Inteligencia Artificial
Práctica 1

Martín Selgas, Blanca (blanca.martins@estudiante.uam.es)
Villar Gómez, Fernando (fernando.villarg@estudiante.uam.es)
Grupo 2302 - Pareja 3

06/03/2018

Índice

1. Introducción

En el presente documento se detalla la realización de los problemas pertenecientes a la primera práctica de Inteligencia Artificial. En ellos se requiere la resolución de determinadas tareas en el lenguaje funciónal LISP, aportando el pseudo-código, la codificación, la documentación y los comentarios de las funciónes utilizadas para tales fines.

El desarrollo de la práctica se ha realizado en la aplicación **Allegro CL 6.2 ANSI** corriendo sobre un sistema Windows 10, pero el código utilizado es compatible con otros entornos LISP como **SBCL**.

2. Ejercicio 1

2.1. Apartado 1.1

Para calcular la similitud coseno entre dos vectores necesitamos realizar varias operaciones, que se pueden resumir en tres:

- **Producto escalar**, en concreto tres de ellos: $x \cdot y$, $x \cdot x$ y $y \cdot y$.
- **Raíz cuadrada**, función nativa en LISP: `sqrt`
- **División**, funcón nativa en LISP: `/`

Por tanto es necesario codificar el producto escalar. Tal y como se dice en el enunciado hay dos formas diferentes de implementarlo: a través de recursión o a través de una función `mapcar`.

Producto escalar usando recursión

PSEUDOCÓDIGO

```
Entrada: x (vector)
         y (vector)
Salida:  x·y (producto escalar)

Procesamiento:
         Si x o y están vacíos,
             evalúa a 0
         en caso contrario,
             evalúa a x[0]·y[0] + producto_escalar(x[1:n],y[1:n])
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; prod-esc-rec (x y)
;;; Calcula el producto escalar de dos vectores de forma recursiva
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: producto escalar entre x e y
;;;
(defun prod-esc-rec (x y)
  (if (or (null x) (null y))
      0
    (+ (* (first x) (first y))
       (prod-esc-rec (rest x) (rest y)))))
;;;
;;; EJEMPLOS:
;;; (prod-esc-rec '() '()) ;-> 0          ; caso base
;;; (prod-esc-rec '(1 2) '(3 4)) ;-> 11   ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Producto escalar usando mapcar

PSEUDOCÓDIGO

Entrada: x (vector)
y (vector)
Salida: $x \cdot y$ (producto escalar)

Procesamiento:
inicializa ret=0
Para i desde 1 a n,
ret <= ret + x[i]·y[i]
evalúa a ret

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; prod-esc-mapcar (x y)
;;; Calcula el producto escalar de dos vectores usando mapcar
;;;
;;; INPUT: x: vector, representado como una lista
;;; y: vector, representado como una lista
;;;
;;; OUTPUT: producto escalar entre x e y
;;;
(defun prod-esc-mapcar (x y)
  (apply #'+ (mapcar #'* x y)))
;;;
;;; EJEMPLOS:
;;; (prod-esc-mapcar '() '()) ;-> 0          ; caso base
;;; (prod-esc-mapcar '(1 2) '(3 1)) ;-> 5    ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Además, entre los requisitos para el correcto funciónamiento del programa para calcular la similitud coseno se encuentra la condición de que se retorne `NIL` en caso de que se introduzcan dos vectores cero como argumentos. Por tanto necesitamos crear otra función que realice esta tarea:

Comprobación de vectores cero

PSEUDOCÓDIGO

Entrada: x (vector)
Salida: t si x es vector cero o vacío, nil en caso contrario

Procesamiento:
Si x está vacío,
evalúa a t
en caso contrario,
evalúa a x[0]==0 and comprobar_cero(x[1:n])

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; check-zero (x)
;;; Comprueba si x es el vector cero
;;;
;;; INPUT: x: vector, representado como una lista
;;;
;;; OUTPUT: t si x es el vector cero o nil, nil en caso contrario
;;;
(defun check-zero (x)
  (if (null x)
      t
    (and (= (first x) 0)
         (check-zero (rest x)))))
;;;
;;; EJEMPLOS:
;;; (check-zero '()) ;-> t          ; caso base
```

```
;;; (check-zero '(0 0 0)) ;-> t     ; caso tipico
;;; (check-zero '(1 0 0)) ;-> nil   ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Una vez tenemos estas funciónes podemos codificar la función principal para el cálculo de la similitud coseno, con comprobación de errores incluida.

Cálculo de similitud coseno

PSEUDOCÓDIGO

Entrada: x (vector)
y (vector)

Salida: nil si se introducen argumentos no permitidos,
$cos\_similarity(x,y) = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$ en caso contrario

Procesamiento:
Si x o y están vacíos o son cero,
evalúa a nil
en caso contrario,
evalúa a prod_esc(x,y) entre raíz(prod_esc(x,x))*raíz(prod_esc(y,y))

CÓDIGO 1 (recursión)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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 sc-rec (x y)
  (unless (or (check-zero x) (check-zero y))
    (/ (prod-esc-rec x y)
       (* (sqrt (prod-esc-rec x x))
          (sqrt (prod-esc-rec y y))))))
;;;
;;; EJEMPLOS:
;;; (sc-rec '() '()) ;-> nil                ; caso no permitido
;;; (sc-rec '(0 0) '(0 0)) ;-> nil          ; caso no permitido
;;; (sc-rec '(1 2) '(2 3)) ;-> 0.99227786   ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

CÓDIGO 2 (mapcar)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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)
  (unless (or (check-zero x) (check-zero y))
    (/ (prod-esc-mapcar x y)
       (* (sqrt (prod-esc-mapcar x x))
          (sqrt (prod-esc-mapcar y y))))))
;;;
;;; EJEMPLOS:
;;; (sc-mapcar '() '()) ;-> nil          ; caso no permitido
;;; (sc-mapcar '(0 0) '(0 0)) ;-> nil    ; caso no permitido
;;; (sc-mapcar '(1 2) '(2 4)) ;-> 1.0    ; caso tipico
```

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

COMENTARIOS:

- En la comprobación de la función `check-zero` ya se comprueba si se ha introducido un vector vacío (y devuelve `t` en tal caso), por lo que no es necesario incluir las cláusulas `(null x)` o `(null y)`.
- No hay ahorro evidente entre ambas funciónes por el uso de recursión o iteratividad, ya que en cualquiera de los casos se tienen que procesar todas las componentes de los vectores sin excepción.

2.2. Apartado 1.2

Para la implementación de la función `sc-conf` requerida en este apartado se han codificado dos funciones auxiliares, para gestionar diccionarios (con pares vector-similitud) en los que se puedan ordenar los vectores según su similitud.

Creación del diccionario

PSEUDOCÓDIGO

```
Entrada: x (vector)
         vs (lista de vectores)
Salida:  diccionario con pares (vector, similitud)

Procesamiento:
         Para cada elemento de vs,
             calcula cos_similarity(x,elemento)
             añade (cos_similarity(x,elemento), elemento) a lst
         evalúa a lst
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf-gendict (x vs)
;;; Genera un diccionario con la similitud coseno entre un vector
;;; y todos los vectores de una lista de vectores
;;;
;;; INPUT: x: vector, representado como una lista
;;; vs: vector de vectores, representado como una lista de listas
;;;
;;; OUTPUT: diccionario formado por las similitudes y los vectores
;;;
(defun sc-conf-gendict (x vs)
  (mapcar #'(lambda (y) (list (sc-rec x y) y)) vs))
;;;
;;; EJEMPLOS:
;;; (sc-conf-gendict '(1 2) '((1 2) (1 3)))
;;; -> ((1.0 (1 2)) (0.98994946 (1 3)))
;;; (sc-conf-gendict '(2 3) '((1 1) (1 2)))
;;; -> ((0.9805807 (1 1)) (0.99227786 (1 2)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Posteriormente, se realizarían las modificaciones pertinentes a este diccionario según los requisitos, en este caso habría que eliminar todos los vectores con una similitud inferior al argumento `conf` y posteriormente ordenarlos de mayor a menor (más adelante se mostrará con qué implementación de las funciones `sort` y `remove`). Finalmente, para obtener la salida que buscamos se usa la siguiente función.

Descomposición del diccionario

PSEUDOCÓDIGO

```
Entrada: dic (diccionario)
Salida:  lista de vectores ordenados del diccionario

Procesamiento:
         Para cada elemento de dic,
```

```
            obtener elemento[1], vector correspondiente
            añade elemento[1] a lst
        evalúa a lst
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf-selvec (lst)
;;; De un diccionario obtiene el vector correspondiente a cada similitud
;;;
;;; INPUT: lst: diccionario de similitudes-vectores
;;;
;;; OUTPUT: lista de vectores procedentes del diccionario en el mismo orden
;;;
(defun sc-conf-selvec (lst)
  (mapcar #'(lambda (y) (first (second y))) lst))
;;;
;;; EJEMPLOS:
;;; (sc-conf-selvec '((1.0 (1 2)) (0.98994946 (1 3)))) ;-> ((1 2) (1 3))
;;; (sc-conf-selvec '((0.9805807 (1 1)) (0.99227786 (1 2)))) ;-> ((1 1) (1 2))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Y finalmente la función que implementa todo el proceso:

Obtención de vectores ordenados por similitud mayores a un nivel de confianza

PSEUDOCÓDIGO

```
 Entrada: cat (categoría)
          vs (lista de vectores)
          conf (nivel de confianza)
 Salida:  lista de vectores ordenados con similitud mayor a conf

 Procesamiento:
          Generar diccionario de pares (cos_similarity(cat,x), x) con x de vs
          eliminar pares con cos_similarity(cat,x)<conf
          ordenar diccionario de mayor a menor según cos_similarity(cat,x)
          descomponer diccionario y obtener lst, lista de vectores
          evalúa a lst
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf (cat vs conf)
;;; Devuelve aquellos vectores similares a una categoria
;;;
;;; INPUT: cat: vector que representa a una categoria, representado como una lista
;;; vs: vector de vectores, representado como una lista de listas
;;; conf: Nivel de confianza
;;; OUTPUT: Vectores cuya similitud es superior al nivel de confianza, ordenados
;;;
(defun sc-conf (cat vs conf)
  (sc-conf-selvec
   (sort
    (remove conf (sc-conf-gendict cat vs) :test #'> :key #'first)
    #'> :key #'first)))
;;;
;;; EJEMPLOS:
;;; (sc-conf '(1 2) '((1 2) (1 3) (1 4) (2 30)) 0.95) ;-> ((1 2) (1 3) (1 4))
;;; (sc-conf '(2 5) '((1 5) (4 10)) 0.9) ;-> ((4 10) (1 5))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- `sort` es una función destructiva, pero en este caso no es importante mantener la lista previa (la desordenada) puesto que sólo estamos interesados en la ordenada. Si quisiéramos mantener la lista previa bastaría con aplicar un `copy-list` a la salida de la instrucción `remove`.

- La utilización de las funciones crear/descomponer diccionario viene justificada por la necesidad de ordenar una serie de vectores en base a un parámetro que no se encuentra en los mismos vectores, sino que se calcula a partir de ellos y es un valor externo.
- Otra posible solución sería crear una lista con las similitudes de los vectores paralelamente a la propia lista de vectores, y posteriormente realizar las mismas operaciones de reordenación en la lista de vectores a medida que se va ordenando la lista de similitudes. Se ha considerado más simple y entendible la opción del uso de diccionarios.

2.3. Apartado 1.3

En este apartado nos piden una tarea similar a la del apartado anterior, pero en esta ocasión sólo tenemos que devolver la categoría de mayor similitud por cada vector. Además, aquí también se incluyen identificaciones por cada vector y categoría, y se puede introducir por argumento qué función usar para calcular la similitud coseno; por lo tanto la implementación de los diccionarios usados en el apartado 1.2 ha de cambiar necesariamente. Las nuevas funciones son las siguientes:

Creación del diccionario

PSEUDOCÓDIGO

Entrada: x (vector)
vs (lista de vectores)
$func$ (función para cálculo de cos_similarity)
Salida: diccionario con pares (similitud, identificador)

Procesamiento:
Para cada elemento de vs,
calcula cos_similarity(x[1:n],elemento[1:n])
añade (cos_similarity(x[1:n],elemento[1:n]), elemento[0]) a lst
evalúa a lst

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-class-gendict (x vs func)
;;; Realiza la misma funcion que sc-conf-gendict pero ahora tenemos
;;; en cuentaque los vectores tienen un identificador, y la funcion
;;; para calcular la similitud se pasa como parametro
;;;
;;; INPUT: x: vector, representado como una lista
;;; vs: vector de vectores, representado como una lista de listas
;;; func: funcion con la que calcular la similitud
;;;
;;; OUTPUT: diccionario formado por las similitudes y los
;;; identificadores de los vectores
;;;
(defun sc-class-gendict (x vs func)
  (mapcar #'(lambda (y)
              (list (funcall func (rest x)
                             (rest y))
                    (car y)))
    vs))
;;;
;;; EJEMPLOS:
;;; (sc-class-gendict '(1 1 2) '((1 1 2) (2 1 3)) #'sc-rec)
;;; -> ((1.0 1) (0.98994946 2))
;;; (sc-class-gendict '(1 2 3) '((1 1 1) (2 1 2)) #'sc-mapcar)
;;; -> ((0.9805807 1) (0.99227786 2))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Descomposición del diccionario

PSEUDOCÓDIGO

Entrada: dic (diccionario)

Salida: lista de vectores ordenados del diccionario

Procesamiento:

```
Para cada elemento de dic,
    obtener elemento[0], identificador, y elemento[1], similitud
    añade (elemento[1] . elemento[0]) a lst
evalúa a lst
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-class-selvec (lst)
;;; Realiza la misma funcion que sc-conf-selvec pero ahora en vez
;;; de devolver el vector, devuelve su identificacion junto con la
;;; similitud en un par
;;;
;;; INPUT: lst: diccionario de similitudes-vectores
;;;
;;; OUTPUT: lista de pares con identificadores de vectores y similitudes
;;;
(defun sc-class-selvec (lst)
  (mapcar #'(lambda (y)
              (cons (second y)
                    (first y)))
    lst))
;;;
;;; EJEMPLOS:
;;; (sc-class-selvec '((1.0 1) (0.98994946 2)))
;;; -> ((1 . 1.0) (2 . 0.98994946))
;;; (sc-class-selvec '((0.9805807 1) (0.99227786 2)))
;;; -> ((1 . 0.9805807) (2 . 0.99227786))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Por tanto, nuestro programa final debe realizar lo mismo que el `sc-conf` del apartado anterior, pero sin usar el comando `remove` y seleccionando únicamente el primer elemento de la lista que nos devuelva `sc-class-selvec`, puesto que será el de mayor similitud. Puesto que vamos a tener que usar un `mapcar` para cada texto, nos ayudaremos de una función intermedia que realice la ordenación y gestione los diccionarios.

Obtención de categorías ordenadas por similitud para un texto

PSEUDOCÓDIGO

Entrada: x (categoría)
vs (lista de textos)
$func$ (función con la que obtener cos_similarity)

Salida: pares similitud-identificador ordenados

Procesamiento:

```
Generar diccionario con pares similitud-identificador
ordenar diccionario según similitud para cada identificador
descomponer diccionario y devolver pares (identificador . similitud)
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-conf-alt (x vs func)
;;; Devuelve lista de pares identificador-similitud ordenados por
;;; similitud a uno dado
;;;
;;; INPUT: x: vector, representado como una lista
;;; vs: vector de vectores, representado como una lista de listas
;;; func: funcion para calcular similitud
;;; OUTPUT: Pares identificador-similitud ordenados
;;;
(defun sc-conf-alt (x vs func)
  (sc-class-selvec
```

```
   (sort (sc-class-gendict x vs func)
         #'> :key #'first)))
;;;
;;; EJEMPLOS:
;;; (sc-conf-alt '(2 1 2) '((1 1 2) (2 1 3) (3 1 4) (4 2 30)) #'sc-rec)
;;; -> ((1 . 1.0) (2 . 0.98994946) (3 . 0.97618705) (4 . 0.9221943))
;;; (sc-conf-alt '(3 2 5) '((1 1 5) (2 4 10)) #'sc-mapcar)
;;; -> ((2 . 1.0) (1 . 0.98328197))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Y finalmente nuestra función principal:

Obtención de categoría de máxima similitud para lista de textos

PSEUDOCÓDIGO

Entrada: *cats* (lista de categorías)
texts (lista de textos)
func (función con la que obtener cos_similarity)
Salida: pares identificador-similitud máximos para cada texto

Procesamiento:
Para cada texto en texts,
obtener lista de pares identificador-similitud de cada categoría ordenados
seleccionar pares[0], el máximo
añadir pares[0] a lst
evalúa a lst

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; sc-classifier (cats texts func)
;;; Clasifica a los textos en categorias.
;;;
;;; INPUT:
;;; cats: vector de vectores, representado como una lista de listas
;;; texts: vector de vectores, representado como una lista de listas
;;; func: funcion para evaluar la similitud coseno
;;; OUTPUT: Pares identificador de categoria con resultado de similitud coseno
;;;
(defun sc-classifier (cats texts func)
  (mapcar #'(lambda (y)
              (car (sc-conf-alt y cats func)))
    texts))
;;;
;;; EJEMPLOS:
;;; (sc-classifier '((1 1 2) (2 5 10)) '((1 1 3) (2 1 4) (3 2 5)) #'sc-rec)
;;; -> ((1 . 0.98994946) (2 . 0.9761871) (1 . 0.9965458))
;;; (sc-classifier '((1 43 23 12) (2 33 54 24)) '((1 3 22 134) (2 43 26 58))
;;; #'sc-mapcar) -> ((2 . 0.48981872) (1 . 0.81555086))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Mismo comentario respecto al tratamiento de los diccionarios que en el apartado anterior, pero además en este caso teníamos que incluir la similitud en el resultado final, por lo que hemos considerado esta opción como la más factible.
- Se ha codificado `sc-conf-alt` como si el primer argumento `x` fuera una categoría, pero luego en `sc-classifier` se ha iterado sobre los textos como primer argumento. Esto se ha realizado por comodidad al definir las funciones, y el resultado final no varía dado que $cos\_similarity(x, y) = cos\_similarity(y, x)$ y es indistinto, por tanto, tratar texto como categoría o al revés.

2.4. Apartado 1.4

Método | Dimensión	3	4	5	6
Recursión	32580	17169	38199	36240
Mapcar	39612	26774	48076	50721

La tabla superior expone los ciclos de reloj de CPU que se han utilizado para aplicar la función `sc-classifier` a vectores de dimensión 3, 4, 5 y 6. En general el número de ciclos para los mismos argumentos puede variar por diversos motivos (como fallos de caché), pero se puede apreciar que hay una tendencia a que la versión iterativa de `mapcar` consuma más ciclos de reloj.

Este hecho puede venir debido a que desde un punto de vista de número de operaciones, conviene más realizar llamadas recursivas que la creación y el manejo de estructuras internas que utilice `mapcar` para funcionar como funciona. Además, es más compatible con la naturaleza del lenguaje funcional, por lo que podríamos también pensar que por cómo funciona LISP se favorezcan más las codificaciones iterativas.

3. Ejercicio 2

A lo largo del Ejercicio 2 se emplea la función `clean`, cuya motivación es el filtrado de las listas formadas únicamente por elementos que son NIL.

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; clean (lst)
;;
;; Funcion no destructiva que devuelve una lista vacia si todos
;; los elementos de lst son NIL.
;;
;; INPUT:
;;
;;    lst: lista que se quiere evaluar.
;;
;; OUTPUT:
;;
;;    NIL si todos los elementos de lst son NIL;
;;    lst en caso contrario.
;;
(defun clean (lst)
  (unless (every #'null lst)
    lst))
;;
;; EJEMPLOS:
;;
;; (clean '(NIL NIL NIL)) ;-> NIL
;; (clean '(NIL 1 NIL)) ;-> (NIL 1 NIL)
;; (clean '(1 2 3)) ;-> (1 2 3)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

3.1. Apartado 2.1

Método de la bisección

PSEUDOCÓDIGO

```
Entrada: f (función)
         a (número real)
         b (número real)
         tol (número real)
Salida:  nil si no se han encontrado raíces,
```

```
          la raíz encontrada en caso contrario.

    Procesamiento:
            Si f(a)*f(b)<0
                Si (b-a)<tol
                    evalúa a (a+b)/2
                en caso contrario
                    Si f(a)*f((a+b)/2)>=0
                        evalúa a la bisección entre (a+b)/2 y b con tolerancia tol
                    en caso contrario
                        evalúa a la bisección entre a y (a+b)/2 con tolerancia tol
            en caso contrario
                evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; bisect (f a b tol)
;;
;; Encuentra una raiz de f entre los puntos a y b utilizando biseccion.
;;
;; Si f(a)f(b)>=0, no hay garantia de que exista una raiz en el
;; intervalo y la funcion devuelve NIL.
;;
;; INPUT:
;;
;;    f: funcion real de un solo parametro con valores reales
;;       de la que queremos encontrar la raiz.
;;    a: limite inferior del intervalo en el que queremos buscar la raiz.
;;    b: b>a limite superior del intervalo en el que queremos buscar la raiz.
;;    tol: tolerancia para el criterio de parada: si b-a < tol la funcion
;;         devuelve (a+b)/2 como solucion.
;;
;; OUTPUT:
;;
;;    raiz de la funcion, o NIL si no se ha encontrado ninguna.
;;
(defun bisect (f a b tol)
  (let ((pto-medio (/ (+ a b) 2)))
    (unless (>= (* (funcall f a) (funcall f b)) 0)
      (if (< (- b a) tol)
          pto-medio
        (if (>= (* (funcall f a) (funcall f pto-medio)) 0)
            (bisect f pto-medio b tol)
          (bisect f a pto-medio tol))))))
;;
;; EJEMPLOS:
;;
;; (bisect #'(lambda(x) (sin (* 6.26 x))) 0.0 0.7 0.001) ;-> NIL
;; (bisect #'(lambda(x) (sin (* 6.26 x))) 0.1 0.7 0.001) ;-> 0.5016602
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Se ha empleado la definición recursiva:

```
bisect(f a b tol) = bisect(f (b+a)/2 b tol) [Si f(a)*f((b+a)/2)>=0]
bisect(f a b tol) = bisect(f a (b+a)/2 tol) [Si f((b+a)/2)*f(b)>0]
```

- No se ha escogido la definición del método:

```
While (b-a)<tol
    c = (b-a)/2
```

```
Si f(a)*f(c)>=0
    a = c
en caso contrario
    b = c
```

pues requiere uso de variables y bucles y, por lo tanto, no es funcional. Por ello, se ha optado por una implementación recursiva.

3.2. Apartado 2.2

Función allroot

PSEUDOCÓDIGO

```
Entrada: f (función)
         lst (lista)
         tol (número real)
Salida:  nil si no se han encontrado raíces,
         lista de raíces encontradas en caso contrario.

Procesamiento:
         Si length(lst)!= 1
             evalúa a una lista compuesta por
               bisección entre los dos primeros elementos de la lista
               allroot de la lista sin el primer elemento
         en caso contrario
             evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; allroot (f lst tol)
;;
;; Encuentra todas las raices situadas entre valores consecutivos
;; de una lista ordenada.
;;
;; Siempre que sgn(f(lst[i])) != sgn(f(lst[i+1])) la funcion buscara
;; una raiz en el correspondiente intervalo.
;;
;; INPUT:
;;
;;    f: funcion real de un solo parametro con valores reales
;;       de la que queremos encontrar la raiz.
;;    lst: lista ordenada de valores reales (lst[i] < lst[i+1]).
;;    tol: tolerancia para el criterio de parada: si b-a < tol la funcion
;;         devuelve (a+b)/2 como solucion.
;;
;; OUTPUT:
;;
;;    Una lista de valores reales conteniendo las raices de la
;;    funcion en los sub-intervalos dados.
;;
(defun allroot-aux (f lst tol)
  (unless (and (null (rest lst)) (not (null (first lst))))
    (cons (bisect f (first lst) (second lst) tol)
          (allroot-aux f (rest lst) tol))))

(defun allroot (f lst tol)
  (clean (allroot-aux f lst tol)))
;;
;; EJEMPLOS:
;;
;; (allroot #'(lambda(x) (sin (* 6.28 x))) '(0.1 2.25) 0.0001) ;-> NIL
;; (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)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- `allroot` se basa en la idea de concatenar el resultado del método de la bisección del intervalo definido por los dos primeros números de la lista, y el de repetir el proceso con el resto de la lista. Siguiendo el mismo esquema que en la bisección, por definición el algoritmo *allroot* sería:

```
ret[]
for i := 0 TO length(lst)-1
    ret[i] = bisección entre lst[i],lst[i+1]
```

  pero se ha descartado por el uso de variables y bucles.

- La función auxiliar `clean` permite evitar salidas del tipo `(NIL [..] NIL)` de forma no destructiva. Sin embargo, incluirla en la función recursiva `allroot` supondría posibles comportamientos indeseados (eliminación de un conjunto de NIL cuando más adelante se iba a insertar un número en la lista), por lo que se ha sacrificado la recursión de cola en `allroot`:

```
(defun allroot (f lst tol)
    (unless (and (null (rest lst)) (not (null (first lst))))
        (cons (bisect f (first lst) (second lst) tol)
            (allroot f (rest lst) tol))))
```

3.3. Apartado 2.3

Función allind

PSEUDOCÓDIGO

```
Entrada: f (función)
         a (número real)
         b (número real)
         N (número real)
         tol (número real)
Salida:  nil si no se han encontrado raíces,
        lista de raíces encontradas en caso contrario.

Procesamiento:
         c = a + (b-a)/(2^N)
         Si c <b
             evalúa a una lista compuesta por
               bisección entre a y c
               allind desde c en adelante
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; allind(f a b N tol)
;;;
;;; Divide un intervalo en cierto numero de sub-intervalos y encuentra
;;; todas las raices de la funcion f en los mismos.
;;;
;;; El intervalo [a,b] es dividido en intervalos [x[i],x[i+1]] con
;;; x[i] = a + i*dlt; en cada intervalo se busca una raiz, y todas
;;; las raices encontradas son devueltas en una lista.
;;;
;;; INPUT:
;;;
;;;    f: funcion real de un solo parametro con valores reales
```

```
;;;        de la que queremos encontrar la raiz.
;;;     a: limite inferior del intervalo en el que queremos buscar la raiz.
;;;     b: b>a limite superior del intervalo en el que queremos buscar la raiz.
;;;     N: Exponente del numero de intervalos en el que se divide [a,b]:
;;;        [a,b] se divide en 2^N intervalos
;;;     tol: tolerancia para el criterio de parada: si b-a < tol la funcion
;;;          devuelve (a+b)/2 como solucion.
;;;
;;; OUTPUT:
;;;
;;;     Lista con todas las raices encontradas.
;;;
(defun allind-aux (f x incr tol max)
  (let ((y (+ x incr)))
    (unless (> y max)
      (cons (bisect f x y tol) (allind-aux f y incr tol max)))))

(defun allind (f a b N tol)
  (clean (allind-aux f a (coerce (/ (- b a) (expt 2 N)) 'real) tol b))
;;;
;;; 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);
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- La definición de *allind* corresponde al siguiente pseudocódigo:

```
ret[]
incr = a + (b-a)/(2^N)
for i := 0 TO (2^N)-1
    ret[i] = bisección entre a y (a + incr)
    a = a + incr
```

  Sin embargo, debido al uso de variables y bucles, en nuestro caso se ha realizado una implementación funcional que aprovecha la recursión.

- La función `allind` con recursión de cola realiza el cómputo de la exponencial en cada llamada recursiva, por lo que se ha decidido descartar esta idea, y así aprovechar además la oportunidad para llamar a la función auxiliar `clean` de manera segura.

4. Ejercicio 3

4.1. Apartado 3.1

PSEUDOCÓDIGO

```
Entrada: elt (átomo)
         lst (lista)
Salida:  lista con todas las posibles combinaciones.

Procesamiento:
         Si lst no vacía
             evalúa a una lista compuesta por:
               sublista conteniendo elt y el primer elemento de lst
               resultado de combine-elt-lst(elt , lst sin el primer elemento)
```

```
            en caso contrario
                evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lst (elt lst)
;;; Genera una lista con las combinaciones de un elemento con los
;;; elementos de una lista
;;;
;;; INPUT: elt: elemento a combinar
;;; lst: lista de elementos a combinar
;;;
;;; OUTPUT: Lista con todas las posibles combinaciones
;;;
(defun combine-elt-lst (elt lst)
  (unless (null lst)
    (cons (list elt (first lst))
          (combine-elt-lst elt (rest lst)))))
;;;
;;; EJEMPLOS:
;;; (combine-elt-lst 'a nil) ;-> nil                      ; caso base
;;; (combine-elt-lst nil '(1 2)) ;-> ((NIL 1) (NIL 2))   ; caso atipico
;;; (combine-elt-lst 'a '(1 2)) ;-> ((A 1) (A 2))        ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Por la simpleza de esta función se ha decidido usar la recursión en lugar de un método iterativo. En el fondo el número de operaciones es el mismo puesto que hace falta inevitablemente recorrer toda la lista `lst`, pero se ha elegido este método que es más natural de la programación funcional.
- Importante considerar el caso base en el que devolvemos un `nil` en caso de recibir una lista vacía, dado que es necesario para que posteriormente, en la recursión, la unión de una lista con un `nil` sea la propia lista, y a partir de ahí se empiece a construir.
- Si se recibe un `nil` como primer argumento la función sencillamente tratará a ese `nil` como un elemento cualquiera y realizará la misma acción, como se puede apreciar en el ejemplo.

4.2. Apartado 3.2

PSEUDOCÓDIGO

```
Entrada: lst1 (lista)
         lst2 (lista)
Salida:  lista con todas las posibles combinaciones.

Procesamiento:
         Si lst1 y lst2 distintas de NIL
             evalúa a una lista compuesta por:
               combine-elt-lst(primer elemento de lst1 con lst2)
               combine-lst-lst(lst1 sin el primer elemento con lst2)
         en caso contrario
             evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lst (lst1 lst2)
;;; Genera el producto cartesiano lst1 X lst2 (todas las combinaciones de
;;; elementos entre ambas listas)
;;;
;;; INPUT: lst1: primera lista de elementos a combinar
;;; lst2: segunda lista de elementos a combinar
;;;
```

```
;;; OUTPUT: Lista con todas las posibles combinaciones
;;;
(defun combine-lst-lst (lst1 lst2)
  (unless (or (null lst1) (null lst2))
    (append (combine-elt-lst (first lst1) lst2)
            (combine-lst-lst (rest lst1) lst2))))
;;;
;;; EJEMPLOS:
;;; (combine-lst-lst '() '()) ;-> nil ; caso no permitido
;;; (combine-lst-lst '() '(1 2)) ;-> nil ; caso base
;;; (combine-lst-lst '(1 2) '(a b)) ;-> ((1 A) (1 B) (2 A) (2 B)) ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Al igual que en el apartado anterior y por los mismos motivos se ha elegido una implementación recursiva en lugar de iterativa.
- La comprobación `(null lst2)` es, en realidad, innecesaria, puesto que se realizaría igualmente en la llamada a la función `combine-elt-lst`, por lo que se podría eliminar. Se ha mantenido por motivos de eficiencia, dado que será menos costoso para la máquina realizar una comprobación sobre `lst2` que realizar modificaciones en la pila de funciones con más llamadas.
- El hecho de que siempre se realice la comprobación `(null lst1)` hace que nunca se vaya a llamar a la función `combine-elt-lst` con primer argumento `nil`; por tanto, pierde importancia el tratamiento que se le dio a dicho caso, dado que siempre que no se llame a la función individualmente, nunca va a darse.
- A efectos de codificación, el caso no permitido y el caso base contenidos en los ejemplos no son diferentes; sin embargo, para la naturaleza del algoritmo sí lo son, puesto que pasar dos listas vacías es claramente un caso no permitido pero pasar la primera lista vacía puede ser uno de los pasos de la recursión cuando se haya terminado de llamar con todos los elementos de `lst1` y `(rest lst1)` esté vacía.

4.3. Apartado 3.3

Dado que venimos de implementar dos funciones que combinan un elemento con una lista y una lista con una lista, para realizar la tarea de combinar entre sí todos los elementos de una lista de listas lo lógico es pensar en una recursión que aplique el siguiente pseudocódigo:

```
combinacion_listas(lista-de-listas):
  combinacion_lista_lista(lista-de-listas[0],
                          combinacion_listas(lista-de-listas[1:n]))
```

Sin embargo, con las funciones anteriormente codificadas no se puede hacer esta tarea, dado que tal y como están programadas, en uno de los pasos intermedios de la recursión podríamos encontrarnos con la necesidad de combinar, por ejemplo, la lista `(1 2)` con la lista de listas `((a +) (a -) (b +) (b-))`, cuyo resultado deseado sería `((1 a +) (1 a -) (1 b +) (1 b -) (2 a +)...)`; pero lo que realmente devolvería dicha función en este paso sería `((1 (a +)) (1 (a -)) ...)`. Por tanto surge la necesidad de programar funciones específicas que traten con listas de listas, que explicamos a continuación:

Combinación elemento-lista de listas

PSEUDOCÓDIGO

```
Entrada: elt (átomo)
         lol (lista de listas)
Salida:  lista de combinaciones

Procesamiento:
         Para cada lista de lol,
             añadir elt al principio de lista
             añadir lista a ret
         evalúa a ret
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-elt-lol (elt lol)
;;; Genera una lista con las combinaciones de un elemento con los
;;; elementos de una lista de listas
;;;
;;; INPUT: elt: elemento a combinar
;;; lol: lista de listas de elementos a combinar
;;;
;;; OUTPUT: Lista con todas las posibles combinaciones
;;;
(defun combine-elt-lol (elt lol)
  (mapcar #'(lambda (y) (cons elt y)) lol))
;;;
;;; EJEMPLOS:
;;; (combine-elt-lol 'a nil) ;-> nil ; caso no permitido
;;; (combine-elt-lol 'a '((1 2) (3 4))) ;-> ((A 1 2) (A 3 4)) ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- A diferencia de las funciones implementadas en los apartados anteriores, se elige aquí una implementación iterativa a través de `mapcar` en lugar de utilizar la recursión, que daría exactamente el mismo resultado con una codificación siguiendo el siguiente pseudocódigo:

```
combine-elt-lol(elt lol):
    si lol está vacío,
      devolver lista vacía
    en caso contrario,
      añadir elt al principio de lol[0],
      devolver unión de lol[0] y combine-elt-lol(elt lol[1:n])
```

- El caso de pasar `nil` como segundo argumento viene contemplado en los ejemplos. ¿Qué ocurre si usamos `nil` como primer argumento? Lo que ocurre es que se trata a `nil` como un elemento más, y por ejemplo, al combinarlo con `((1 2) (3 4))` surgiría `((NIL 1 2) (NIL 3 4))`.

Combinación lista-lista de listas

PSEUDOCÓDIGO

```
Entrada: lst (lista)
         lol (lista de listas)
Salida:  lista de combinaciones

Procesamiento:
         Para cada elemento de lst,
             combine-elt-lol(elemento lol)
             se añade la combinación a ret
         evalúa a ret
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-lst-lol (lst lol)
;;; Genera el producto cartesiano de una lista con todas las listas
;;; de una lista de listas
;;;
;;; INPUT: lst: lista de elementos a combinar
;;; lol: lista de listas de elementos a combinar
;;;
;;; OUTPUT: Lista con todas las posibles combinaciones
;;;
(defun combine-lst-lol (lst lol)
  (mapcan #'(lambda (z) (combine-elt-lol z lol)) lst))
;;;
```

```
;;; EJEMPLOS:
;;; (combine-lst-lol nil '((1 2) (3 4))) -> nil   ; caso no permitido
;;; (combine-lst-lol '(1 2) nil) -> nil           ; caso no permitido
;;; (combine-lst-lol '(a b) '((1 2) (3 4)))
;;; -> ((A 1 2) (A 3 4) (B 1 2) (B 3 4))          ; caso tipico
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Al igual que en la función anterior se ha elegido el método iterativo para la implementación, aunque existe su versión recursiva, similar a la utilizada en la función del apartado 3.2; sin embargo, dado que no hay diferencia de operaciones porque en cualquier caso hay que iterar sobre todos los elementos de todas las listas, no hay un beneficio muy cuantioso en utilizar la recursión, y sin embargo la versión iterativa es mucho más simple de codificar.
- Importante la utilización de `mapcan` en lugar de `mapcar`. En este caso, usamos `mapcan` para que todas las combinaciones se encuentren en una misma lista. Si usáramos `mapcar`, la salida del último ejemplo expuesto en el código sería `(((A 1 2) (A 3 4)) ((B 1 2) (B 3 4)))`, que no es lo que buscamos.

Con estas dos funciones definidas, ahora sí que podemos plantear la recursión mencionada anteriormente para terminar de construir nuestro combinador de listas:

Combinación de un número arbitrario de listas

PSEUDOCÓDIGO

```
Entrada: lstolsts (lista de listas)
Salida:  lista de combinaciones

Procesamiento:
         Si lstolsts está vacío,
             devuelve ()
         en caso contrario,
             juntar combine-lst-lol (lstolsts[0]) con combine-lstolsts(lstolsts[1:n])
             devolver lista con la unión anterior
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; combine-list-of-lsts (lstolsts)
;;; Genera una lista con todas las disposiciones de elementos
;;; pertenecientes a listas contenidas en una lista de listas
;;;
;;; INPUT:
;;; lstolsts: lista de listas de la que sacar todas las combinaciones
;;;
;;; OUTPUT: Lista con todas las posibles combinaciones
;;;
(defun combine-list-of-lsts (lstolsts)
  (if (null lstolsts)
      '(nil)
    (append (combine-lst-lol (first lstolsts)
                             (combine-list-of-lsts (rest lstolsts))))))
;;;
;;; EJEMPLOS:
;;; (combine-list-of-lsts '()) -> (nil) ; caso base
;;; (combine-list-of-lsts '(() (+ -) (1 2 3 4))) -> nil ; caso no permitido
;;; (combine-list-of-lsts '((a b c) () (1 2 3 4))) -> nil ; caso no permitido
;;; (combine-list-of-lsts '((a b c) (+ -) ())) -> nil ; caso no permitido
;;; (combine-list-of-lsts '((a b c))) -> ((a) (b) (c)) ; caso particular
;;; (combine-list-of-lsts '((a b c) (+ -) (1 2 3 4))) -> ; caso tipico
;;; ((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))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- En esta función es extremadamente importante el caso base de la recursión. Gracias a que cuando se recibe una lista vacía se devuelve `(nil)` conseguimos que todos los pasos se realicen correctamente, ya que `(combine-lst-lol '(1 2 3) '(nil))`, por ejemplo, nos devuelve `((1) (2) (3))`, porque en LISP concatenar un `nil` detrás de un elemento cualquiera en una lista es como no hacer nada. A partir de esta primera lista de listas con un elemento es como se van generando todas las demás en los pasos consecutivos, dando lugar a la salida deseada.
- Cabe destacar también el hecho de que cuando una de las listas de `lstolsts` es vacía, el programa devuelve `nil`. Esto viene como consecuencia de que `combine-lst-lol` devuelve `nil` si `lst` está vacía, que es exactamente lo que ocurre en uno de los pasos de la recursión cuando se llega a la lista vacía dentro de `lstolsts`.

5. Ejercicio 4

5.1. Apartado 4.1

5.1.1. Apartado 4.1.1

Función positive-literal-p

PSEUDOCÓDIGO

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

Procesamiento:
         Si x es NIL, T, un conector, o una lista
             evalúa a NIL
         en caso contrario evalúa a T
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.1
;; positive-literal-p
;;
;; 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)
  (not (or (truth-value-p x)
           (connector-p x)
           (listp x))))
;;
;; EJEMPLOS:
;; (positive-literal-p 'p)
;; evalua a T
;;
;; (positive-literal-p T)
;; (positive-literal-p NIL)
;; (positive-literal-p '~)
;; (positive-literal-p '=>)
;; (positive-literal-p '(p))
;; (positive-literal-p '(~ p))
;; (positive-literal-p '(~ (v p q)))
;;; evaluan a NIL
```

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

COMENTARIOS:

- Esta función utiliza la definición de un literal positivo, es decir: es un átomo (equivalente a decir que no es una lista en LISP), que no es un conector ni un valor de verdad.
- Si aplicamos la Leyes de De Morgan, otra implementación correcta hubiese sido:

```
(and (not (truth-value-p x))
     (not (connector-p x))
     (atom x))
```

5.1.2. Apartado 4.1.2

Función negative-literal-p

PSEUDOCÓDIGO

```
Entrada: x (expresión)
Salida:  T si la expresión es un literal negativo,
         NIL en caso contrario.

Procesamiento:
         Si x es un átomo
             evalúa a NIL
         si x es una lista en la que el primer elemento no es conector unario
             evalúa a NIL
         si x es una lista en la que el segundo elemento no es un literal positivo
             evalúa a NIL
         si x es una lista en la que existe tercer elemento
             evalúa a NIL
         en caso contrario evalúa a T
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.2
;; negative-literal-p
;;
;; 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)
  (unless (or (atom x)
              (not (and (unary-connector-p (first x))
                        (positive-literal-p (second x))
                        (null (third x)))))
    t))
;;
;; 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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.1.3. Apartado 4.1.3

Función literal-p

PSEUDOCÓDIGO

Entrada: x (expresión)
Salida: T si la expresión es un literal,
NIL en caso contrario.

Procesamiento:
Si x es un literal positivo o un literal negativo
evalúa a T
en caso contrario evalúa a NIL

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.3
;; literal-p
;;
;; 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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.1.4. Apartado 4.1.4

Función wff-infix-p

PSEUDOCÓDIGO

Entrada: x (expresión)
Salida: T si la expresión está en formato infijo,
NIL en caso contrario.

Procesamiento:
Si x es NIL
evalúa a NIL.
Si x es un literal
evalúa a T.
Si x es una lista
Si su primer término es un conector unitario
evalúa a T si solo tiene un operador y este está en notación infijo.
Si el segundo término es un conector binario

```
                evalúa a T si solo tiene dos operadores y estos están en notación infijo.
            Si el segundo término es un conector n-ario
                evalúa a T si todos sus operadores están en notación infijo.
            Si el primer término es un conector n-ario
                evalúa a T si no tiene operadores.
            Evalúa a NIL en caso contrario.
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.4
;; wff-infix-p
;;
;; Predicado para determinar si una expresion esta en formato infijo
;;
;; RECIBE   : expresion x
;; EVALUA A : T si x esta en formato infijo,
;;            NIL en caso contrario.
;;
(defun wff-infix-p (x)
  (unless (null x)
    (or (literal-p x)
        (and (listp x)
             (let ((op1 (first x))
                   (op2 (first (rest x)))
                   (op3 (rest (rest x))))
               (cond
                ((unary-connector-p op1)
                 (and (null op3)
                      (wff-infix-p op2)))
                ((binary-connector-p op2)
                 (and (null (rest op3))
                      (wff-infix-p (first x))
                      (wff-infix-p (first op3))))
                ((n-ary-connector-p op2)
                 (let ((tmp (first (rest op3))))
                   (when (or (eql op2 tmp)
                             (null tmp))
                     (and (wff-infix-p op1)
                          (wff-infix-p (first op3))))))
                ((n-ary-connector-p op1)
                 (null (rest x)))
                (t NIL)))))))
;;
;; EJEMPLOS:
;;
;; (wff-infix-p 'a)                                        ; T
;; (wff-infix-p '(^))                                      ; T
;; (wff-infix-p '(v))                                      ; T
;; (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:

- Esta función aprovecha el conocimiento de la estructura de las expresiones lógicas para determinar si están en forma infijo, aunque otra implementación de la función podría ser evalúar si se intercalan literales y conectores, sin hacer distinciones entre estos últimos.
- Una observación es que se ha supuesto que las expresiones a evaluar serán siempre expresiones en forma infijo y prefijo bien formadas, pues ante `(A v B ^ C)` la función arrojaría `T`.

5.1.5. Apartado 4.1.5

Función infix-to-prefix

PSEUDOCÓDIGO

```
Entrada: wff (FBF en formato infijo)
Salida:  FBF en formato prefijo
         NIL si la FBF no estaba en formato infijo o estaba mal formada.

Procesamiento:
         Si wff en formato infijo
             Si wff es un literal
                 evalúa a wff.
             Si wff[0] == conector unario (negación)
                 evalúa a "wff[0] + infix-to-prefix(wff[1])".
             si wff[1] == conector binario
                 evalúa a "wff[1] + infix-to-prefix(wff[0]) + infix-to-prefix(wff[2])".
             Si wff[1] == conector n-ario
                 Si wff+2 == null
                     evalúa a infix-to-prefix(wff[0]).
                 en caso contrario ïnfix-to-prefix(wff[0]) + infix-to-prefix(wff+2)"
             evalúa a NIL en caso contrario.
         evalúa a NIL en caso contrario.
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.5
;; infix-to-prefix
;;
;; 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-eliminate-connectors-aux (op)
  (unless (null op)
    (cons (first op) (infix-to-prefix-eliminate-connectors-aux
                      (rest (rest op))))))

(defun infix-to-prefix (wff)
  (when (wff-infix-p wff)
    (if (literal-p wff)
        wff
```

```
      (let* ((op1 (first wff))
             (op2 (first (rest wff)))
             (op3 (rest (rest wff)))
             (funcion-aux (infix-to-prefix-eliminate-connectors-aux op3)))
        (cond
         ((unary-connector-p op1)
          (list op1
                (infix-to-prefix op2)))
         ((binary-connector-p op2)
          (list op2
                (infix-to-prefix op1)
                (infix-to-prefix (first op3))))
         ((n-ary-connector-p op2)
          (if (null op3)
              (infix-to-prefix op1)
            (cons op2
                  (cons (infix-to-prefix op1) (mapcar #'infix-to-prefix funcion-aux)))))
         (t NIL))))))
;;
;; 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 '(^ (^ (<=> 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))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.1.6. Apartado 4.1.6

Función clause-p

PSEUDOCÓDIGO

Entrada: wff (fórmula bien formada en formato prefijo)
Salida: T si wff es una cláusula,

```
            NIL en caso contrario.

    Procesamiento:
             evalúa a:
                 wff es una lista AND
                 el primer elemento de wff es v (or) AND
                 el resto de elementos de wff son literales
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.6
;; clause-p
;;
;; 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)
  (and (listp wff)
       (eql (first wff) +or+)
       (every #'identity
              (mapcar #'(lambda (lit)
                          (literal-p lit))
                (rest wff)))))
;;
;; 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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.1.7. Apartado 4.1.7

Función cnf-p

PSEUDOCÓDIGO

```
    Entrada: wff (fórmula bien formada en formato prefijo)
    Salida:  T si wff está en FNC con conectores,
             NIL en caso contrario.

    Procesamiento:
             si wff[0] es ^ y wff[1:n] son cláusulas:
                 evalúa a T,
             en caso contrario evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.1.7
;; cnf-p
;;
```

```
;; 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 (literal-p wff)
    (and (equal (first wff) +and+)
         (every #'clause-p (rest wff)))))
;;
;; 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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.2. Apartado 4.2

5.2.1. Apartado 4.2.2

Función eliminate-conditional

PSEUDOCÓDIGO

Entrada: wff (FBF en formato prefijo sin conectores bicondicionales)
Salida: FBF en formato prefijo equivalente sin conectores condicionales.
wff si wff es NIL o un literal.

Procesamiento:

```
si wff es NIL o un literal,
    evalúa a wff
en caso contrario,
    si wff[0] es el conector =>,
        evalúa a (v (¬ wff[1]) wff[2])
    en caso contrario,
        evalúa a (wff[0] eliminate-conditional(wff[1:n]))
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.2
;; eliminate-conditional
;;
;; 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)))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.2.2. Apartado 4.2.3

Función reduce-scope-of-negation

PSEUDOCÓDIGO

Entrada: wff (FBF en formato prefijo sin conectores binarios)
Salida: FBF en formato prefijo equivalente en la que la negación aparece solo en literales. NIL si wff es NIL.

Procesamiento:

```
Si wff != NIL
    Si wff es un literal
        evalúa a wff.
    en caso contrario
        Si wff[0] == not
            Si wff[1] es un literal negativo
                evalúa a wff[1][1]. (= conversión a positivo)
            en caso contrario
                Para cada elemento x de wff[1]
                    Si x es un conector n-ario
                        cambiamos el conector
                    en caso contrario
                        evalúa a reduce-scope-of-negation(negacion de x)
                    Continuamos con el resto de wff
        en caso contrario
            evalúa a wff[0] + reduce-scope-of-negation de todos los demas
en caso contrario
    evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.3
;; reduce-scope-of-negation
;;
```

```
;; 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)
  (unless (null wff)
    (if (literal-p wff)
        wff
      (if (equal +not+ (first wff))
          (if (negative-literal-p (second wff))
              (second (second wff)) ;Caso doble negacion
            (let ((resto (rest wff)))
              (append (mapcar #'(lambda(x) (if (n-ary-connector-p x)
                                               (exchange-and-or x)
                                             (reduce-scope-of-negation
                                              (list +not+ x))))
                              (first resto))
                      (reduce-scope-of-negation (rest resto)))))
        (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))
;; (reduce-scope-of-negation '(^ (v P (v (~ a) (^ b (~ c) d)))
;; (^ (^ (v (~ p) (~ q)) (v (~ (~ q)) p)) p) e))
;; -> (^ (V P (V (~ A) (^ B (~ C) D))) (^ (^ (V (~ P) (~ Q)) (V Q P)) P) E)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Se ha implementado la función no solo para satisfacer los ejemplos dados, sino casos como el nuevo añadido en el que la negación no afecta a toda la expresión.
- Si quisiesemos que nuestra función solo tratase con expresiones que comienzan con ¬ el código se correspondería con:

```
(defun reduce-scope-of-negation (wff)
  (unless (null wff)
    (if (literal-p wff)
      wff
      (if (negative-literal-p (second wff))
        (second (second wff)) ;Caso doble negación
        (let ((resto (rest wff)))
         (append (mapcar #'(lambda(x) (if (n-ary-connector-p x)
                                          (exchange-and-or x)
                                          (reduce-scope-of-negation (list +not+ x))))
                         (first resto))
                 (reduce-scope-of-negation (rest resto))))))))
```

Donde se suprime la búsqueda del caracter que marca la negación de alguna parte de la expresión.

5.2.3. Apartado 4.2.5

Función eliminate-connectors

PSEUDOCÓDIGO

```
Entrada: cnf (FBF en FNC con conectores n-arios)
Salida:  FBF en FNC con conectores eliminados
         NIL si cnf es NIL.

Procesamiento:
         Si cnf != NIL
             Si cnf[0] == AND
                 evalúa a eliminate-connectors de todas las cláusulas de cnf
             Si cnf[0] == OR
                 evalúa a cnf++
             en caso contrario
         evalúa a cnf
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.5:
;; eliminate-connectors
;;
;; Dada una FBF en  FNC
;; evalua a lista de listas sin conectores
;; que representa una conjuncion de disyunciones de literales
;;
;; RECIBE   : FBF en FNC con conectores ^, v
;; EVALUA A : FBF en FNC (con conectores ^, v eliminados)
;;
(defun eliminate-connectors (cnf)
  (unless (null cnf)
    (let ((primero (first cnf))
          (resto   (rest cnf)))
      (cond ((equal primero +and+)
             (mapcar #'eliminate-connectors resto))
            ((equal primero +or+)
             resto)
            (t cnf)))))
;;
;; 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 p (~ q)) (v) (v k r)))
;; -> ((P (~ Q)) NIL (K R))
;; (eliminate-connectors '(^ (v a b))) -> ((A B))
;; (eliminate-connectors '(^ (v p (~ q)) (v k r))) -> ((P (~ Q)) (K R))
;; (eliminate-connectors '(^ (v p (~ q)) (v q (~ a)) (v s e f) (v b)))
;; -> ((P (~ Q)) (Q (~ A)) (S E F) (B))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- La función hace uso de la estructura de una FNC (conjunción de disjunciones de literales) en formato prefijo. De esta manera, si encuentra el conector AND lo eliminará del comienzo de la expresión y continuará con cada una de las cláusulas de la FNC. De la misma manera, eliminará los conectores

OR al comienzo de cada cláusula, devolviendo el resto de la cláusula. Cuando la función detecte que el argumento no comienza por un conector, devolverá el argumento sin modificar.

5.2.4. Apartado 4.2.6

Función wff-infix-to-cnf

PSEUDOCÓDIGO

```
Entrada: wff (FBF en formato infijo)
Salida:  FBF en FNC con conectores eliminados

Procesamiento:
         evalúa a:
         eliminate-connectors(
             cnf(
                 reduce-scope-of-negation(
                     eliminate-conditional(
                         eliminate-biconditional(
                             infix-to-prefix(wff)
                         )
                     )
                 )
             )
         )
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.2.6
;; wff-infix-to-cnf
;;
;; 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)))))))
;;
;; 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))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.3. Apartado 4.3

Para los siguientes apartados se ha desarrollado una pequeña función siguiendo la motivacón de no repetir código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; test-contenido
;;
;; Comprueba si un literal esta contenido en una expresion
;;
;; RECIBE   : x (literal)
;;            y (expresion)
;; EVALUA A : T si x esta contenido en y
;;            NIL en caso contrario
;;
(defun test-contenido (x y)
  (some #'(lambda(z) (equal x z)) y))
;;
;; EJEMPLOS :
;;
;; (test-contenido 'c '(a b c)) -> T
;; (test-contenido 'c '(a b)) -> NIL
;; (test-contenido 'c 'nil) -> NIL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.3.1. Apartado 4.3.1

Función eliminate-repeated-literals

PSEUDOCÓDIGO

Entrada: k (cláusula)
Salida: cláusula equivalente sin literales repetidos.

Procesamiento:

```
Si k != NIL
    Si k[0] es igual a alguno de los siguientes
        evalúa a eliminate-repeated-literals de k++
    en caso contrario
        evalúa a k[0] + eliminate-repeated-literals de k++
en caso contrario
    evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.1
;; eliminate-repeated-literals
;;
;; 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)
  (unless (null k)
    (let* ((primero   (first k))
           (resto     (rest k))
           (eliminar-sig (eliminate-repeated-literals resto)))
      (if (test-contenido primero resto)
          eliminar-sig
        (cons primero
              eliminar-sig)))))
;;
;; EJEMPLO:
;;
;; (eliminate-repeated-literals '(a b (~ c) (~ a) a c (~ c) c a))
;; -> (B (~ A) (~ C) C A)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Se ha escogido una función que devuelve una nueva cláusula, en la que no inserta un literal si más adelante se va a incluir otro repetido. De esta manera, se implementa un algoritmo previsor y funcional que no requiere almacenar conocimiento sobre anteriores ejecuciones del mismo sobre la misma cláusula.

5.3.2. Apartado 4.3.2

Para la implementación de este apartado nos hemos ayudado de una función que mide la igualdad de dos cláusulas

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; test-clauses
;;
;; Comprueba si una clausula esta contenida en otra
;;
;; RECIBE   : k1, k2 - clausulas
;; EVALUA A : T si k1 esta contenida en k2
;;            NIL en caso contrario
;;
(defun test-clauses (k1 k2)
  (if (null k1)
    t
    (let ((primero (first k1)))
      (when (test-contenido primero k2)
        (unless (test-contenido primero
                                (remove primero (copy-list k2)
                                        :test #'equal))
          (and t (test-clauses (rest k1) k2)))))))
;;
;; EJEMPLOS:
;; (test-clauses '(a b) '(a b c d)) -> T
;; (test-clauses '(a b c d) '(a b)) -> NIL
;; (test-clauses 'nil '(a b)) -> T
;; (test-clauses '(a b) 'nil) -> NIL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Función eliminate-repeated-clauses

PSEUDOCÓDIGO

Entrada: cnf (FBF en FNC)
Salida: FBF en FNC equivalente sin cláusulas repetidas.

Procesamiento:

```
Si cnf != NIL
    Sea elt = cnf[0] sin literales repetidos
    Si elt es igual a alguno de los siguientes
        evalúa a eliminate-repeated-clauses de cnf++
    en caso contrario
        evalúa a elt + eliminate-repeated-clauses de cnf++
en caso contrario
    evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.2
;; eliminate-repeated-clauses
;;
;; 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)
  (unless (null cnf)
    (let* ((elt         (eliminate-repeated-literals (first cnf)))
           (resto        (rest cnf))
           (eliminar-sig (eliminate-repeated-clauses resto)))
      (if (some #'(lambda(x) (and (test-clauses elt x)
                                  (test-clauses x elt))) resto)
          eliminar-sig
        (cons elt
              eliminar-sig)))))
;;
;; 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:

- Siguiendo el mismo esquema que en el apartado anterior, esta función no inserta una cláusula en la FNC a no ser que más adelante no vaya ninguna otra igual. Además, según va recorriendo la FNC original, va eliminando los elementos repetidos de cada una de las cláusulas que la forman.

5.3.3. Apartado 4.3.3

Función subsume

PSEUDOCÓDIGO

```
Entrada: k1 (cláusula)
         k2 (cláusula)
Salida:  k1 si k1 subsume a k2
         NIL en caso contrario.

Procesamiento:
         Si k1 está contenido en k2
             evalúa a una lista que contiene a k1
         en caso contrario
             evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.3
;; subsume
;;
;; Predicado que determina si una clausula subsume otra
;;
;; RECIBE   : K1, K2 clausulas
;; EVALUA a : K1 si K1 subsume a K2
;;            NIL en caso contrario
;;
(defun subsume (k1 k2)
  (when (test-clauses k1 k2)
    (list k1)))
;;
;;  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 '(d b (~ e)) '(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

- Puesto que se había desarrollado una función `test-clauses`, que determinaba si los literales de una cláusula estaban contenidos en la otra, se ha aprovechado esta para la implementación de la función `subsume`, con la única diferencia de que `subsume` debía devolver la cláusula que subsumía la otra dentro de una lista.

5.3.4. Apartado 4.3.4

En este ejercicio se ha utilizado la función auxiliar `eliminar-aux`, que elimina las cláusulas subsumidas por otra de una cnf:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; eliminar-aux
;;
;; Elimina las clausulas subsumidas por k de cnf
;;
;; RECIBE   : k   (clausula)
;;            cnf (FBF en FNC)
;; EVALUA A : FBF en FNC equivalente a cnf sin clausulas subsumidas por k
;;
(defun eliminar-aux (k cnf)
  (unless (null cnf)
    (let ((primero       (first cnf))
          (eliminar-sig  (eliminar-aux k (rest cnf))))
      (if (null (subsume k primero))
          (cons primero
                eliminar-sig)
        eliminar-sig))))
;;
;; EJEMPLO:
;; (eliminar-aux '(a b) '((a b c) (b c) (a (~ c) b)
;; ((~ a) b) (a b (~ a)) (c b a))) -> ((B C) ((~ A) B))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Función eliminate-subsumed-clauses

PSEUDOCÓDIGO

```
Entrada: cnf (FNC)
Salida:  FNC equivalente sin cláusulas subsumidas.

Procesamiento:
         Si cnf != NIL
             Si cnf[0] no es subsumida por ninguna otra cláusula de cnf++
                 se eliminan las cláusulas subsumidas por cnf[0] de cnf++
                 evalúa a cnf[0] + eliminate-subsumed-clauses(cnf++)
             en caso contrario
                 evalúa a eliminate-subsumed-clauses(cnf++)
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.4
;; eliminate-subsumed-clauses
;;
;; 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)
  (unless (null cnf)
    (let ((primero  (first cnf))
          (resto    (rest cnf)))
      (if (every #'null (mapcar #'(lambda(x)
                                    (subsume x primero)) resto))
          (cons primero
                (eliminate-subsumed-clauses
                 (eliminar-aux primero
                               resto)))
        (eliminate-subsumed-clauses resto)))))
;;
;;  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))
;; (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))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- En esta función, no se añade una cláusula en la nueva FNC a no ser que no sea subsumida por ninguna otra. Puesto que en este caso la implementación requiere conocimiento sobre iteraciones previas del algoritmo, se ha construido la función `eliminar-aux`, que elimina todas las cláusulas subsumidas por una cláusula de una FNC. Así, podemos asegurar una implementación recursiva y funcional, al eliminar todas las cláusulas subsumidas por una cláusula k al añadirla a la nueva FNC.

5.3.5. Apartado 4.3.5

Función tautology-p

PSEUDOCÓDIGO

```
Entrada: k (cláusula)
Salida:  T si k es tautología
         NIL en caso contrario.

Procesamiento:
         Si k != NIL
             Si (not k[0]) está contenido en k++
                 evalúa a T
             en caso contrario
                 evalúa a tautology-p(k++)
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.5
;; tautology-p
;;
;; Predicado que determina si una clausula es tautologia
;;
;; RECIBE   : K (clausula)
;; EVALUA a : T si K es tautologia
;;            NIL en caso contrario
;;
(defun tautology-p (k)
  (unless (null k)
    (let ((resto (rest k)))
```

```
      (or (test-contenido
           (reduce-scope-of-negation (list +not+ (first k)))
           resto)
          (tautology-p resto)))))
;;
;;  EJEMPLOS:
;;
;; (tautology-p '((~ B) A C (~ A) D)) -> T
;; (tautology-p '((~ B) A C D))       -> NIL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Esta función busca la definición de tautología en una cláusula, es decir, busca la disyunción de un literal y su negación.

5.3.6. Apartado 4.3.6

Función eliminate-tautologies

PSEUDOCÓDIGO

```
Entrada: cnf (FBF en FNC)
Salida:  FNC equivalente a cnf sin tautologías.

Procesamiento:
         Si cnf != NIL
             Si cnf[0] es tautología
                 evalúa a eliminar tautologías de cnf++
             en caso contrario
                 evalúa a cnf[0] + eliminar tautologías de cnf++
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.6
;; eliminate-tautologies
;;
;; 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)
  (unless (null cnf)
    (let ((primero       (first cnf))
          (eliminar-sig  (eliminate-tautologies (rest cnf))))
      (if (tautology-p primero)
          eliminar-sig
        (cons primero
              eliminar-sig)))))
;;
;;  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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- A la hora de eliminar tautologías, es un caso más sencillo, pues simplemente eliminamos la cláusula si nos encontramos con ella, sin necesidad de comprobar nada con el resto de la FBF en FNC.

5.3.7. Apartado 4.3.7

Función simplify-cnf

PSEUDOCÓDIGO

```
Entrada: cnf (FBF en FNC)
Salida:  FNC equivalente a cnf sin literales repetidos,
         cláusulas repetidas/subsumidas ni tautologías.

Procesamiento:
         evalúa a:
             eliminate-subsumed-clauses(
                 eliminate-repeated-clauses(
                     eliminate-tautologies(cnf)
                 )
             )
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.3.7
;; simplify-cnf
;;
;; 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)
  (unless (null cnf)
    (eliminate-subsumed-clauses
     (eliminate-repeated-clauses
      (eliminate-tautologies cnf)))))
;;
;;  EJEMPLOS:
;;
;; (simplify-cnf '((a a) (b) (a) ((~ b)) ((~ b)) (a b c a)
;; (s s d) (b b c a b))) -> ((B) ((~ B)) (S D) (A)) en cualquier orden
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.4. Apartado 4.4

En los apartados 4.4.1, 4.4.2 y 4.4.3 se ha utilizado el mismo prototipo de función, y por lo tanto se ha unificado en uno solo.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; extract-clauses
;;
;; Unificacion de las funciones de extraen las clausulas de una FNC
;;
;; RECIBE   : cnf      - FBF en FNC simplificada
;;            lit      - literal positivo
;;            function - funcion para la condicion de extracion
;; EVALUA A : clausulas de cnf que cumplen la funcion function de lit
;;
(defun extract-clauses (lit cnf function)
  (unless (null cnf)
```

```
    (let ((primero      (first cnf))
          (extraer-sig  (extract-clauses lit (rest cnf) function)))
      (if (funcall function lit primero)
        (cons primero
              extraer-sig)
        extraer-sig))))
;;
;; EJEMPLO:
;; (extract-clauses 'p '((a b) (p b) (a b)) #'test-contenido) -> ((P B))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Las funciones `extract-neutral-clauses`, `extract-negative-clauses` y `extract-positive-clauses`, siguen la misma idea: evaluar si una cláusula cumple una determinada condición, y si es así seleccionarla. Este es el procedimiento que siguen todas ellas que se unifica en esta función.

5.4.1. Apartado 4.4.1

Función extract-neutral-clauses

PSEUDOCÓDIGO

Entrada: *lit* (literal positivo)
cnf (FBF en FNC simplificada)
Salida: conjunto de cláusulas de cnf que no contienen ni lit ni ¬lit

Procesamiento:

```
Si cnf != NIL
    Si cnf[0] contiene lit o ¬lit
        evalúa a extraer las cláusulas neutras de cnf++
    en caso contrario
        evalúa a cnf[0] + extraer las cláusulas neutras de cnf++
```

CÓDIGO

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

;; Funcion auxiliar
(defun aux-neutral (x y)
  (every #'(lambda(z) (not (if (positive-literal-p z)
                               (equal z x)
                             (equal (second z) x))))
         y))

(defun extract-neutral-clauses (lit cnf)
  (extract-clauses lit cnf #'aux-neutral))
;;
;;  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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.4.2. Apartado 4.4.2

Función extract-positive-clauses

PSEUDOCÓDIGO

Entrada: *lit* (literal positivo)
cnf (FBF en FNC simplificada)
Salida: conjunto de cláusulas de cnf que contienen el literal lit

Procesamiento:

```
Si cnf != NIL
    Si cnf[0] contiene lit
        evalúa a cnf[0] + extraer las cláusulas positivas de cnf++
    en caso contrario
        evalúa a extraer las cláusulas positivas de cnf++
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.2
;; extract-positive-clauses
;;
;; Construye el conjunto de clausulas lit-positivas para una FNC
;;
;; RECIBE   : cnf    - FBF en FNC simplificada
;;            lit - literal positivo
;; EVALUA A : cnf_lit^(+) subconjunto de clausulas de cnf
;;            que contienen el literal lit
;;
(defun extract-positive-clauses (lit cnf)
       (extract-clauses lit cnf #'test-contenido))
;;
;;  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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.4.3. Apartado 4.4.3

Función extract-negative-clauses

PSEUDOCÓDIGO

Entrada: *lit* (literal positivo)
cnf (FBF en FNC simplificada)
Salida: conjunto de cláusulas de cnf que contienen el literal $\neg$lit

Procesamiento:

```
            Si cnf != NIL
                Si cnf[0] contiene ¬lit
                    evalúa a cnf[0] + extraer las cláusulas negativas de cnf++
                en caso contrario
                    evalúa a extraer las cláusulas negativas de cnf++
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.3
;; extract-negative-clauses
;;
;; Construye el conjunto de clausulas lit-negativas para una FNC
;;
;; RECIBE   : cnf    - FBF en FNC simplificada
;;            lit - literal positivo
;; EVALUA A : cnf_lit^(-) subconjunto de clausulas de cnf
;;            que contienen el literal ~lit
;;

;; Funcion auxiliar
(defun aux-neg (x y)
  (some #'(lambda(z) (when (negative-literal-p z)
                       (equal (second z) x)))
        y))

(defun extract-negative-clauses (lit cnf)
  (extract-clauses lit cnf #'aux-neg))
;;
;; 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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.4.4. Apartado 4.4.4

Se han utilizado las siguientes funciones auxiliares cuyo objetivo es simplemente facilitar la lectura del código:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; eliminar-rastro
;;
;; Funcion no destructiva que elimina la presencia de un literal
;; en una clausula
;;
;; RECIBE   : lit  - literal positivo
;;            k    - clausula simplificada
;; EVALUA A : clausula los literales lit ~lit eliminados
;;
(defun eliminar-rastro (lit k)
  (remove (reduce-scope-of-negation (list +not+ lit))
          (remove lit (copy-list k) :test #'equal) :test #'equal))
;;
;; EJEMPLO:
;; (auxiliar 'a '(a b c (~ a))) -> (B C)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; possible-resolve-on
;;
;; Determina si es posible aplicar resolucion
;;
;; RECIBE   : lit       - literal positivo
;;            k1, k2    - clausulas simplificadas
;; EVALUA A : T si es posible
;;            NIL en caso contrario
;;
(defun possible-resolve-on (lit k1 k2)
  (if (test-contenido lit k1)
      (test-contenido (list +not+ lit) k2)
    (and (test-contenido (list +not+ lit) k1)
         (test-contenido lit k2))))
;;
;; EJEMPLOS:
;; (possible-resolve-on 'p '(a b (~ c)) '(p b a q r s)) -> NIL
;; (possible-resolve-on 'p '(a b (~ c) p) '((~ p) b a q r s)) -> T
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Función resolve-on

PSEUDOCÓDIGO

Entrada: lit (literal positivo)
$k1$ (cláusula simplificada)
$k2$ (cláusula simplificada)

Salida: lista de cláusulas que resultan de aplicar resolución sobre k1, k2 con los literales repetidos eliminados

Procesamiento:

```
Si k1 y k2 != NIL
    Si es posible aplicar resolución
        evalúa a una lista con los literales repetidos eliminados de
        k1 + k2 con el rastro de lit eliminado
    en caso contrario
        evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.4
;; resolve-on
;;
;; Resolvente de dos clausulas
;;
;; RECIBE   : lit      - literal positivo
;;            K1, K2     - clausulas simplificadas
;; EVALUA A : res_lit(K1,K2)
;;                       - lista que contiene la
;;                         clausula que resulta de aplicar resolucion
;;                         sobre K1 y K2, con los literales repetidos
;;                         eliminados
;;
(defun resolve-on (lit k1 k2)
  (unless (or (null k1) (null k2))
    (when (possible-resolve-on lit k1 k2)
      (list (eliminate-repeated-literals
             (eliminar-rastro lit (append k1 k2)))))))
;;
;;  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, pese a su simplicidad, tiene grandes implicaciones computacionales, por lo que se ha decidido dividirla en funciones modulares para facilitar su comprensión.
- Tomamos dos cláusulas, comprobamos que en una de ellas aparece el literal sobre el que queremos aplicar resolución y en la otra la negación del mismo, y una vez eliminados estos literales de las cláusulas, se concatenan generando una nueva cláusula con todos los literales diferentes a aquéllos sobre los que se aplica la resolución.

5.4.5. Apartado 4.4.5

Para realizar este ejercicio se han codificado las siguientes funciones:

Función extract-positive-literals-clause

PSEUDOCÓDIGO

```
Entrada: clause (cláusula)
Salida:  lista de literales positivos

Procesamiento:
         si clause es NIL,
             evalúa a NIL
         en caso contrario,
             si clause[0] es un literal positivo,
                 concatenar clause[0] con extract-positive-literals-clause(clause[1:n])
             en caso contrario,
                 evalúa a extract-positive-literals-clause(clause[1:n])
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; extract-positive-literals-clause
;;
;; Extra la lista de literales positivos de una clausula
;;
;; RECIBE   : clause  - clausula
;; EVALUA A : NIL si no hay literales positivos en clause,
;;            lista de literales positivos en caso contrario
(defun extract-positive-literals-clause (clause)
  (if (null clause)
      nil
    (let ((first-literal (first clause))
          (next-it (extract-positive-literals-clause (rest clause))))
      (if (positive-literal-p first-literal)
          (cons first-literal next-it)
        next-it))))
;;
;; EJEMPLOS:
;; (extract-positive-literals-clause '((~ a) (~ b))) -> NIL
;; (extract-positive-literals-clause '(a b (~ c) d)) -> (A B D)
```

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

Función extract-positive-literals-cnf

PSEUDOCÓDIGO

```
Entrada: cnf (cláusula)
Salida:  lista de literales positivos

Procesamiento:
         lista = ()
         para cada x en cnf:
             lista += extract-positive-literals-clause(x)
         evalúa a eliminate-repeated-literals(lista)
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; extract-positive-literals-cnf
;;
;; Extra la lista de literales positivos de una expresion en FNC
;;
;; RECIBE   : cnf  - expresion en CNF
;; EVALUA A : NIL si no hay literales positivos en cnf,
;;            lista de literales positivos en caso contrario
(defun extract-positive-literals-cnf (cnf)
  (eliminate-repeated-literals (mapcan #'(lambda(x)
                                           (extract-positive-literals-clause x))
                                       cnf)))
;;
;; EJEMPLO:
;; (extract-positive-literals-cnf '((a b d) ((~ p) q) ((~ c) a b)
;; ((~ b) (~ p) d) (c d (~ a)))) -> (Q A B C D)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- El orden en el que se presenten los literales positivos dentro de la lista de retorno es irrelevante, lo importante es que estén todos para luego aplicar el algoritmo para determinar si una FBF es SAT o UNSAT.
- Se asume que, como en todas las funciones desde aquí en adelante, la FBF en FNC de entrada se encuentra sin conectores.

Finalmente:

Función build-RES

PSEUDOCÓDIGO

```
Entrada: lit (literal positivo)
         cnf (FBF en FNC simplificada)
Salida:  RES_lit(cnf) con las clausulas repetidas eliminadas

Procesamiento:
         neutral-clauses <= extract-neutral-clauses(lit, cnf)
         positive-clauses <= extract-positive-clauses(lit, cnf)
         negative-clauses <= extract-negative-clauses(lit, cnf)
         lista-res <= ()
         para i desde 0 hasta length(negative-clauses),
             para cada x en positive-clauses
                 lista-res += resolve-on(lit, x, negative-clauses[i])
         evalúa a eliminate-repeated-clauses(
             concatenar(neutral-clauses, lista-res))
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.4.5
;; build-RES
;;
;; Construye el conjunto de clausulas RES para una FNC
;;
;; RECIBE   : lit - literal positivo
;;            cnf    - FBF en FNC simplificada
;;
;; EVALUA A : RES_lit(cnf) con las clauses repetidas eliminadas
;;
(defun build-RES-aux (lit positivas negativas)
  (unless (or (null positivas) (null negativas))
    (append (mapcan #'(lambda(x) (resolve-on lit x (first negativas)))
              positivas)
            (build-RES-aux lit positivas (rest negativas)))))

(defun build-RES (lit cnf)
  (unless (null cnf)
    (eliminate-repeated-clauses
     (append (extract-neutral-clauses lit cnf)
             (build-RES-aux lit
                            (extract-positive-clauses lit cnf)
                            (extract-negative-clauses lit cnf))))))
;;
;;  EJEMPLOS:
;;
;; (build-RES 'p NIL) -> NIL
;; (build-RES 'P '((A  (~ P) B) (A P) (A B))) -> ((A B))
;; (build-RES 'P '((B  (~ P) A) (A P) (A B))) -> ((B A))
;; (build-RES 'p '(NIL)) -> (NIL)
;; (build-RES 'p '((p) ((~ p)))) -> (NIL)
;; (build-RES 'q '((p q) ((~ p) q) (a b q) (p (~ q)) ((~ p) (~ q))))
;; -> ((P) ((~ P) P) ((~ P)) (B A P) (B A (~ P)))
;; (build-RES 'p '((p q) (c q) (a b q) (p (~ q)) (p (~ q))))
;; -> ((A B Q) (C Q))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- En esta función, a priori, se repiten bastantes operaciones que no son necesarias, debido a que se realiza una resolución sobre todas las posibles parejas que surgen de la lista de cláusulas positivas y la lista de cláusulas negativas, lo cual genera un retorno que, si bien está exento de repeticiones por el uso de la función `eliminate-repeated-clauses`, puede contener expresiones que juntas no tienen mucho sentido. Por ejemplo, en uno de los ejemplos podemos ver que en la salida se incluyen las cláusulas (P) y ((∼ P)). Sin embargo, como esta función va a ser posteriormente utilizada de forma recursiva, este tipo de "errores" terminan por eliminarse.

5.5. Apartado 4.5

Función RES-SAT-p

PSEUDOCÓDIGO

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

Procesamiento:
         si cnf es NIL,
             cnf-new <= NIL
         en caso contrario,
             pos-literals <= extract-positive-literals-cnf(cnf)
```

```
                si pos-literals está vacío,
                    cnf-new <= (NIL)
                en caso contrario,
                    cnf-new <= cnf
                    para i = 0 en hasta length(pos-literals):
                        cnf-new <= simplifiy-cnf(build-RES(pos-literals[i], cnf-new))
            si cnf-new es NIL,
                evalúa a T
            en caso contrario (cnf-new es (NIL)),
                evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.5
;; RES-SAT-p
;;
;; 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-aux (pos-literals cnf)
  (cond
   ((null cnf) NIL)
   ((null pos-literals) (list NIL))
   (t (RES-SAT-aux (rest pos-literals)
                   (simplify-cnf (build-RES (first pos-literals)
                                            cnf))))))

(defun RES-SAT-p (cnf)
  (unless (equal (RES-SAT-aux (extract-positive-literals-cnf cnf)
                              cnf)
                 '(NIL))
    t))
;;
;; EJEMPLOS:
;;
;; SAT Examples
;;
;; (RES-SAT-p nil)  -> T
;; (RES-SAT-p '((p) ((~ q)))) -> T
;; (RES-SAT-p '((a b d) ((~ p) q) ((~ c) a b) ((~ b) (~ p) d)
;; (c d (~ a)))) -> T
;; (RES-SAT-p '(((~ p) (~ q) (~ r)) (q r) ((~ q) p) ((~ q))
;; ((~ p) (~ q) r))) -> T
;;
;; UNSAT Examples
;;
;; (RES-SAT-p '((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
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

COMENTARIOS:

- Cabe mencionar como recordatorio que cuando se obtiene una salida `NIL`, estamos ante una expresión que es SAT, mientras que cuando la salida es `(NIL)`, la expresión es UNSAT.
- Además, se puede comprobar que la salida de RES-SAT-aux siempre va a ser `NIL` o `(NIL)`, dado

que se genera a través de un condicional en el que con una condición se genera lo primero, con otra lo segundo y el tercer caso es una llamada recursiva.

- Usamos aquí la función `simplify-cnf` para intentar disminuir el número de llamadas recursivas de `RES-SAT-aux`, que aumentaría si incluyéramos expreisones más grandes.
- Como último detalle, se puede observar que la salida de `RES-SAT-aux` es la "inversa" de `RES-SAT-p`; cuando la primera devuelve `NIL`, la segunda devuelve `T`, mientras que se devuelve `NIL` en la segunda cuando la salida de `RES-SAT-aux` es `(NIL)`.

5.6. Apartado 4.6

Función logical-consequence-RES-SAT-p

PSEUDOCÓDIGO

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

 Procesamiento:
          α <= wff-infix-to-cnf(wff)
          β <= wff-infix-to-cnf(¬ w)
          si RES-SAT-p(concatenar(α, β)) es NIL,
              evalúa a T
          en caso contrario,
              evalúa a NIL
```

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 4.6:
;; logical-consequence-RES-SAT-p
;;
;; 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)
  (let ((alpha (wff-infix-to-cnf wff))
        (beta (wff-infix-to-cnf (list +not+ w))))
    (when (null (RES-SAT-p (append alpha beta)))
      t)))
;;
;;  EJEMPLOS:
;;
;; (logical-consequence-RES-SAT-p NIL 'a) -> NIL
;; (logical-consequence-RES-SAT-p NIL NIL) -> NIL
;; (logical-consequence-RES-SAT-p '(q ^ (~ q)) 'a) -> T
;; (logical-consequence-RES-SAT-p '(q ^ (~ q)) '(~ a)) -> T
;; (logical-consequence-RES-SAT-p '((p => (~ p)) ^ p) 'q) -> T
;; (logical-consequence-RES-SAT-p '((p => (~ p)) ^ p) '(~ q)) -> T
;; (logical-consequence-RES-SAT-p '((p => q) ^ p) 'q) -> T
;; (logical-consequence-RES-SAT-p '((p => q) ^ p) '(~ q)) -> NIL
;; (logical-consequence-RES-SAT-p '(((~ p) => q) ^ (p => (a v (~ b))) ^
;;   (p => ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q)))) '(~ a)) -> T
;; (logical-consequence-RES-SAT-p '(((~ p) => q) ^ (p => (a v (~ b))) ^
;;   (p => ((~ a) ^ b)) ^ ( (~ p) => (r  ^ (~ q)))) 'a) -> T
;; (logical-consequence-RES-SAT-p '(((~ p) => q) ^ (p => ((~ a) ^ b)) ^
;;   ( (~ p) => (r  ^ (~ q)))) 'a) -> NIL
;; (logical-consequence-RES-SAT-p '(((~ p) => q) ^ (p => ((~ a) ^ b)) ^
```

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

COMENTARIOS:

- Una vez disponemos de todas las funciones construidas previamente, esta última función es de lo más sencillo. Transformamos cada FBF en infijo a una FNC sin conectores, y comprobamos la condición de que $\{\alpha, \sim \mathtt{w}\}$ sea una expresión SAT o UNSAT. En el primer caso, `w` no es consecuencia lógica de α, mientras que en el caso contrario sí lo es.

6. Ejercicio 5

6.1. Apartado 5.1

Supongamos que tenemos un árbol binario completo de profundidad 2; es decir, tenemos un nodo padre que cuyo valor es "1", sus hijos serán "2" y "3", los hijos del 2 serán "4" y "5" y los hijos del 3 serán "6" y "7". Supongamos también que "6" es el nodo meta, y el nodo inicial es el "1".

El algoritmo BFS actuará de la siguiente forma:

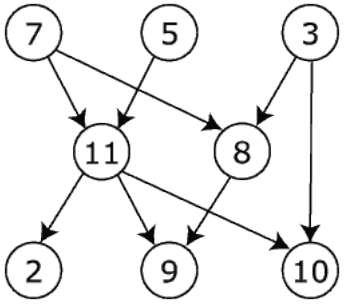
1. Explora el nodo "1", no es meta y ya ha acabado con todos los nodos disponibles para explorar; elimina "1"'de la lista y añade a sus vecinos, "2" y "3".
2. Explora el nodo "2", no es meta, lo elimina de la lista, añade a sus vecinos "4" y "5" y avanza al siguiente.
3. Explora el nodo "3", no es meta, lo elimina de la lista, añade a sus vecinos "6" y "7" y avanza al siguiente.
4. Mismo proceso con "4" y "5" sin añadir más nodos a la lista ya que no tienen vecinos, hasta que se llega al "6" que sí es meta.

En el grafo de ejemplo del problema ocurrirá lo siguiente (suponiendo que el nodo inicial es **e** y el nodo final es **d**:

1. Explora "e", no es meta, lo elimina y añade a la lista sus vecinos "b" y "f".
2. Explora "b", no es meta, lo elimina y añade sus vecinos "d" y "f".
3. Explora "f", no es meta, lo elimina y no tiene vecinos, así que no añade nada.
4. Explora "d", es meta y termina.

Como podemos ver, lo que se hace es implementar una cola, con la que los primeros elementos que se introducen al ser vecinos de nodos exploraron, son los primeros que se exploran en las siguientes iteraciones.

Por último, en el siguiente grafo:



Si queremos ir desde "7" hasta "10", se darían los siguientes pasos:

1. Explora "7", no es meta, lo elimina y añade a la cola sus vecinos "11" y "8"
2. Explora "11", no es meta, lo elimina y añade a la cola sus vecinos "2", "9" y "10"
3. Explora "8", no es meta, lo elimina y añade a la cola su vecino "9"
4. Explora "2", no es meta, lo elimina y no tiene vecinos
5. Explora "9", no es meta, lo elimina y no tiene vecinos
6. Explora "10", que es la meta.

6.2. Apartado 5.2

PSEUDOCÓDIGO

```
Entrada: end (nodo final)
         start (nodo inicial)
         Q (cola de caminos)
         G (grafo)
Salida:  Lista con los nodos a recorrer.



Procesamiento (implementación común):
         para todos los nodos x del grafo
             state(x) = NOT-VISITED
             dist(x) = ∞
             parent(x) = NIL
         state(start) = VISITED
         dist(start) = 0
         parent(start) = NIL
         queue = ()
         add-queue(queue, start)
         mientras queue no esté vacía,
             node = extract-queue(queue)
             para cada vecino v de node
                 si state(v) = NOT-VISITED,
                     state(v) = VISITED
                     dist(v) = dist(node) + 1
                     parent(v) = node
                     add-queue(queue, v)

Procesamiento (implementación del 5.3):
         si Q está vacía,
             evalúa a NIL
         en caso contrario,
             camino <= Q[0]
```

```
nodo <= Q[0][0]
si nodo es meta,
    evalúa a inverso(camino)
en caso contrario,
    nuevos-nodos <= ()
    para todo x tal que x es vecino de nodo
        nuevos-nodos += concatenar(x, camino)
    Q <= concatenar(Q[1:n], nuevos-nodos)
evalúa a bfs(end, Q, G)
```

6.3. Apartado 5.5

El algoritmo propuesto resuelve el problema de encontrar el final óptimo para todos aquellos caminos que están contenido en la cola inicial; es decir, de todos los caminos de la cola, al final del algoritmo se encuentra el camino óptimo para llegar al nodo meta siempre y cuando el inicio de dicho camino estuviera contenido entre los caminos de la cola inicial.

Por tanto, lo único que hace la función `shortest-path` es inicializar la cola con un único camino, que es el formado por el nodo inicial. De esta forma, se encuentra un camino óptimo cuyo inicio sea este único camino contenido en la cola, que es, efectivamente, el nodo inicial, por lo que el camino solución es el óptimo entre el nodo inicial y el nodo final.

6.4. Apartado 5.6

Tras evaluarlo utilizando la macro *trace* se ha obtenido la siguiente salida:

```
(shortest-path 'a 'f '((a d) (b d f) (c e) (d f) (e b f) (f)))
    0: (BFS F ((A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
        1: (NEW-PATHS (A) A ((A D) (B D F) (C E) (D F) (E B F) (F)))
        1: NEW-PATHS returned ((D A))
        1: (BFS F ((D A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
            2: (NEW-PATHS (D A) D ((A D) (B D F) (C E) (D F) (E B F) (F)))
            2: NEW-PATHS returned ((F D A))
            2: (BFS F ((F D A)) ((A D) (B D F) (C E) (D F) (E B F) (F)))
            2: BFS returned (A D F)
        1: BFS returned (A D F)
    0: BFS returned (A D F)
(A D F)
```

Viendo los retornos de la función `NEW-PATHS` podemos observar los caminos que se van creando desde el nodo inicial A. Dado que su único vecino es D, y el único vecino de D es F, se acaba creando el camino F-D-A; y como F es meta, se devuelve el camino invertido (empezando desde A) que es A-D-F.

6.5. Apartado 5.7

El grafo quedará representado por la lista `((a b c d e) (b a d e f) (c a g) (d a b g h) (e a b g h) (f b h) (g c d e h) (h d e f g))`, por lo que la expresión a evaluar será:

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

Que resulta en `(F B A C)`.

6.6. Apartado 5.8

Supongamos que en el grafo del apartado anterior las aristas (C, A) y (C, G) son dirigidas y en ambas el inicio es el nodo C; por tanto, tendríamos una nueva lista representando a este grafo: `((a b d e) (b a d e f) (c a g) (d a b g h) (e a b g h) (f b h) (g d e h) (h d e f g))`. Si tratamos de ir

desde el nodo F hasta el nodo C, como no hay un camino posible, el algoritmo se queda en un bucle infinito viajando en cualquiera de los ciclos que se forman entre F, B, E, H...

Por tanto, nos tenemos que encargar de que una vez exploramos un nodo, no lo volvamos a explorar nunca más. Se puede conseguir eliminando los nodos que se exploran de la lista que representa el grafo, que se corresponde al último argumento de la llamada recursiva de `BFS`, ya que cuando `NEW-PATHS` intente buscar adyacencias por segunda vez, no tendrá esa información disponible nunca más. Esto da lugar a:

CÓDIGO

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EJERCICIO 5.8
;; bfs-improved, shortest-path-improved
;;
;; Solucionan el problema de la inexistencia de caminos
;; para grafos con ciclos
;;
;; Entradas y salidas iguales que para bfs y shortest-path
;;
(defun bfs-improved (end queue net)
  (if (null queue) '()
    (let* ((path (first queue))
           (node (first path)))
      (if (eql node end)
          (reverse path)
        (bfs-improved end
                      (append (rest queue)
                              (new-paths path node net))
                      (remove (assoc node net) net))))))

(defun shortest-path-improved (start end net)
  (bfs-improved end (list (list start)) net))
;;
;; EJEMPLO:
;; (shortest-path-improved 'f 'c '((a b d e) (b a d e f)
;; (c a g) (d a b g h) (e a b g h) (f b h) (g d e h) (h d e f g)))
;; -> NIL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```