PRÁCTICA 1: LISP

FECHAS DE ENTREGA:

- Entrega parcial (ejercicios 1, 2 y 3):
  - Todos los grupos: 12 de febrero de 2018 (23:55)
- Entrega final (todos los ejercicios, incluyendo los ejercicios 1,2 y 3 corregidos):
  - Todos los grupos: 07 de marzo de 2018 (23:55)

EVALUACIÓN:

- Normativa de prácticas.
- Criterios de evaluación.
- Pesos en la evaluación:
  - Correcto funcionamiento: 40 %
  - Estilo de programación: 30 %
  - Memoria: 30 %
- Errores más frecuentes (a evitar).

Forma de entrega: Según lo dispuesto en las normas de entrega generales, publicadas en Moodle

- El código de los ejercicios 1, 2, 3 y 5 debe estar en un único fichero.
- El código del ejercicio 4 debe estar en el fichero `p1ej4.lisp` facilitado.
- La evaluación del código no debe dar errores en Allegro CL 6.2 ANSI with IDE (recuerda: CTRL+A CTRL+E debe evaluar todo el código sin errores).

Material recomendado:

- En cuanto a estilo de programación LISP:
  `http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/lisp/part1/faq.html`
- Como referencia de funciones LISP:
  `http://www.lispworks.com/documentation/common-lisp.html`

1. Similitud Coseno. (1.5 puntos)

Una de las principales aplicaciones de la inteligencia artificial aplicada al procesamiento de lenguaje natural es averiguar como de similares son varios textos y clasificar a estos en categorías. Por ejemplo, se puede clasificar que páginas web pertenecen a la categoría de deportes, a la categoría de noticias, a la categoría de videojuegos, etc...

Una metodología básica es que si conseguimos representar a las categorías por vectores que las definan, si tenemos un algoritmo que nos diga cuál es la distancia de cualquier vector que represente una página web a los vectores de categorías, entonces el nuevo vector (la nueva página web) quedará clasificado en la categoría que minimice la distancia con respecto al vector de la nueva página web y el de las categorías.

Una forma básica de implementar esta metodología es la siguiente. Para representar a las categorías en vectores se elige una serie de términos que las representen. Por ejemplo, para la categoría de deportes podemos elegir los siguientes términos: {Jugador, fútbol, partido, árbitro, equipo, baloncesto}, para la categoría de noticias: {Suceso, ha pasado, acontecimiento, noticia, declarado, entrevista}... Podremos elegir tantos términos como queramos. Cuando hayamos terminado, concatenamos todos estos términos en un único vector $\mathbf{v}$ = {term_1_cat_1,...,term_n_cat_1, ... , term_1_cat_N, ..., term_m_cat_N}.

Un nuevo vector será generado por un contador de cada término que aparezca en cada nueva página web. Por ejemplo, si en una nueva página web ha aparecido 3 veces la palabra fútbol, 2 veces partido y una vez equipo, entonces el vector V de esa página web tendrá esos contadores en la posición correspondiente de cada término y 0 en el resto. Para representar las categorías, una persona experta en cada categoría debe personalizar un vector para cada una de ellas. Por ejemplo, para deporte podemos tener que aparezcan entre 1 y 3 veces cada término de deportes y 0 el resto de términos.

Una vez hecha la representación, queda definir como calcular la distancia entre estos vectores. Una medida clásica que se ha utilizado para esta tarea es la similitud coseno, sobre la que versa este ejercicio. Dados dos vectores $\mathbf{x} = \{x_1, ..., x_n\}$ y $\mathbf{y} = \{y_1, ..., y_n\}$ en $\mathcal{R}^n$, la similitud coseno $\cos(\theta)$, viene dada por la siguiente expresión:

$$\text{cos\_similarity}(\mathbf{x},\mathbf{y}) = \cos(\theta) = \frac{\mathbf{x}\cdot\mathbf{y}}{\|\mathbf{x}\|_2\|\mathbf{y}\|_2} = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$$

El resultado está acotado entre -1 (totalmente opuesto) y 1 (totalmente similar). Se nos ha pedido implementar esta medida en LISP. Si algún valor de este vector fuese negativo, la función deberá devolver NIL, al igual que si la longitud de los vectores es distinta entre sí.

1.1 Implementa una función que calcule la similitud coseno entre dos vectores, x e y, de dos formas:

Recursiva:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 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) ...)
```

Usando 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) ...)
```

Pista: Haz funciones auxiliares para este ejercicio, recuerda, divide y vencerás. Se pide recursión y mapcar, pero de los componentes principales que forman la expresión. Calcúlalos de esa forma en funciones auxiliares, y después en la función principal programa el cálculo final.

1.2 Codifica una función que reciba un vector que representa a una categoría, un conjunto de vectores y un nivel de confianza que esté entre 0 y 1. La función deberá retornar el conjunto de vectores ordenado según mas se parezcan a la categoría dada si su semejanza es superior al nivel de confianza.

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

1.3 Clasificador por similitud coseno. Codifica una función que reciba un conjunto de categorías, cuyo primer elemento es su identificador, un conjunto de vectores que representan textos, cuyo primer elemento es su identificador, y devuelva una lista que contenga pares definidos por un identificador de la categoría que maximiza la similitud coseno con respecto a ese texto y el resultado de la similitud coseno. El tercer argumento es la función usada para evaluar la similitud coseno (mapcar o recursiva).

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

Los identificadores serán autoincrementales empezando por 1 para las categorías y los textos según el orden por el que sean suministrados a la función. Un ejemplo de categorías y textos sería:

```
»(setf cats '((1 43 23 12) (2 33 54 24)))
»(setf texts '((1 3 22 134) (2 43 26 58)))
```

1.4 Haz pruebas llamando a sc-classifier con las distintas variantes de la distancia coseno y para varias dimensiones de los vectores de entrada. Verifica y compara tiempos de ejecución. Documenta en la memoria las conclusiones a las que llegues.

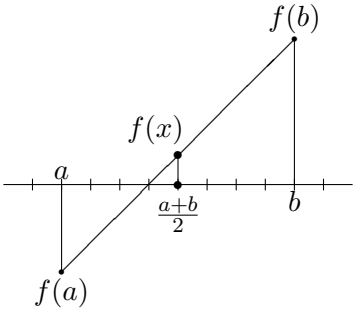
Un ejemplo sencillo de llamadas que podrían hacerse sería el siguiente:

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

2. Raíces de una función. (1.5 puntos)

Dada una función $f(x) : \mathbb{R} \to \mathbb{R}$ y un intervalo $[a, b] \subseteq \mathbb{R}$, queremos encontrar la raíz o raíces de f, es decir, el valor o valores $x \in [a, b]$ tales que $f(x) = 0$.

Un método muy sencillo para encontrar una raíz aproximada de la función es el método de *bisección*. Supongamos que los valores $f(a)$ y $f(b)$ tengan signo opuesto, es decir $\text{sgn}(f(a))\text{sgn}(f(b)) < 0$. Si la función es continua (algo que siempre asumiremos), entonces hay una raíz en $[a, b]$. Para reducir la incertidumbre sobre la localización de esta raíz consideramos el punto medio $x = (a + b)/2$ y consideramos el valor $f(x)$. Es fácil darse cuenta (véase el diagrama) que si $\text{sgn}(f(a))\text{sgn}(f(x)) < 0$, entonces hay una raíz en $[a, x]$, mientras si $\text{sgn}(f(x))\text{sgn}(f(b)) < 0$ hay una raíz en $[x, b]$ (si $f(x) = 0$ hemos encontrado la raíz: si este no es el caso es fácil ver que una de las dos desigualdades tiene que cumplirse dado que, ex hipothesi, $\text{sgn}(f(a))\text{sgn}(f(b)) < 0$).



Ahora que sabemos en que parte del intervalo está la raíz, podemos llamar recursivamente nuestra función con un intervalo la mitad del original. Seguimos así recursivamente hasta llegar a un intervalo $[a, b]$ de longitud menor que una tolerancia establecida.

El método es sencillo, pero presenta ciertas limitaciones:

i) si en el intervalo $[a, b]$ hay un número par de raíces, entonces $\text{sgn}(f(a))\text{sgn}(f(b)) > 0$, y la función no encontrará ninguna raíz;

ii) si en el intervalo $[a, b]$ hay un número impar de raices, el método sólo encontrará una.

2.1 Implemente una función *bisect* que aplique el método de bisección para encontrar *una* raíz de una función en un intervalo dado;

```
;;  Finds a root of f between the points a and b using bisection.
;;
;;  If f(a)f(b)>0 there is no guarantee that there will be a root in the
;;  interval, and the function will return NIL.
;;
;;  f: function of a single real parameter with real values whose root
;;     we want to find
;;  a: lower extremum of the interval in which we search for the root
;;  b: b>a upper extremum of the interval in which we search for the root
;;  tol: tolerance for the stopping criterion: if b-a < tol the function
;;       returns (a+b)/2 as a solution.
;;
(defun bisect (f a b tol)
 ...
)
```

Ejemplos:

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

2.2 Implemente una función *allroot* que recibe una función, una lista de valores y una tolerancia. Si entre los elementos $l[i]$ y $l[i+1]$ de la lista hay una raíz, la función devuelve el valor de la raíz. El resultado es una lista con todas las raíces encontradas.

Formalmente, si $x = [x_1, \ldots, x_n]$, $r = [r_1, \ldots, r_m]$ y $r = ((allroot)fl)$, entonces:

$$\forall i \, (1 \leq i \leq m \rightarrow \exists k \, (l[k] < r[i] < l[k+1], f(r[k]) = 0))$$

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

Ejemplos:

```
(allroot #'(lambda(x) (sin (* 6.28 x))) '(0.25 0.75 1.25 1.75 2.25) 0.0001)
;;---> (0.50027466 1.0005188 1.5007629 2.001007)
(allroot #'(lambda(x) (sin (* 6.28 x))) '(0.25 0.9 0.75 1.25 1.75 2.25) 0.0001)
;;---> (0.50027466 1.0005188 1.5007629 2.001007)
```

2.3 Implemente una función *allind* que recibe una funcion, dos límites del intervalo, un valor entero N y la tolerancia. La función divide el intervalo en 2^N secciones y busca en cada una una raíz de la función (dentro de la tolerancia). Devuelve una lista con las raices encontradas.

```
;;
;;  Divides an interval up to a specified length and find all the roots of
;;  the function f in the intervals thus obtained.
;;
;;  Parameters:
;;
;;  f: function of a single real parameter with real values whose root
;;     we want to find
;;  a: lower extremum of the interval in which we search for the root
;;  b: b>a upper extremum of the interval in which we search for the root
;;  N: Exponent of the number of intervals in which [a,b] is to be divided:
;;     [a,b] is divided into 2^N intervals
;;  tol: tolerance for the stopping criterion: if b-a < tol the function
;;       returns (a+b)/2 as a solution.
```

```
;;
;;  The interval (a,b) is divided in intervals (x[i], x[i+i]) with
;;  x[i]= a + i*dlt; a root is sought in each interval, and all the roots
;;  thus found are assembled into a list that is returned.
;;
;;
;;  Hint:
;;  One might find a way to use allroot to implement this function. This is
;;  possible, of course, but there is a simple way of doing it recursively
;;  without using allroot.
;;
(defun allind (f a b N tol) ...)
```

Ejemplos:

```
(allind #'(lambda(x) (sin (* 6.28 x))) 0.1 2.25 0.1 0.0001)
;;---> (0.50027084 1.0005027 1.5007347 2.0010324)
```

3. Combinación de listas (1 punto)

3.1 Define una función que combine un elemento dado con todos los elementos de una lista:

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

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

3.2 Diseña una función que calcule el producto cartesiano de dos listas:

```
(defun combine-lst-lst (lst1 lst2) ...)

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

3.3 Diseña una función que calcule todas las posibles disposiciones de elementos pertenecientes a N listas de forma que en cada disposición aparezca únicamente un elemento de cada lista:

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

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

4. Inferencia en lógica proposicional (5 puntos)

En este apartado de la práctica implementaremos varios algoritmos de inferencia en lógica proposicional para bases de conocimiento en forma normal conjuntiva (FNC). Para ello, utilizaremos la representación de una FNC (= conjunción de cláusulas disyuntivas), como un conjunto (en LISP una lista) de cláusulas. Asimismo, una cláusula disyuntiva (= disyunción de literales) será representada como un conjunto (en LISP una lista) de literales.

- En primer lugar construiremos funciones para convertir una FBF en formato infijo a una FBF en FNC, representada como lista de listas.
- A continuación construiremos un motor de inferencia, que puedes utilizar para resolver problemas de lógica proposicional.

NOTA: Todo el código que desarrolles en este apartado debes incluirlo en el fichero `p1ej4.lisp`, en los lugares indicados.

4.1 Predicados en LISP para definir literales, FBFs en forma prefijo e infijo, cláusulas y FNCs (1 punto)

En lógica proposicional se definen:

- **Literal:**
  - *Literal positivo:* Un átomo.
    En LISP: un átomo LISP distinto de los que se utilizan para representar valores de verdad (T, NIL) y de los que representan conectores.
    Ejemplo: `'p`.
  - *Literal negativo:* Negación de un literal positivo.
    En LISP: lista cuyo primer elemento es el conector que representa la negación y cuyo segundo elemento es un literal positivo.
    Ejemplo: `'(¬p)`.
- **Cláusula (cláusula disyuntiva):** Fórmula bien formada que consiste en una disyunción de literales.
- **Forma normal conjuntiva (FNC):** Fórmula bien formada que consiste en una conjunción de cláusulas disyuntivas.

Al comienzo del programa `p1ej4.lisp` se definen los símbolos que representan los conectores lógicos y los predicados para evaluar si una expresión LISP es un valor de verdad o un conector. Los conectores pueden tomar 1 argumento (unario), 2 argumentos (binario) o n argumentos (n $\geq$ 0, n-ario).

Ejercicio 4.1.1. Escribe una función LISP para determinar si una expresión es un literal positivo. Completa en el fichero `p1ej4.lisp` el código de la función `positive-literal-p`.

Ejercicio 4.1.2. Escribe una función LISP para determinar si una expresión es un literal negativo. Completa en el fichero `p1ej4.lisp` el código de la función `negative-literal-p`.

Ejercicio 4.1.3. Escribe una función LISP para determinar si una expresión es un literal. Completa en el fichero `p1ej4.lisp` el código de la función `literal-p`.

Ejercicio 4.1.4. La función `wff-prefix-p` facilitada en el fichero `p1ej4.lisp` implementa un predicado para determinar si una determinada expresión está en formato prefijo. Tomando el código de esta función como referencia, implementa una nueva función LISP para determinar si una expresión dada está en formato infijo. Completa en el fichero `p1ej4.lisp` el código de la función `wff-infix-p`. Puedes utilizar funciones auxiliares si lo consideras necesario.

Ejercicio 4.1.5. La función `prefix-to-infix` facilitada en el fichero `p1ej4.lisp` transforma una FBF en formato prefijo a una FBF en formato infijo. Tomando el código de esta función como referencia, implementa una nueva función LISP para transformar FBFs en formato infijo a formato prefijo. Completa en el fichero `p1ej4.lisp` el código de la función `infix-to-prefix`.

Ejercicio 4.1.6. Escribe una función LISP para determinar si una FBF en formato prefijo es una cláusula (disyuntiva). Completa en el fichero `p1ej4.lisp` el código de la función `clause-p`. Puedes utilizar funciones auxiliares si lo consideras necesario.

NOTA: Una *cláusula* (disyuntiva) es una disyunción de literales, es decir, debería tener el formato `'(∨ lit1 lit2 ... litn)`.

CASOS ESPECIALES:

- Cláusula vacía (su valor de verdad es False): `'(∨)`
- Cláusula con un único literal: `'(∨ lit1)`

Ejercicio 4.1.7. Escribe una función LISP para determinar si una FBF en formato prefijo está en FNC. Completa en el fichero `p1ej4.lisp` el código de la función `cnf-p`. Puedes utilizar funciones auxiliares si lo consideras necesario.

NOTA: Una *FNC* (Forma Normal Conjuntiva) es una conjunción de cláusulas, es decir, debería tener el formato `'(∧ K1 K2 ... Kn)`.

CASOS ESPECIALES:

- Conjunción vacía (su valor de verdad es True): `'(∧)`
- Conjunción con cláusula vacía (su valor de verdad es False): `'(∧ (∨))`
- Conjunción con una única cláusula formada por un único literal: `'(∧ (∨ lit1))`

4.2 Algoritmo de transformación de una FBF a FNC (1 punto)

En este apartado implementarás el algoritmo de transformación de una FBF a forma normal conjuntiva. El algoritmo consta de los siguientes pasos:

PASO 1: Eliminación del conector bicondicional (`<=>`). La regla a aplicar es:

`(fbf1 <=> fbf2)` ≡ `((fbf1 => fbf2)` ∧ `(fbf2 => fbf1))`

donde `fbf1` y `fbf2` son FBFs que deben ser previamente procesadas (de forma recursiva) para que no contengan el conector bicondicional.

Ejercicio 4.2.1. La función `eliminate-biconditional` facilitada en el fichero `p1ej4.lisp` elimina el conector bicondicional en una FBF. Incluye comentarios en el código de la función `eliminate-biconditional`.

PASO 2: Eliminación del conector condicional (`=>`). La regla a aplicar es:

`(fbf1 => fbf2)` ≡ `((¬ fbf1)` ∨ `fbf2)`

donde `fbf1` y `fbf2` son FBFs que deben ser previamente procesadas (de forma recursiva) para que no contengan el conector condicional.

Ejercicio 4.2.2. Escribe una función LISP para eliminar el conector condicional en una FBF. Completa en el fichero `p1ej4.lisp` el código de la función `eliminate-conditional`.

PASO 3: Reducción del ámbito de la negación (¬). Deben aplicarse las leyes de De Morgan y de eliminación de negaciones repetidas hasta que el conector negación se aplique únicamente a átomos:

- Leyes de De Morgan:

  `(¬ (fbf1 ∨ fbf2 ∨ fbf3)) ≡ ((¬ fbf1) ∧ (¬ fbf2) ∧ (¬ fbf3))`
  `(¬ (fbf1 ∧ fbf2 ∧ fbf3)) ≡ ((¬ fbf1) ∨ (¬ fbf2) ∨ (¬ fbf3))`

- Eliminación de la doble negación:

  `(¬ (¬ c1)) ≡ c1`

Si las cláusulas `fbf1`, `fbf2` o `fbf3` no son literales positivos, entonces deberán volverse a aplicar las reglas de este paso (paso 3) sobre cada una de ellas.

Ejercicio 4.2.3. Escribe una función LISP para reducir el ámbito de la negación en una FBF. Completa en el fichero `p1ej4.lisp` el código de la función `reduce-scope-of-negation`. Puedes utilizar la función auxiliar `exchange-and-or` que te facilitamos, así como utilizar otras funciones auxiliares si lo consideras necesario.

PASO 4: Traducción a FNC distribuyendo y agrupando conectores conjunción (∧) y disyunción (∨). A partir de una cláusula en la que hay conectores de conjunción y de disyunción sin ningún orden o jerarquía predefinidos, se debe obtener una FBF en forma normal conjuntiva (conjunción de disyunciones de literales). Para ello se debe utilizar la siguiente regla general:

`((c1 ∧ c2) ∨ (c3 ∨ c4)) ≡ ((c1 ∧ c2) ∨ c3 ∨ c4)`
`≡ ((c1 ∨ c3 ∨ c4) ∧ (c2 ∨ c3 ∨ c4))`

Ejercicio 4.2.4. La función `cnf` facilitada en el fichero `p1ej4.lisp`, junto con las funciones auxiliares `simplify`, `exchange-NF`, `exchange-NF-aux` y `combine-elt-lst`, traduce una FBF (en formato prefijo, sin conectores condicional y bicondicional y con la negación siempre en literales negativos) a FNC. Incluye comentarios en el código de todas estas funciones.

Una vez implementado el algoritmo de transformación a FNC, implementarás algunas funciones adicionales para simplificar las listas que representan la FBF.

Ejercicio 4.2.5. Escribe una función LISP que, dada una FBF en FNC, elimine los conectores conjunción (∧) y disyunción (∨) para pasar a un formato de lista de listas en el que la conjunción de cláusulas y la disyunción de literales dentro de cada cláusula están sobreentendidas. Completa en el fichero `p1ej4.lisp` el código de la función `eliminate-connectors`.

Ejercicio 4.2.6. Escribe una función LISP que transforme una FBF en notación infija a FNC con conjunciones y disyunciones implícitas. Completa en el fichero `p1ej4.lisp` el código de la función `wff-infix-to-cnf`.

IMPORTANTE: A partir de este momento, una FBF en FNC estará representada como una lista de listas de literales. Es decir, en las listas que representan cláusulas (disyunciones de literales) se sobreentiende el conector ∨. En la lista de listas que representan la FNC (conjunción de cláusulas) se sobreentiende el conector ∧.

Ejemplo. Representaciones LISP de FBFs en FNC:

- Con conectores: `(∧ (∨ p (¬ q)) (∨ q) (∨ (¬ a)) (∨ s e f) (∨ b))`
- Sin conectores: `((p (¬ q))(q) ((¬ a)) (s e f)(b))`

Con esta nueva representación:

- **Cláusula (cláusula disyuntiva):** Fórmula bien formada que consiste en una disyunción de literales.
  En LISP: Lista de literales.
  Nota: La lista vacía puede representar una cláusula. Corresponde a la cláusula vacía, cuyo valor de verdad es FALSE.

- **Forma normal conjuntiva (FNC):** Fórmula bien formada que consiste en una conjunción de cláusulas disyuntivas.
  En LISP: Lista de cláusulas.
  Nota: La lista vacía puede representar una FNC. Corresponde a una conjunción de cláusulas vacía, cuyo valor de verdad es TRUE.

4.3 Simplificación de FBFs en FNC (1 punto)

Una vez tenemos la FBF en forma normal conjuntiva, la simplificaremos eliminando repeticiones (literales repetidos en una cláusula y cláusulas repetidas en la FNC), cláusulas subsumidas y tautologías. En los siguientes ejercicios implementarás las funciones LISP que realizan estas simplificaciones.

Eliminación de repeticiones:

Ejercicio 4.3.1. Escribe una función LISP que elimine los literales repetidos en una cláusula. Completa en el fichero `p1ej4.lisp` el código de la función `eliminate-repeated-literals`.

Ejercicio 4.3.2. Escribe una función LISP que elimine las cláusulas repetidas en una FNC. Completa en el fichero `p1ej4.lisp` el código de la función `eliminate-repeated-clauses`. Puedes utilizar funciones auxiliares si lo consideras necesario.

Eliminación de cláusulas subsumidas:

La cláusula K_1 subsume a la cláusula K_2 si todos los literales de K_1 están en K_2.

Ejercicio 4.3.3. Escribe una función LISP que determine si una cláusula subsume a otra. Completa en el fichero `p1ej4.lisp` el código de la función `subsume`.

Ejercicio 4.3.4. Escribe una función LISP que elimine las cláusulas subsumidas en una FNC. Completa en el fichero `p1ej4.lisp` el código de la función `eliminate-subsumed-clauses`. Puedes utilizar funciones auxiliares si lo consideras necesario.

Eliminación de tautologías:

Ejercicio 4.3.5. Escribe una función LISP que determine si una cláusula es tautología. Completa en el fichero `p1ej4.lisp` el código de la función `tautology-p`. Puedes utilizar funciones auxiliares si lo consideras necesario.

Ejercicio 4.3.6. Escribe una función LISP que elimine las cláusulas en una FNC que son tautología. Completa en el fichero `p1ej4.lisp` el código de la función `eliminate-tautologies`.

Simplificación de una FNC:

Ejercicio 4.3.7. Escribe una función LISP que simplifique FBFs en FNC mediante la aplicación sucesiva de las eliminaciones anteriores: eliminación de literales repetidos en cada una de las cláusulas, eliminación de cláusulas repetidas, eliminación de tautologías y eliminación de cláusulas subsumidas. Completa en el fichero `p1ej4.lisp` el código de la función `simplify-cnf`.

4.4 Construcción de RES (1 punto)

Sean α una FBF en FNC y λ un literal positivo. Podemos escribir α como:

$$\alpha = \alpha_\lambda^{(0)} \cup \alpha_\lambda^{(+)} \cup \alpha_\lambda^{(-)}$$

Con:

- $\alpha_\lambda^{(0)}$ = Subconjunto de cláusulas de α que no contienen ni λ ni $\neg\lambda$.
- $\alpha_\lambda^{(+)}$ = Subconjunto de cláusulas de α que contienen λ.
- $\alpha_\lambda^{(-)}$ = Subconjunto de cláusulas de α que contienen $\neg\lambda$.

Si se han eliminado las tautologías de α, entonces se tiene que:

$$\alpha_\lambda^{(+)} \cap \alpha_\lambda^{(-)} = \emptyset$$

Y por tanto $\alpha_\lambda^{(0)} \cup \alpha_\lambda^{(+)} \cup \alpha_\lambda^{(-)}$ es una partición de α.

DEFINICIÓN: El conjunto $RES_\lambda(\alpha)$ es la unión entre $\alpha_\lambda^{(0)}$ y las cláusulas resultantes de resolver entre cláusulas de $\alpha_\lambda^{(+)}$ y cláusulas de $\alpha_\lambda^{(-)}$ respecto al átomo λ.

En los siguientes ejercicios implementarás las funciones LISP necesarias para construir el conjunto $RES_\lambda(\alpha)$.

Ejercicio 4.4.1. Escribe una función LISP que calcule el conjunto de cláusulas lambda-neutras ($\alpha_\lambda^{(0)}$) para una FNC. Completa en el fichero `p1ej4.lisp` el código de la función `extract-neutral-clauses`.

Ejercicio 4.4.2. Escribe una función LISP que calcule el conjunto de cláusulas lambda-positivas ($\alpha_\lambda^{(+)}$) para una FNC. Completa en el fichero `p1ej4.lisp` el código de la función `extract-positive-clauses`.

Ejercicio 4.4.3. Escribe una función LISP que calcule el conjunto de cláusulas lambda-negativas ($\alpha_\lambda^{(-)}$) para una FNC. Completa en el fichero `p1ej4.lisp` el código de la función `extract-negative-clauses`.

Ejercicio 4.4.4. Escribe una función LISP que calcule el resolvente entre dos cláusulas $res_\lambda(K_1, K_2)$. El resolvente $res_\lambda(K_1, K_2)$ es el resultado de aplicar resolución en λ sobre K_1 y K_2, con los literales repetidos eliminados. Completa en el fichero `p1ej4.lisp` el código de la función `resolve-on`.

Ejercicio 4.4.5. Escribe una función LISP que calcule el conjunto $RES_\lambda(\alpha)$ para una FNC α. Completa en el fichero `p1ej4.lisp` el código de la función `build-RES`. Puedes utilizar funciones auxiliares si lo consideras necesario.

4.5 Algoritmo para determinar si una FNC es SAT (0.5 puntos)

Observaciones importantes que pueden resultar de utilidad:

- Una cláusula vacía es insatisfacible (contradicción).
- Una FNC vacía es válida (tautología).

Por lo tanto, en nuestras expresiones LISP:

- `(NIL)` es una FNC insatisfacible (contradicción).
- `((A B (¬ C)) NIL ((¬ R))(P Q) (E))` es una FNC insatisfacible (contradicción).
- `NIL` es una FNC válida (tautología).

Para determinar si la FBF en FNC α es SAT, utilizaremos el siguiente algoritmo:

Data: λ_1, λ_2,..., λ_k literales positivos diferentes que aparecen en las cláusulas de α
$\alpha_0 = \alpha$;
$j = 0$;
repeat
  $j = j + 1$;
  $\alpha_j = RES_{\lambda_j}(\alpha_{j-1})$;
  simplificar(α_j);
until *se obtiene la cláusula vacía (α es UNSAT);*
o no se pueden hacer más resoluciones (α es SAT);

Ejercicio 4.5. Escribe una función LISP que compruebe si una FNC α es SAT calculando el conjunto $RES_\lambda(\alpha)$ para todos los átomos λ en α. Completa en el fichero `p1ej4.lisp` el código de la función `RES-SAT-p`. Puedes utilizar funciones auxiliares si lo consideras necesario.

4.6 Algoritmo para determinar si una FBF es consecuencia lógica de una base de conocimiento (0.5 puntos)

En este último apartado implementarás el algoritmo para determinar si una FBF w es consecuencia lógica de una base de conocimiento α.

Sea α una base de conocimiento (se supone SAT), y sea w una FBF. Para determinar si w es o no consecuencia lógica de α, procedemos del siguiente modo:

1. Transformar α a FNC (α_{FNC}).
2. Transformar $\neg w$ a FNC.
3. Aplicar resolución para determinar si el conjunto de cláusulas $\hat{\alpha} = \alpha_{FNC} \cup \neg w$ es SAT.

Si la resolución basada en algún algoritmo para inferencia completo genera la cláusula vacía entonces $\hat{\alpha}$ **es UNSAT**, y, por tanto, w **es consecuencia lógica de** α.

En caso contrario, $\hat{\alpha}$ **es SAT** y, por tanto, w **no es consecuencia lógica de** α.

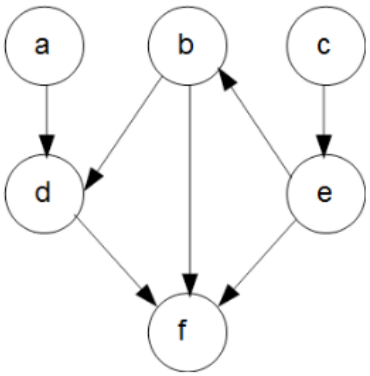
Ejercicio 4.6. Escribe una función LISP que, utilizando `RES-SAT-p`, compruebe si una FBF w es consecuencia lógica de una base de conocimiento α. Completa en el fichero `p1ej4.lisp` el código de la función `logical-consequence-RES-SAT-p`.

5. Búsqueda en anchura (1 punto)

Los *grafos* son un tipo de datos de los más utilizados en informática para modelar distintos problemas. Matemáticamente, un *grafo* viene determinado por un conjunto de *nodos* o *vértices* (abreviado normalmente mediante V), y un conjunto de *aristas* entre pares de vértices (denotado comúnmente E). Dado que los elementos de E son pares de vértices en V, se suelen usar dos representaciones de grafos:

- *Matriz de adyacencia*: una matriz cuadrada del tamaño de V en la que la casilla (i, j) es 1 si existe una arista del vértice i al vértice j, y 0 si no existe tal arista.
- *Listas de adyacencia*: una lista por cada vértice en la que se especifican sus *vecinos* o *vértices adyacentes* (es decir, con qué nodos está conectado mediante alguna arista).

Dado que LISP trabaja nativamente con listas usaremos una representación basada en listas de adyacencia. En concreto, un grafo viene dado por una lista de listas de adyacencia, donde el primer elemento de cada lista de adyacencia es el vértice origen y el resto son sus vecinos. Por ejemplo, dado el grafo



Su representación será la lista

```
((a d) (b d f) (c e) (d f) (e b f) (f))
```

La *búsqueda en anchura* (Breadth-First Search, BFS) es probablemente el algoritmo más intuitivo para recorrer un grafo. El nombre proviene del hecho de que la búsqueda se realiza "a lo ancho" a partir de un nodo raíz. Primero se exploran los vecinos de dicho nodo raíz (vecinos de primer nivel). A continuación, para cada uno de estos vecinos se exploran sus respectivos vecinos (vecinos de segundo nivel). El proceso se repite de la misma forma para los distintos niveles, hasta completar el grafo. A diferencia de la *búsqueda en profundidad* (Depth-First Search o DFS), no se empiezan a procesar los vértices de un nivel hasta que no se hayan procesado todos los vértices del nivel anterior.

5.1 Ilustra el funcionamiento del algoritmo resolviendo a mano algunos ejemplos ilustrativos:

- Grafos especiales.
- Caso típico (grafo dirigido ejemplo).
- Caso típico distinto del grafo ejemplo anterior.

5.2 Escribe el pseudocódigo correspondiente al algoritmo BFS.

5.3 Estudia con detalle la siguiente implementación del algoritmo BFS, tomada del libro "*ANSI Common Lisp*" de Paul Graham [http://www.paulgraham.com/acl.html]. Asegúrate de que comprendes su funcionamiento con algún grafo sencillo.

La función *assoc* devuelve la sublista dentro de una lista cuyo *car* sea el elemento que se le pase. Por ejemplo, si llamamos a la lista de arriba *grafo*, (assoc 'b grafo) devolvería la sublista (b d f).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Breadth-first-search in graphs
;;;
(defun bfs (end queue net)
  (if (null queue) '()
    (let* ((path (first queue))
           (node (first path)))
      (if (eql node end)
          (reverse path)
        (bfs end
             (append (rest queue)
                     (new-paths path node net))
             net)))))
(defun new-paths (path node net)
  (mapcar #'(lambda(n)
              (cons n path))
          (rest (assoc node net))))
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

5.4 Pon comentarios en el código anterior, de forma que se ilustre cómo se ha implementado el pseudocódigo propuesto en 4.2).

5.5 Explica por qué esta función resuelve el problema de encontrar el camino más corto entre dos nodos del grafo:

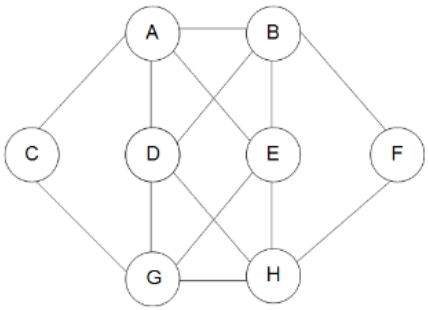
```
(defun shortest-path (start end net)
    (bfs end (list (list start)) net))
```

5.6 Ilustra el funcionamiento del código especificando la secuencia de llamadas a las que da lugar la evaluación:

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

La macro *trace* es especialmente útil para este tipo de tareas. Con *untrace* se desactivan las trazas.

5.7 Utiliza el código anterior para encontrar el camino más corto entre los nodos F y C en el siguiente grafo no dirigido. ¿Cuál es la llamada concreta que tienes que hacer? ¿Qué resultado obtienes con dicha llamada?



5.8 El código anterior falla (entra en una recursión infinita) cuando hay ciclos en el grafo y el problema de búsqueda no tiene solución. Ilustra con un ejemplo este caso problemático y modifica el código para corregir este problema:

```
(defun bfs-improved (end queue net) ...)
(defun shortest-path-improved (end queue net) ...)
```