;; (~ (V (~ P) Q (~ R) (~ S)))


(infix-to-prefix
 (prefix-to-infix
  '(V (~ P) Q (~ R) (~ S))))
;;-> (V (~ P) Q (~ R) (~ S))

(infix-to-prefix
 (prefix-to-infix
  '(~ (V (~ P) Q (~ R) (~ S)))))
;;-> (~ (V (~ P) Q (~ R) (~ S)))


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

(infix-to-prefix '(~ ((~ p) v q v (~ r) v (~ s))))
;; (~ (V (~ P) Q (~ R) (~ S)))

(infix-to-prefix  (prefix-to-infix '(^ (v p (=> a (^ b (~ c) d))))))
; '(v p (=> a (^ b (~ c) d))))
(infix-to-prefix  (prefix-to-infix '(^ (^ (<=> p (~ q)) p ) e)))
; '(^ (^ (<=> p (~ q)) p ) e))
```

```
(infix-to-prefix (prefix-to-infix '( v (~ p) q (~ r) (~ s))))
; '( v (~ p) q (~ r) (~ s)))

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

Comentarios

En este apartado en el código, nos ayudamos de la función list-def que lo que hace es eliminar de una lista el conector n-ario. ¿Por qué hacemos esto? Porque si tenemos una expresión de la forma (op1 conector op2 conector ... opn) y queremos pasarla a prefijo, tenemos que llegar a algo del estilo (conector op1 op2 op3..). Por este motivo existe esta función. Por tanto, luego solo habrá que pasar a notación prefijo los distintos operadores y añadir el conector eliminado al principio de la expresión resultante.

4.1.6 Clause-p

Pseudocódigo

Entrada:
FBF en formato prefijo
Salida:
Devuelve true si wff es una cláusula, nil en caso contrario.
Procesamiento:
clause-p
Si wff nil devolvemos nil
Si wff no lista devolvemos nil
Si first wff == 'v devolvemos clause-p-aux
Otro caso devolvemos nil
clause-p-aux
Si wff nil devolvemos t
Si first wff no literal devolvemos nil
En otro caso devolvemos clause-p-aux (rest wff)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.6
;; Predicado para determinar si una FBF es una clausula
;;
;; RECIBE    : FBF en formato prefijo
;; EVALUA A : T si FBF es una clausula, NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun clause-p-aux (wff)
  (cond ((null wff) t)
        ((equal nil (literal-p (first wff))) nil)
```

```
        (t (clause-p-aux (cddr wff))))))

(defun clause-p (wff)
  (cond
   ((null wff) nil)
   ((not (listp wff)) nil)
   ((equal (first wff) 'v) (clause-p-aux (rest wff)))
   (t nil)))

;;
;; 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
```

Comentarios

Hacemos una función auxiliar que es la que hace la recursión mirando si cada elemento es un literal. La función principal tan solo se encarga de comprobar una serie de requisitos

4.1.7 cnf-p

Pseudocódigo

Mismo pseudocódigo que la función anterior lo único que cambia es que en la auxiliar en vez de comprobar si es un literal comprueba que es una cláusula

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.7
;; Predicado para determinar si una FBF esta en FNC
;;
;; RECIBE    : FFB en formato prefijo
;; EVALUA A  : T si FBF esta en FNC con conectores,
;;             NIL en caso contrario.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
(defun cnf-p-aux (wff)
  (cond ((null wff) t)  ;; Si es nil --> nil
        ((equal nil (clause-p (first wff))) nil)
        (t (cnf-p-aux (rest wff)))))

(defun cnf-p (wff)
  (cond
   ((null wff) nil)       ;;Si wff nil --> nil
   ((not (listp wff)) nil)  ;;Si wff no es una lista --> nil
   ((equal (first wff) '^) (cnf-p-aux (rest wff)))
   (t nil)))


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

4.2 Algoritmo de Transformación de una FBF a FNC

4.2.1 Eliminate-Biconditional

Pseudocódigo

Entrada:
FBF: frase bien formada
Salida:
FBF equivalente eliminando la bicondición
Procesamiento:
Si es nulo o literal lo devuelve
Si el primer elemento es un conector bicondicional sustituimos por la formula equivalente
Si no estamos en ningún caso anterior, hacemos un mapcar del resto de elementos usando la bicondicional

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

(defun eliminate-biconditional (wff)
  (if (or (null wff) (literal-p wff))
      wff
    (let ((connector (first wff)))
      (if (eq connector +bicond+)
          (let ((wff1 (eliminate-biconditional (second wff)))
                (wff2 (eliminate-biconditional (third  wff))))
            (list +and+
                  (list +cond+ wff1 wff2)
                  (list +cond+ wff2 wff1)))
        (cons connector
              (mapcar #'eliminate-biconditional (rest wff)))))))

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

```
;;          (=> (^ S (~ Q)) (^ (=> P Q) (=> Q P))))
```

Comentarios

Función dada, solo había que comentarla, para seguir los comentarios en profundidad, consultar código

4.2.2 Eliminate-Conditional

Pseudocódigo

Entrada:
wff: frase bien formada
Salida:
wff equivalente sin el conector condición
Procesamiento:
Si wff es nula o literal la devuelve
Si el primer elemento es el conector buscado, se sustituye por su equivalente y se llama recursivamente a la funcion
Si no es ninguno de los casos anteriores, hacemos un mapcar del resto de wff usando la propia funcion recursivamente

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.2
;; Dada una FBF, que contiene conectores => evalua a
;; una FBF equivalente que no contiene el connector =>
;;
;; RECIBE   : wff en formato prefijo sin el connector <=>
;; EVALUA A : wff equivalente en formato prefijo
;;            sin el connector =>
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-conditional (wff)
  (if (or (null wff) (literal-p wff))
      wff
    (let ((connector (first wff)))
      (if (eq connector +cond+)
          (let ((wff1 (eliminate-conditional (second wff)))
                (wff2 (eliminate-conditional (third  wff))))
            (list +or+
                  (list +not+ wff1)
                  wff2))
        (cons connector
              (mapcar #'eliminate-conditional (rest wff)))))))

;;
;; EJEMPLOS:
```

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

Comentarios

En este caso se ha seguido el guión del ejemplo anterior.
Solo se modificó la equivalencia para que se correspondiera con la de la condición simple

4.2.3 Reduce-Scope-Of-Negation

Pseudocódigo

Entrada:
wff: frase bien formada
Salida:
frase bien formada equivalente tras eliminar la doble negación y aplicar DeMorgan
Procesamiento:
Si es nulo o literal devuelve la propia wff
Si el primer elemento es un not y el segundo es un literal negativo, reduce la doble negación
Si el primer elemento es un not y el segundo no es un literal negativo o bien el conector interno es un and/or o es una negación. Si es un and/or uso función auxiliar y aplico DeMorgan si no llamo recursivamente
Si el primer conector no es un not llamo recursivamente directamente a la función.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.3
;; Dada una FBF, que no contiene los conectores <=>, =>
;; evalua a una FNF equivalente en la que la negacion
;; aparece unicamente en literales negativos
;;
;; RECIBE    : FBF en formato prefijo sin conector <=>, =>
;; EVALUA A : FBF equivalente en formato prefijo en la que
;;              la negacion  aparece unicamente en literales
;;              negativos.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun exchange-and-or (connector)
  (cond
   ((eq connector +and+) +or+)
   ((eq connector +or+) +and+)
   (t connector)))
```

```
(defun reduce-scope-of-negation (wff)
  (if (or (null wff) (literal-p wff))
    wff
      (if (eq +not+ (first wff))
        (if (negative-literal-p (second wff))
          (second (second wff))
          (if (eq +not+ (first (second wff)))
            (reduce-scope-of-negation (rest (second wff)))
            (cons (exchange-and-or (first (second wff)))
              (mapcar #'(lambda(x)
                (reduce-scope-of-negation (list +not+ x))) (rest (second wff))))))
        (cons (first wff)
          (mapcar #'reduce-scope-of-negation (rest wff))))))

;;
;;   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))
```

Comentarios

La función se basa en que para aplicar tanto DeMorgan como la doble negación, lo primero que tengo que encontrar es un not.
También usa fuertemente que es formato prefijo y que los conectores solo pueden ser and u or.

4.2.4 CNF

Pseudocódigo

Entrada:
wff: frase bien formada
Salida:
La fnc (forma normal conjuntiva) equivalente a la fbf eliminando conectores.
Procesamiento:
Si está ya en fnc la devuelve como está.
Si es un literal, devuelve una lista con conector and y dentro otra con el or y la wff
Si es un and o un or llama a las funciones auxiliares

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

```
;; evalua a una FNC equivalente en FNC con conectores ^, v
;;
;; RECIBE   : FBF en formato prefijo sin conector <=>, =>,
;;            en la que la negacion aparece unicamente
;;            en literales negativos
;; EVALUA A : FBF equivalente en formato prefijo FNC
;;            con conectores ^, v
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun combine-elt-lst (elt lst)
  (if (null lst)
      (list (list elt))
    (mapcar #'(lambda (x) (cons elt x)) lst)))
(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)))))))

(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))))))

(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)))

(defun cnf (wff)
```

```
  (cond
   ((cnf-p wff) wff)
   ((literal-p wff)
    (list +and+ (list +or+ 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))))))))))


(cnf 'a)

(cnf '(v (~ a) b c))
(print (cnf '(^ (v (~ a) b c) (~ e) (^ e f (~ g) h) (v m n) (^ r s q)
(v u q) (^ x y))))
(print (cnf '(v (^ (~ a) b c) (~ e) (^ e f (~ g) h) (v m n) (^ r s q)
(v u q) (^ x y))))
(print (cnf '(^ (v p (~ q)) a (v k r (^ m n)))))
(print (cnf '(v p q (^ r m) (^ n a) s )))
(exchange-NF '(v p q (^ r m) (^ n a) s ))
(cnf '(^ (v a b (^ y r s) (v k l)) c (~ d) (^ e f (v h i) (^ o p))))
(cnf '(^ (v a b (^ y r s)) c (~ d) (^ e f (v h i) (^ o p))))
(cnf '(^ (^ y r s (^ p q (v c d))) (v a b)))
(print (cnf '(^ (v (~ a) b c) (~ e) r s
                (v e f (~ g) h) k (v m n) d)))
;;
(cnf '(^ (v p (~ q)) (v k r (^ m n))))
(print  (cnf '(v (v p q) e f (^ r m) n (^ a (~ b) c) (^ d s))))
(print (cnf '(^ (^ (~ y) (v r (^ s (~ x)) (^ (~ p) m (v c d)))
(v (~ a) (~ b))) g)))

;;
;; EJEMPLOS:
;;
(cnf NIL)               ; NIL
(cnf 'a)                ; (^ (V A))
(cnf '(~ a))            ; (^ (V (~ A)))
(cnf '(V (~ P) (~ P))) ; (^ (V (~ P) (~ P)))
(cnf '(V A))            ; (^ (V A))
(cnf '(^ (v p (~ q)) (v k r (^ m n))))
;;;    (^ (V P (~ Q)) (V K R M) (V K R N))
(print  (cnf '(v (v p q) e f (^ r m) n (^ a (~ b) c) (^ d s))))
;;; (^ (V P Q E F R N A D)       (V P Q E F R N A S)
```

```
;;;       (V P Q E F R N (~ B) D)   (V P Q E F R N (~ B) S)
;;;       (V P Q E F R N C D)       (V P Q E F R N C S)
;;;       (V P Q E F M N A D)       (V P Q E F M N A S)
;;;       (V P Q E F M N (~ B) D)   (V P Q E F M N (~ B) S)
;;;       (V P Q E F M N C D)       (V P Q E F M N C S))
;;;
(print
 (cnf '(^ (^ (~ y) (v r (^ s (~ x))
                    (^ (~ p) m (v c d)))(v (~ a) (~ b))) g)))
;;;(^ (V (~ Y)) (V R S (~ P)) (V R S M)
;;;     (V R S C D) (V R (~ X) (~ P))
;;;     (V R (~ X) M) (V R (~ X) C D)
;;;     (V (~ A) (~ B)) (V G))
```

Comentarios

Función dada para comentar, los comentarios se encuentran en el código.

4.2.5 Eliminate-Connectors

Pseudocódigo

Entrada:
cnf: frase bien formada en forma normal conjuntiva (cnf)
Salida:
fbf en forma normal conjuntiva sin ands ni ors
Procesamiento:
Realiza el mapcar del resto del resto de la cnf

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.5:
;;
;; Dada una FBF en  FNC
;; evalua a lista de listas sin conectores
;; que representa una conjuncion de disyunciones de literales
;;
;; RECIBE    : FBF en FNC con conectores ^, v
;; EVALUA A  : FBF en FNC (con conectores ^, v eliminaos)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun eliminate-connectors (cnf)
  (mapcar #'rest (rest cnf)))

(eliminate-connectors 'nil)
(eliminate-connectors (cnf '(^ (v p (~ q)) (v k r (^ m n)))))
```

```
(eliminate-connectors
 (cnf '(^ (v (~ a) b c) (~ e) (^ e f (~ g) h) (v m n) (^ r s q) (v u q) (^ x y))))

(eliminate-connectors (cnf '(v p  q  (^ r  m)  (^ n  q)  s )))
(eliminate-connectors (print (cnf '(^ (v p  (~ q)) (~ a) (v k  r  (^ m
n))))))

(eliminate-connectors '(^))
(eliminate-connectors '(v))
(eliminate-connectors '(^ (v p (~ q)) (v) (v k r)))
(eliminate-connectors '(^ (v a b)))

;;    EJEMPLOS:
;;

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

Comentarios

Se basa en la idea de que al estar en formato prefijo, lo primero serán conectores y los elimina.

4.2.6 WFF-Infix-To-CNF

Pseudocódigo

Entrada:
wff: frase bien formada en formato infijo
Salida:
devuelve su forma normal conjuntiva sin conectores equivalente
Procesamiento:
Si la wff es nula o un literal la devuelve tal cual
Si no, hace llamadas a múltiples funciones anteriores

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.6
;; Dada una FBF en formato infijo
;; evalua a lista de listas sin conectores
;; que representa la FNC equivalente
;;
;; RECIBE    : FBF
;; EVALUA A : FBF en FNC (con conectores ^, v eliminados)
;;
```

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

(defun wff-infix-to-cnf (wff)
  (if (or (null wff) (literal-p wff))
    wff
    (eliminate-connectors
      (cnf
        (reduce-scope-of-negation
          (eliminate-conditional
            (eliminate-biconditional
              (infix-to-prefix wff))))))))

;;
;; EJEMPLOS:
;;
(wff-infix-to-cnf 'a)
(wff-infix-to-cnf '(~ a))
(wff-infix-to-cnf  '( (~ p) v q v (~ r) v (~ s)))
(wff-infix-to-cnf  '((p v (a => (b ^ (~ c) ^ d))) ^
((p <=> (~ q)) ^ p) ^ e))
;; ((P (~ A) B) (P (~ A) (~ C)) (P (~ A) D) ((~ P) (~ Q)) (Q P) (P) (E))
```

Comentarios

En esta función se prueba que la buena modularización del código facilita el trabajo para realizar una función que a priori es mucho más complicada.
Consiste básicamente en llamadas a distintas funciones una detrás de otras.
Primero convertimos a prefijo.
Después eliminamos bicondición, condición, reducimos al ámbito de la negación.
Transformamos cnf y por último, eliminamos conectores.

4.3 Eliminación de literales y cláusulas repetidas, tautoligías, y cláusulas subsumidas

4.3.1 eliminated-reapeted-literals

Pseudocódigo

Entrada:
k es una cláusula de literales
Salida:
Clausula sin literales repetidos
Procesamiento
Si k nil devolvemos nil
Si firts k no repetido devolvemos eliminar-literales (rest k + first k)

Otro caso devolvemos eliminar-literales rest k

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; 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-aux (k)
  (cond
   ((equal k nil ) '(nil))
   ((equal (find (first k) (rest k) :test #'equal) nil)
               (cons (first k) (eliminate-repeated-literals (rest k))))
   (t (eliminate-repeated-literals (rest k)))))

(defun eliminate-repeated-literals (k)
  (if (null k)
      nil
    (eliminate-repeated-literals-aux (remove nil k :test #'equal))))

(eliminate-repeated-literals '(nil nil nil))

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

Comentarios

Se concatena el primer elemento ya que si no, se perdería al llamar a la función recursiva con el resto de la lista.
La función auxiliar se necesita ya que si no, habría un problema con la lista (nil nil ... nil) porque el find devolvería nil lo que nos daría a entender que ningún elemento se repite o que el que se repite es el elemento nil.
Como no hay forma de diferenciar esta situación se manda a la función auxiliar la lista con todos los nil borrados a través de la función remove.

4.3.2 eliminated-repeated-clauses

La función contenido implementa el contenido de conjuntos. La función our-equal implemente la igualdad de cojuntos.

Pseudocódigo

Entrada:
FBF en FCN
Salida:
FBN equivalente sin cláusulas repetidas.
Procesamiento:
Si cnf es null devolvemos nil
Si first cnf repetido, devolvemos eliminar-clausulas (rest cnf + first cnf)
Otro caso, devolvemos eliminar-clausulas rest k

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.2
;; eliminacion de clausulas repetidas en una FNC
;;
;; RECIBE   : cnf - FBF en FNC (lista de clausulas, conjuncion implicita)
;; EVALUA A : FNC equivalente sin clausulas repetidas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun eliminate-repeated-clauses (cnf)
  (let ((operador (eliminate-repeated-literals (first cnf))))
  (cond
   ((equal cnf nil) nil)  ;; Si cnf nil --> nil
   ((equal (find operador (rest cnf) :test #'our-equal) nil)
   (cons operador (eliminate-repeated-clauses (rest cnf))))
   (t (eliminate-repeated-clauses (rest cnf))))))

;; Funcion que nos determina si L1 esta contenida en L2 (contenido de conjuntos)
(defun contenido (l1 l2)
  (if (equal nil l1)
      t
    (and (not (equal nil (find (first l1) (eliminate-repeated-literals l2)
                        :test #'equal)))
         (contenido (rest l1) l2))))

;; Definimos que dos clausulas si l1 esta contenido en l2 y viceversa
(defun our-equal (l1 l2)
  (and (contenido l1 l2) (contenido l2 l1)))

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

Comentarios

Se definen dos funciones auxiliares muy importantes que son el contenido e igualdad de conjuntos, ya que para determinar si dos cláusulas son iguales no importa el orden en el que aparezcan.

4.3.3 subsume

Pseudocódigo

Entrada:
k1, k2 cláusulas
Salida:
lista de K1 si K1 subsume K2
Procesamiento:
Si k1 contenido en k2 devolver lista k1
en otro caso nil

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

(defun subsume (K1 K2)
  (if (equal t (contenido k1 k2))
      (list k1)
    nil))

;;
;;  EJEMPLOS:
;;
(subsume '(a) '(a b (~ c)))
;; ((a))
(subsume NIL '(a b (~ c)))
;; (NIL)
(subsume '(a b (~ c)) '(a) )
;; NIL
(subsume '( b (~ c)) '(a b (~ c)) )
;; (( b (~ c)))
(subsume '(a b (~ c)) '( b (~ c)))
;; NIL
(subsume '(a b (~ c)) '(d  b (~ c)))
```

```
;;  nil
(subsume '(a b (~ c)) '((~ a) b (~ c) a))
;;  ((A B (~ C)))
(subsume '((~ a) b (~ c) a) '(a b (~ c)) )
;;  nil
```

Comentarios

Esta función es muy sencilla de implementar si utilizamos el contenido de conjuntos definido en el apartado anterior.

4.3.4 eliminate-subsumed-clauses

Pseudocódigo

remove-subsumed
Esta función trabaja con dos listas que al comienzo son cnf. La lista final siempre se mantiene igual y la otra va iterando por los elementos de la lista comprobando cláusula a cláusula si está subsume a otra de final.
Entrada:
l1 y final son FBF en FNC
Salida:
FBF en FNC sin cláusulas subsumidas
Procesamiento:
Si l1 y final sin nil devolvemos nil
Si first l1 is-subsumed final devolvemos

is-subsumed
Entrada:
K es una cláusula
l es una FBF en FNC
Salida:
t si K subsume en alguna de las cláusulas de l, nil en caso contrario
Procesamiento:
Si l es nil devolvemos nil
((first l) subsumido por y no es él mismo) OR (is-subsumed k resto l1)
En otro caso devolvemos (first l1) + (is-subsumed k resto l1)

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

```
(defun eliminate-subsumed-clauses (cnf)
  (remove-subsumed cnf cnf))

(defun is-subsumed (k l)
  (unless (null l)
    (or (and (not(null (subsume (car l) k)))
             (not (eq k (car l))))
        (is-subsumed k (rest l)))))

(is-subsumed '(a b c) '( (a b) (c d)))
(is-subsumed '(a b c) '((a b c) '(a b c)))

(defun remove-subsumed (l1 final)
  (cond
   ((null l1) nil)
   ((is-subsumed (first l1) final) (remove-subsumed (rest l1) final))
   (t (cons (first l1) (remove-subsumed (rest l1) final )))))
```

Comentarios

Sabemos que se podría haber hecho de alguna forma utilizando la diferencia de conjuntos, pero nunca llegamos a la solución correcta de esta manera. Seguramente con la diferencia de conjuntos salga una forma mucho más eficiente.

4.3.5 tautology-p

Hay que mencionar que equal-inv es una función que hace lo siguiente:
Si a es un literal positivo y b es un literal negativo compruebo si a es igual al segundo elemento de b.
Si a es negativo y b es positivo hago la misma comprobación pero al revés. Ejemplo: Si a = p y b = (NO p). Compruebo que si p == q que en este caso no lo son.

Por otro lado, en-tauto busca si respecto de un átomo hay tautología en una lista de cláusulas en función de la igualdad de cláusulas definida por equal-inv. Si hay tautología devuelve t y nil en caso contrario.

Pseudocódigo

Entrada:
k cláusula
Salida:
t si k es una tautología, nil en caso contrario.
Procesamiento:
Si k es nil devolvemos nil (covención)
En caso contrario, devolvemos (en-tauto (rest k) (first k)) or tautologia-p (rest k)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.5
;; Predicado que determina si una clausula es tautologia
;;
;; RECIBE   : K (clausula)
;; EVALUA a : T si K es tautologia
;;            NIL en caso contrario
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun tautology-p (k)
  (let ((op (first k)))
    (cond
     ((equal k nil) nil)
     (t (or (tautology-p (rest k)) (en-tauto op k))))))

(defun equal-inv (a b)
  (cond
   ((and (positive-literal-p a) (negative-literal-p b))
    (equal (second b) a))
  ((and (negative-literal-p a) (positive-literal-p b))
    (equal (second a) b))
   (t nil)))

(equal-inv 'a '(~ a))

(defun en-tauto (a k)
  (if (equal (find a k :test #'equal-inv) nil)
      nil
    t))

(en-tauto 'b '(a a b (~ b) c))

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

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

4.3.6 eliminate-tautologies

Pseudocódigo

Entrada:
FBF en FNC
Salida:
FBF equivalente en sin tautologías
Procesamiento:
Si es cnf es nil devuelves nil
Si es tautologia devolvemos eliminar-tautologias resto
En cualquier caso devolvemos (first cnf) + eliminar-tautologias (resto)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.6
;; eliminacion de clausulas en una FBF en FNC que son tautologia
;;
;; RECIBE   : cnf - FBF en FNC
;; EVALUA A : FBF en FNC equivalente a cnf sin tautologias
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun eliminate-tautologies (cnf)
  (cond
   ((null cnf) nil)
   ((tautology-p (first cnf)) (eliminate-tautologies (rest cnf)))
   (t (cons (first cnf) (eliminate-tautologies(rest cnf))))))
```

4.3.7 eliminate-tautologies

Pseudocódigo

Entrada:
FBF en FNC
Salida:
La FCN simplificada
Procesamiento: eliminamos-tautologias (eliminamos-clausulas-subsumidas(eliminamos-clausulas-repetidas)

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

```
;;              y sin clausulas subsumidas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

4.4 Construcción de RES

4.4.1 Extract-Neutral-Clauses

Pseudocódigo

Entrada:
lamda: literal positivo
cnf: frase bien formada simplificada
Salida:
Clausulas lamdas neutras de la cnf
Procesamiento:
Creamos dos funciones auxiliares:

- Una saca las claúsulas no positivas
- La otra saca las claúsulas no negativas

Ambas funciones auxiliares comprueban si lamda o cnf es nulo y devuelven nil.
Si no, comprueban si tienen como miembro al lamda/no lamda y se llaman recursivamente
La función principal es la intersección de las anteriores

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.1
;; Construye el conjunto de clausulas lambda-neutras para una FNC
;;
;; RECIBE   : cnf    - FBF en FBF simplificada
;;            lamda - literal positivo
;; EVALUA A : cnf_lamda^(0) subconjunto de clausulas de cnf
;;            que no contienen el literal lambda ni ~lambda
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun extract-neutral-clauses (lamda cnf)
  (intersection (extract-no-positive-clauses lamda cnf)
        (extract-no-negative-clauses lamda cnf)))

(defun extract-no-positive-clauses (lamda cnf)
  (if (or (null cnf) (literal-p cnf))
    NIL
    (if (equal (member lamda (first cnf) :test #'equal) NIL)
      (cons (first cnf) (extract-no-positive-clauses lamda (rest cnf)))
```

```
      (extract-no-positive-clauses lamda (rest cnf)))))

(defun extract-no-negative-clauses (lamda cnf)
  (if (or (null cnf) (literal-p cnf))
    NIL
    (if (equal (member (list +not+ lamda) (first cnf) :test #'equal)NIL)
      (cons (first cnf) (extract-no-negative-clauses lamda (rest cnf)))
      (extract-no-negative-clauses lamda (rest cnf)))))

;;
;;  EJEMPLOS:
;;
(extract-neutral-clauses 'p
   '((p (~ q) r) (p q) (r (~ s) q) (a b p) (a (~ p) c) ((~ r) s)))
;; ((R (~ S) Q) ((~ R) S))


(extract-neutral-clauses 'r NIL)
;; NIL

(extract-neutral-clauses 'r '(NIL))
;; (NIL)

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

(extract-neutral-clauses 'p
   '((p (~ q) r) (p q) (r (~ s) p q) (a b p) (a (~ p) c) ((~ r) p s)))
;; NIL
```

Comentarios

La función quedó muy compleja porque durante la realización del ejercicio se pensó más en hacerlo más subdividido y modular que eficiente o corto. Pero realmente las funciones auxiliares son prácticamente idénticas y la principal solo interseca a las anteriores.

4.4.2 Extract-Positive-Clauses

Pseudocódigo

Entrada:
lamda: literal positivo
cnf: frase bien formada en forma normal conjuntiva simplificada
Salida:
Subconjunto de claúsulas de cnf que contienen el literal positivo lamda

Procesamiento:
Si es nulo o lamda o cnf devuelve nil
Si no, revisa si cada claúsula interna posee el literal positivo, si lo tiene la devuelve juntando la claúsula con el resto de claúsulas que lo posean (para esto último se usa recursividad).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.2
;; Construye el conjunto de clausulas lambda-positivas para una FNC
;;
;; RECIBE    : cnf    - FBF en FNC simplificada
;;             lambda - literal positivo
;; EVALUA A : cnf_lambda^(+) subconjunto de clausulas de cnf
;;             que contienen el literal lambda
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun extract-positive-clauses (lamda cnf)
  (if (or (null cnf) (literal-p cnf))
    NIL
    (if (equal (member lamda (first cnf) :test #'equal) NIL)
      (extract-positive-clauses lamda (rest cnf))
      (cons (first cnf) (extract-positive-clauses lamda (rest cnf))))))

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


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

Comentarios

Idéntica a la auxiliar de extract-no-positive-clauses solo cambia el orden de las respuestas del if.

4.4.3 Extract-Negative-Clauses

Pseudocódigo

Entrada:
lamda: literal positivo
cnf: frase bien formada en forma normal conjuntiva simplificada
Salida:
Subconjunto de claúsulas de cnf que contienen el literal de lamda negado
Procesamiento:
Si es nulo o lamda o cnf devuelve nil
Si no, revisa si cada claúsula interna posee el literal negativo, si lo tiene la devuelve juntando la claúsula con el resto de claúsulas que lo posean (para esto último se usa recursividad).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.3
;; Construye el conjunto de clausulas lambda-negativas para una FNC
;;
;; RECIBE   : cnf    - FBF en FNC simplificada
;;            lambda - literal positivo
;; EVALUA A : cnf_lambda^(-) subconjunto de clausulas de cnf
;;            que contienen el literal lambda
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun extract-negative-clauses (lamda cnf)
  (if (or (null cnf) (literal-p cnf))
    NIL
    (if (equal (member (list +not+ lamda) (first cnf) :test #'equal)NIL)
      (extract-negative-clauses lamda (rest cnf))
      (cons (first cnf) (extract-negative-clauses lamda (rest cnf))))))

;;
;;  EJEMPLOS:
;;
(extract-negative-clauses 'p
    '((p (~ q) r) (p q) (r (~ s) q) (a b p) (a (~ p) c) ((~ r) s)))
;; ((A (~ P) C))

(extract-negative-clauses 'r NIL)
;; NIL
(extract-negative-clauses 'r '(NIL))
;; NIL
(extract-negative-clauses 'r
    '((p (~ q) r) (p q) (r (~ s) q) (a b p) (a (~ p) c) ((~ r) s)))
;; (((~ R) S))
(extract-negative-clauses 'p
    '(( p (~ q) r) ( p q) (r (~ s) p q) (a b p) ((~ r) p s)))
;; NIL
```

Comentarios

Idéntica a la auxiliar de extract-no-positive-clauses solo cambia el orden de las respuestas del if.

4.4.4 Resolve-On

Pseudocódigo

Entrada:
lamda: literal positivo
K1: primera de las dos claúsulas simplficadas
K2: segunda de las dos claúsulas simplficadas
Salida:
lista con la claúsula resultante de aplicar resolución sobre K1 y K2 con literales repetidos eliminados.
Procesamiento:
Si alguna de las claúsulas es nil entonces devuelve nil
Si en una cláusula se encuentra el lamda y en la otra el no lamda, crea una lista con ambas claúsulas eliminando literales repetidos y los propios lamdas y no lamdas

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.4
;; resolvente de dos clausulas
;;
;; RECIBE    : lamda     - literal positivo
;;             K1, K2    - clausulas simplificadas
;; EVALUA A  : res_lamda(K1,K2)
;;                       - lista que contiene la
;;                         clausula que resulta de aplicar resolucion
;;                         sobre K1 y K2, con los literales repetidos
;;                         eliminados
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun resolve-on (lamda K1 K2)
  (if (or (equal K1 nil) (equal K2 nil))
    NIL
    (cond
      ((and (not (equal (member lamda K1 :test #'equal) nil))
        (not (equal (member (list +not+ lamda) K2 :test #'equal) nil)))
          (list(eliminate-repeated-literals
                (union (remove lamda K1 :test #'equal)
            (remove (list +not+ lamda) K2 :test #'equal)))))
      ((and (not(equal (member (list +not+ lamda) K1 :test #'equal)nil))
        (not (equal (member lamda K2 :test #'equal) nil)))
          (list(eliminate-repeated-literals
                (union (remove (list +not+ lamda) K1 :test #'equal)
            (remove lamda K2 :test #'equal)))))
```

```
        (t   nil))))
;;
;;   EJEMPLOS:
;;
(resolve-on 'p '(a b (~ c) p) '((~ p) b a q r s))
;; (((~ C) B A Q R S))

(resolve-on 'p '(a b (~ c) (~ p)) '( p b a q r s))
;; (((~ C) B A Q R S))

(resolve-on 'p '(p) '((~ p)))
;; (NIL)


(resolve-on 'p NIL '(p b a q r s))
;; NIL

(resolve-on 'p NIL NIL)
;; NIL

(resolve-on 'p '(a b (~ c) (~ p)) '(p b a q r s))
;; (((~ C) B A Q R S))

(resolve-on 'p '(a b (~ c)) '(p b a q r s))
;; NIL
```

Comentarios

Esta función no estaba explicada en ningún sitio así que supusimos que esta era la idea.
Se basa en buscar lamda y no lamda en cada una de las claúsulas gracias a member.
Segunda condición es idéntica a la primera cambiando donde se encuentra el lamda positivo y donde el negativo.

4.4.5 Build-RES

Pseudocódigo

Entrada:
lamda: literal positivo
cnf: fbf en fnc simplificada
Salida:
Aplica resolución y construye su conjunto de claúsulas
Procesamiento:
La función principal simplemente aplica la propia definición de resolución

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

```
;; EJERCICIO 4.4.5
;; Construye el conjunto de clausulas RES para una FNC
;;
;; RECIBE   : lamda - literal positivo
;;            cnf   - FBF en FNC simplificada
;;
;; EVALUA A : RES_lambda(cnf) con las clauses repetidas eliminadas
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun aux (lamda clause cnf)
  (mapcan #'(lambda(x) (if (not(equal x lamda))
        (resolve-on lamda clause x) nil)) cnf))

(defun build-RES (lamda cnf)
 (eliminate-repeated-literals(eliminate-repeated-clauses
  (union
    (extract-neutral-clauses lamda cnf)
    (mapcan #'(lambda (x) (aux lamda x cnf))
        (eliminate-repeated-clauses cnf))))))
```

Comentarios

La función auxiliar se encarga de resolver para una clausula
Función principal aplica la definición de resolución dada en el enunciado

4.5 Algoritmo para determinar si una FNC es SAT

Pseudocódigo

Entrada:
cnf: frase bien formada en forma normal conjuntiva simplificada
Salida:
T si es satisfacible, Nil si no lo es
Procesamiento:
Si cnf es null entonces es unsat
si cnf es la lista con nil dentro también es unsat
si tiene algún miembro que es nil también es unsat.
Si no entra en ninguna de las condiciones de parada anteriores llama a la recursiva, aplicando el algoritmo dado

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.5
;; Comprueba si una FNC es SAT calculando RES para todos los
;; atomos en la FNC
;;
;; RECIBE   : cnf - FBF en FNC simplificada
```

```
;; EVALUA A : T  si cnf es SAT
;;                NIL  si cnf es UNSAT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun  RES-SAT-p (cnf)
  (cond
    ((null cnf) t)
    ((equal '(nil) cnf) nil)
    ((member nil cnf) nil)
    (t (RES-SAT-p-rec (extract-positive-literals cnf) cnf))
    )
  )

(defun RES-SAT-p-rec (lamdas cnf)
  (cond
   ((or (null cnf) (null lamdas)) t)
    (t
      (let* ((newlamda (first lamdas))
             (newalpha (simplify-cnf (build-RES newlamda cnf))))
        (if (some #'null newalpha)
            nil
          (RES-SAT-p-rec (rest lamdas) newalpha))))))

(defun extract-positive-literals (cnf)
  (if (null cnf)
    NIL
    (eliminate-repeated-literals (remove nil (mapcan #'(lambda (x)
      (mapcar #'(lambda (y)
        (if (equal (positive-literal-p y) t)
            y nil)) x))cnf)))))
;;
;;  EJEMPLOS:
;;
;;
;; SAT Examples
;;
(RES-SAT-p nil)  ;;; T
(RES-SAT-p '((p) ((~ q))))  ;;; T
(RES-SAT-p
 '((a b d) ((~ p) q) ((~ c) a b) ((~ b) (~ p) d) (c d (~ a))))  ;;; T
(RES-SAT-p
 '(((~ p) (~ q) (~ r)) (q r) ((~ q) p) ((~ q)) ((~ p) (~ q) r)))  ;;;T
(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
(RES-SAT-p '(((~ q)) (q) ((~ A))))
```

Comentarios

La definición de una función auxiliar recursiva es necesaria debido a que el algoritmo usa como resultado parcial alfa que además es una variable.
La función auxiliar de extraer literales positivos de una cnf devuelve una lista con los mismos y es necesaria para aplicar bien el algoritmo dado.

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

Pseudocódigo

Entrada:
wff: fbf en formato infijo
w: literal
Salida:
T si es consecuencia lógica, Nil si no lo es
Procesamiento:
La función llama a la del 4.5 con la unión entre la wff y la negación de w tal y como explica el algoritmo

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.6:
;; Resolucion basada en RES-SAT-p
;;
;; RECIBE   : wff - FBF en formato infijo
;;            w   - FBF en formato infijo
;;
;; EVALUA A : T   si w es consecuencia logica de wff
;;            NIL en caso de que no sea consecuencia logica.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; funcion auxiliar que devuelve not w
(defun notw (w)
  (if (negative-literal-p w)          ;; si es negativo w
    (list(list(second w)))            ;; devuelvo w
    (list(list(list +not+ w)))))      ;; si no, not w
```

```
(notw '(~ a))

;; Funcion principal que aplica el algoritmo
(defun logical-consequence-RES-SAT-p (wff w)
  (not (res-sat-p (union
                   (simplify-cnf (wff-infix-to-cnf wff))
                   (notw w)))))      ;; metemos w negado y hacemos resolucion con w

;;
;;  EJEMPLOS:
;;
(wff-infix-to-cnf '(q))


(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 '((p => (~ p)) ^ p) 'q)
;; T

(logical-consequence-RES-SAT-p '((p => (~ p)) ^ p) '(~ q))
;; T

(logical-consequence-RES-SAT-p '((p => q) ^ p) 'q)
;; T

(logical-consequence-RES-SAT-p '((p => q) ^ p) '(~ q))
;; NIL

(logical-consequence-RES-SAT-p
 '(((~ p) => q) ^ (p => (a v (~ b))) e^ (p => ((~ a) ^ b)) ^ ( (~ p) => (r
^ (~ q))))
 '(~ a))
;; T

(logical-consequence-RES-SAT-p
 '(((~ p) => q) ^ (p => (a v (~ b))) ^ (p => ((~ a) ^ b)) ^ ( (~ p) => (r
^ (~ q))))
 'a)
;; T

(logical-consequence-RES-SAT-p
 '(((~ p) => q) ^ (p => ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
```

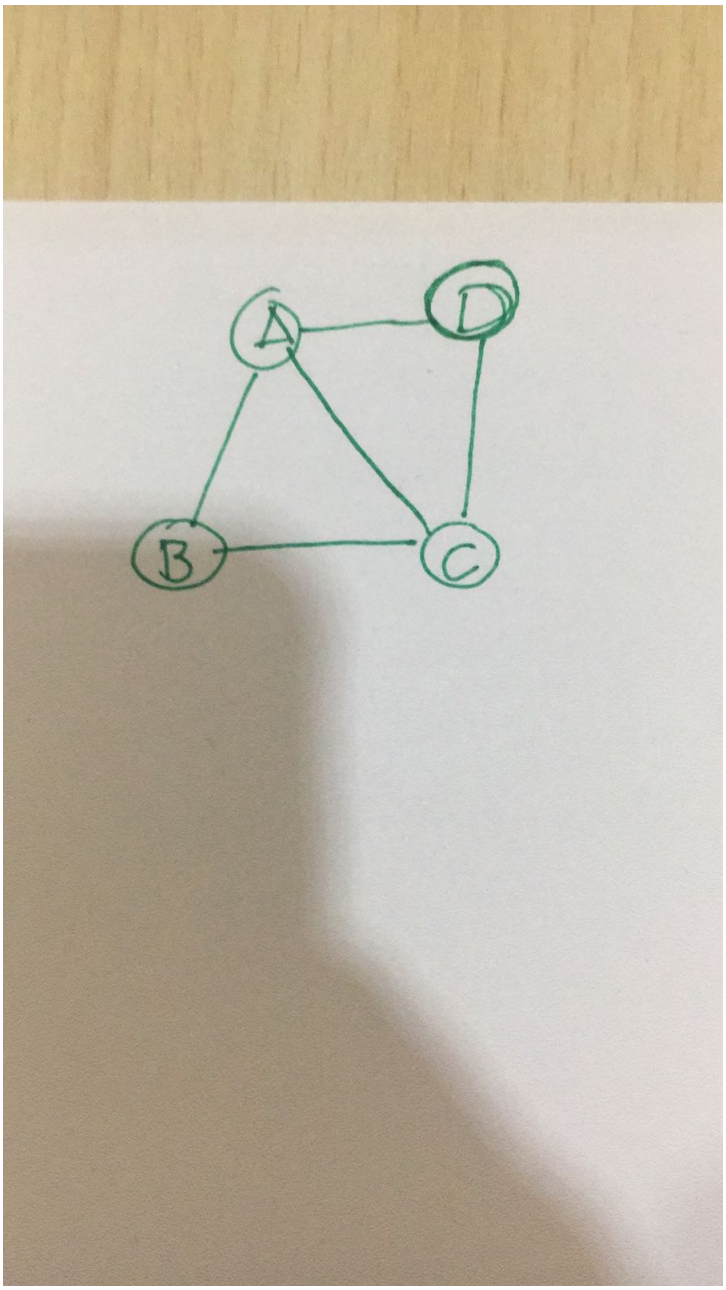
```
 'a)
;; NIL

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

(logical-consequence-RES-SAT-p
 '(((~ p) => q) ^ (p <=> ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
 'q)
;; NIL

(logical-consequence-RES-SAT-p
 '(((~ p) => q) ^ (p <=> ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
 '(~ q))
;; NIL

(or
 (logical-consequence-RES-SAT-p '((p => q) ^ p) '(~q))        ;; NIL
 (logical-consequence-RES-SAT-p
  '(((~ p) => q) ^ (p => ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
  'a) ;; NIL
 (logical-consequence-RES-SAT-p
  '(((~ p) => q) ^ (p <=> ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
  'q) ;; NIL
 (logical-consequence-RES-SAT-p
  '(((~ p) => q) ^ (p <=> ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q))))
  '(~ q)))
```

Comentarios

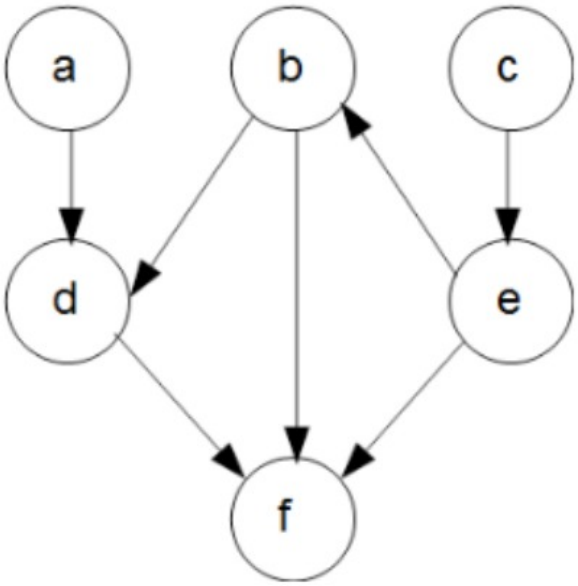
Usamos una función auxiliar notw para convertir de w a su negado de tal forma que después su unión con wff sea satisfactoria.

5. Búsqueda en anchura

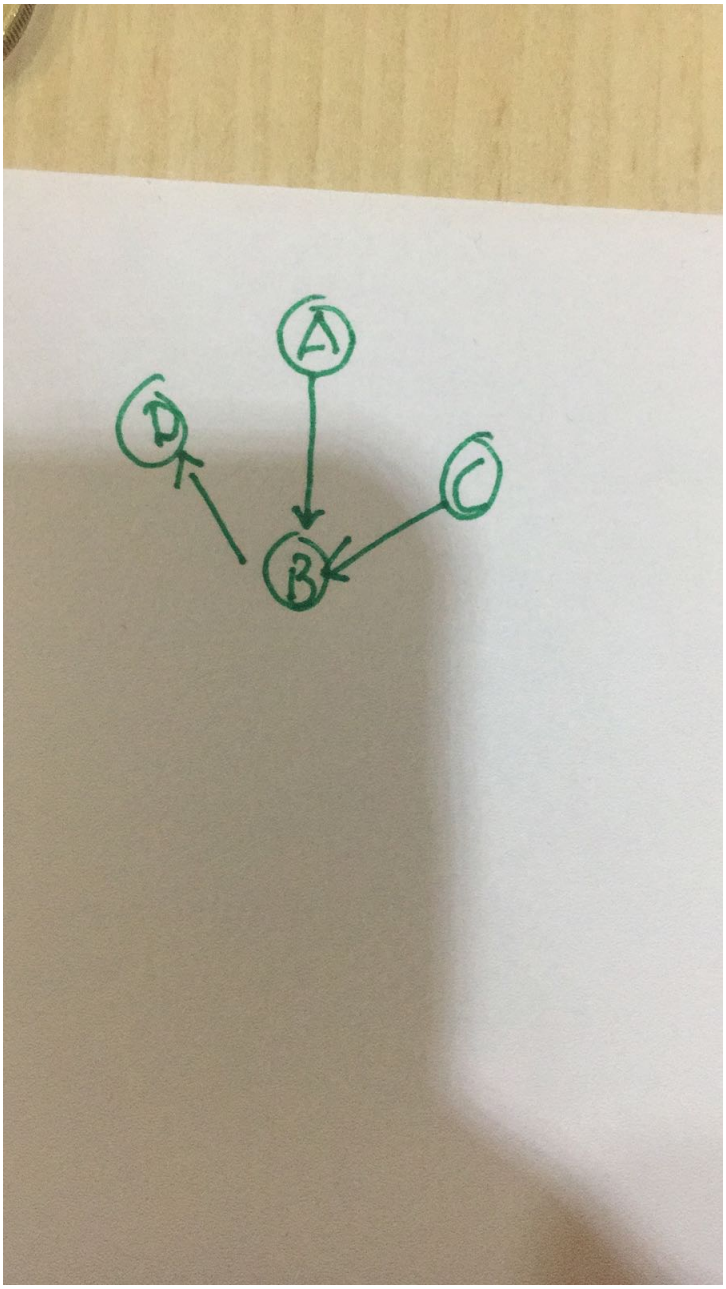
5.1 Resolviendo algoritmos a mano



Supongamos que queremos ir desde el nodo D al nodo B. Como D es distinto de B, lo exploramos y metemos sus sucesores en la lista de nodos a explorar (en orden alfabético). A continuación, sacamos de la lista el nodo A y como es distinto de B exploramos y metemos en la lista de nodos a explorar B y C. Sacamos C, que es distinto de B y metemos sus sucesores A, B y D. Sacamos de la lista B que es la meta, por lo que el algoritmo para. El camino encontrado es D-A-B.



Supongamos que queremos ir del nodo b al f. Primero comprobamos b. Como b no es la meta exploramos b y metemos sus sucesores en la lista de nodos a explorar. Por tanto ahora, tenemos en la lista d y e. Exploramos d (por orden alfabético) y como no es la meta introducimos en la lista el nodo f. Como f es la meta, el algoritmo para, construyendo el camino a-b-e.



Suponemos que vamos desde el A al D. El algoritmo empieza comparando el A con la meta. Como es distinto mete sus sucesores en la lista de nodos a explorar que es B. A continuación sacamos B de la lista y metemos sus sucesores que es tan solo B. Por último, saco B de la lista y como es la meta, el algoritmo termina.

5.2 Pseudocódigo

```
1  metodo BFS(Grafo, origen):
2      creamos una cola Q
```

```
      agregamos origen a la cola Q
      marcamos origen como visitado
      mientras Q no este vacio:
          sacamos un elemento de la cola Q llamado v
          para cada vertice w adyacente a v en el Grafo:
              si w no ah sido visitado:
                  marcamos como visitado w
                  insertamos w dentro de la cola Q
```

5.3 Algoritmo

Ver comentarios en el código.

5.4 Comentarios

Ver comentarios en el código.

5.6 camino más corto del grafo del ejemplo

La solución de (shortest-path 'a 'f '((a d) (b d f) (c e) (d f) (e b f) (f))) es a-d-f

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  0[5]: (SHORTEST-PATH A F ((A D) (B D F) (C E) (D F) (E B F) (F)))
;;    1[5]: (BFS F ((A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
;;      2[5]: (BFS F ((D A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
;;        3[5]: (BFS F ((F D A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
;;        3[5]: returned (A D F)
;;      2[5]: returned (A D F)
;;    1[5]: returned (A D F)
;;  0[5]: returned (A D F)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

En la llamada a trace se ve como primero saca A y lo explora, luego D (sucesor de A) y por último la meta F.

5.7 Camino más corto del grafo del ejemplo

El algoritmo resuelto a mano nos da el camino F-B-A-C o F-H-G-C, depende de como el algoritmo elija que nodo meter antes en la cola.
El evaluamos el la sentencia (shortest-path 'f 'c '((a c d e) (b d e f) (c a g) (d a b h g) (e a b h g) (f b h) (g c d e h) (h d e f g))) nos da el resultado f-b-a-c que concuerda con lo esperado.

5.8 Algoritmo alternativo

Si introducimos bucles en un grafo: (shortest-path 'a 'd '((a b) (b c) (c a) (d)))
No tiene solución, ya que hay un recursión infinita por culpa de un lazo
Para solucionarlo, tan solo habrá que añadir al algoritmo del primer apartado la eliminación de

estados repetidos. Para ello, lo que he hecho es definir una nueva función para crear los nuevos caminos a partir de un nodo. Esta función comprueba que si el nuevo nodo a introducir en la lista de nodos a visitar, ya está en el path (camino más corto) no lo introducimos en la lista para que así no haya bucles infinitos.
El desarrollo del algoritmo se especifica en el código adjunto.

Conclusiones

En esta práctica hemos aprendido que si queremos hacer algo intuitivo y rápido no elejimos un lenguaje de programación funcional, ya que a la hora de implementar funciones es muy lioso. Sin embargo, si buscamos eficiencia (obviando que lisp es interpretado) conseguimos resultados bastante buenos. En conclusión hemos realizado una inteligencia capaz de resolver problemas de lógica preposicional muy potente. Por otro lado, la práctica nos ha permitido manejar muchísimo la notación prefijo que no es la que manejamos habitualmente. También, la subdivisión del ejercicio 4, permite que el código se haya modularizado mucho llegando a conseguir que incluso podamos a ejecutar funciones que de primeras no seríamos capaces de empezar a hacer.