# Inteligencia Artificial 2017-2018

## Práctica 1: LISP

Celia San Gregorio Moreno

Álvaro Martínez Morales

Grupo: 2213

Fecha: 03/03/2018

Índice

[Inteligencia Artificial 2017-2018 1](#_Toc508217342)

[Práctica 1: LISP 1](#_Toc508217343)

[1. Similitud Coseno 3](#_Toc508217344)

[Apartado 1.1 3](#_Toc508217345)

[Apartado 1.2 3](#_Toc508217346)

[Apartado 1.3 4](#_Toc508217347)

[Apartado 1.4 5](#_Toc508217348)

[2. Raíces de una función 6](#_Toc508217349)

[Apartado 2.1 6](#_Toc508217350)

[Apartado 2.2 6](#_Toc508217351)

[Apartado 2.3 6](#_Toc508217352)

[3. Combinación de listas 8](#_Toc508217353)

[Apartado 3.1 8](#_Toc508217354)

[Apartado 3.2 8](#_Toc508217355)

[Apartado 3.3 8](#_Toc508217356)

[4. Inferencia lógica proposicional 9](#_Toc508217357)

[Apartado 4.1 9](#_Toc508217358)

[Apartado 4.2 11](#_Toc508217359)

[Apartado 4.3 13](#_Toc508217360)

[Apartado 4.4 16](#_Toc508217361)

[Apartado 4.5 18](#_Toc508217362)

[Apartado 4.6 19](#_Toc508217363)

[5. Búsqueda en anchura 20](#_Toc508217364)

[Apartado 5.1 20](#_Toc508217365)

[Apartado 5.2 20](#_Toc508217366)

[Apartado 5.3 20](#_Toc508217367)

[Apartado 5.4 20](#_Toc508217368)

[Apartado 5.5 21](#_Toc508217369)

[Apartado 5.6 21](#_Toc508217370)

[Apartado 5.7 21](#_Toc508217371)

[Apartado 5.8 21](#_Toc508217372)

# 1. Similitud Coseno

## Apartado 1.1

**Similitud coseno de forma recursiva**

Para implementar la similitud coseno entre dos vectores de forma recursiva, primero definimos una función auxiliar prod-escalar-rec (x y).

Esta función comprueba primero si x o y son NIL; si lo son devolverá el valor 0, ya que si uno de los vectores es una lista vacía, ello supone multiplicar por 0 cada uno de los elementos del otro vector.

En caso de que ni x ni y sean NIL, calcula la suma de los productos de los primeros elementos de x e y, y de los productos del resto de los elementos. La función termina cuando llega al final de los vectores.

En cuanto a sc-rec(x y), primero comprueba si los módulos de los vectores (o el producto escalar de sí mismos) es 0. Si lo es, devuelve 0 para evitar un caso de división por 0.

Si el módulo de x e y es distinto de 0, calcula la similitud coseno siguiendo la fórmula:

**Similitud coseno usando mapcar**

Para implementar la similitud coseno entre dos vectores usando mapcar, también definimos una función auxiliar prod-escalar-mapcar (x y).

Esta función realiza el producto sobre cada uno de los elementos de x e y utilizando mapcar, y suma cada producto mediante una llamada a reduce. De igual forma que en prod-escalar-rec (x y), si ambos vectores son NIL devuelve 0.

Por último, sc-mapcar(x y) realiza el mismo cálculo que sc-rec(x y),, pero con llamadas a prod-escalar-rec (x y) para realizar las operaciones. De nuevo, si los módulos de x o y son 0, devolverá 0.

## Apartado 1.2

Para obtener un conjunto de vectores ordenado según su similitud a una categoría dada, sc-conf (cat vs conf) utiliza la función auxiliar crea-vectores-conf (cat vs conf).

Esta función auxilar, para cada vector del conjunto de vectores, comprueba si su similitud coseno con respecto a cat es distinta de NIL y es además mayor que el nivel de confianza conf.

Si la condición se cumple, añade el vector a una lista de vectores cuya similitud coseno es mayor que conf. En caso contrario, ignora el vector y evalúa el resto de vectores del conjunto de manera recursiva hasta que no queden más elementos.

En cuanto a sc-conf (cat vs conf), la función obtiene la lista de vectores con similitud coseno superior a conf y la ordena de mayor a menor similitud.

## Apartado 1.3

Para obtener una lista de pares (<id-categoría> <similitud-coseno>) a partir de un vector de vectores con categorías y otro con textos, la función sc-classifier (cats texts func) se sirve de varias funciones auxiliares:

1. **crea-vectores-similares (cats text func).** A partir de un vector de vectores que representa las distintas categorías, un vector que representa un texto y la función de similitud coseno, crea-vectores-similares (cats text func) va construyendo una lista de listas.

Cada elemento de esa lista de listas es un par (<id-categoría> <similitud-coseno>) que se obtiene al calcular la similitud coseno entre el texto y cada una de las categorías contenidas en cats, todo ello de forma recursiva.

1. **vector-mas-similar-rec (elem vectors).** A partir de un vector de vectores de tipo ((<id-categoría> <sc-1>) (<id-categoría> <sc-2>) ... (<id-categoría> <sc-n>)), devuelve el par (<id-categoría> <sc>) con mayor valor de similitud coseno.

Inicialmente, toma como referencia el primer elemento de vectors (elem); su similitud coseno será la máxima encontrada hasta el momento.

Si al recorrer cada vector de la lista la función encuentra un elemento con mayor similitud coseno que elem, lo tomará como referencia y evaluará el resto de la lista. En caso contrario, ignorará el elemento y seguirá manteniendo elem como referencia.

Todo el recorrido se realiza de forma recursiva.

1. **vector-mas-similar (elem vectors).** Obtiene la lista de pares (<id-categoría> <similitud-coseno>) y devuelve el que mayor similitud coseno tenga.

A partir de estas funciones, sc-classifier (cats texts func) construye de forma recursiva una lista de pares (<id-categoría> <similitud-coseno>) con el mayor valor de similitud coseno que existe entre cada texto y las categorías del vector de vectores cats.

## Apartado 1.4

Realizamos una serie de pruebas con vectores de textos y categorías de distintos tamaños, cuyos resultados están comentados en el código.

Además, medimos los tiempos de ejecución y observamos lo siguiente:

* En todas las llamadas a sc-classifier con #’sc-rec y #’sc-mapcar, la versión mapcar tarda significativamente más ciclos de reloj en completar la operación.
* Cuantos más elementos tenga cada lista de cats y texts, la función recursiva #’sc-rec y sc-classifier tardan menos ciclos de reloj en terminar la operación.
* Por el contrario, si un vector tiene menos elementos, sc-classifier tarda más ciclos de reloj en computar el resultado.

# 2. Raíces de una función

## Apartado 2.1

Para encontrar una raíz en el intervalo [a, b], la función bisect (f a b tol) sigue varios pasos.

En primer lugar, calcula x como el punto medio entre a y b. Después, multiplica los resultados de aplicar el método para encontrar la raíz sobre a y b (es decir, f(a) \* f(b)). Si detecta que el producto es mayor o igual que cero, devuelve NIL (ya que no es seguro que encuentre raíces).

En caso contrario, comprueba si b – a tiene un valor menor que la tolerancia mínima permitida. Si es menor, hemos encontrado la raíz x. Sino, la función multiplicará f(a) \* f(x) para buscar la raíz en [a, x] o [x, b] de forma recursiva, dependiendo del signo de f(a) \* f(x)

## Apartado 2.2

Para calcular la raíz en cada par de intervalos de una lista, allroot (f lst tol) realiza una serie de cálculos.

Primero, comprueba si hay al menos dos números en la lista que pueda tomar como intervalo y así calcular la raíz. Si no hay, devuelve NIL.

Si hay, obtiene los dos primeros elementos de la lista (por ejemplo, lst[i] y lst[i+1]) y realiza el producto ente f(lst[i]) y f(lst[i+1]), siendo f la función para obtener la raíz. Si dicho producto es negativo, hará la bisección en el intervalo [i, i+1].

En caso contrario, se desplazará en la lista de forma recursiva y seguirá buscando intervalos sobre los que obtener raíces. Todas las raíces que encuentre allroot se devolverán en forma de lista.

## Apartado 2.3

Para encontrar todas las raíces de las secciones de un intervalo, la función allind (f a b N tol) se sirve de la función auxiliar make-interval-list (a b i).

Esta función auxiliar crea una lista de secciones que abarcan desde a hasta b, pasando por sucesivos incrementos de (a +i) hasta alcanzar b. Es decir:

[a (a+i ) (a+i+i) (a+i+i+i) ... b]

En cuanto a allind (f a b N tol), la función divide el intervalo [a, b] en 2N secciones mediante la operación (|a| + |b|) / 2N y calcula todas las raíces de las secciones mediante una llamada a allroot.

# 3. Combinación de listas

## Apartado 3.1

Para combinar un átomo elt con cada elemento de una lista y crear un listado de cada combinación, la función combine-elt-lst (elt lst) sigue estos pasos.

Si lst es NIL (ha llegado al final de la lista para combinar), devolverá NIL. En caso contrario, creará una lista de pares (elt <elemento-lista>) de forma recursiva.

## Apartado 3.2

Para crear el producto cartesiano de dos listas, combine-lst-lst (lst1 lst2) comprueba primero si lst1 o lst2 es NIL. Si se cumple la condición, devolverá NIL.

Sino, creará una lista de pares (<elemento-lst1> <elemento-lst2>) con cada elemento de lst1 y la lista lst2, de forma recursiva. La función terminará cuando no haya más elementos de lst1 que combinar.

## Apartado 3.3

Para crear el producto cartesiano de un conjunto de sublistas encapsulado en una lista, primero tomamos el caso base de aplicar un map de la función list sobre la última sublista, lo que hace que ya, de base, devuelve el producto cartesiano de esa única sublista.

Seguidamente, mediante dos funciones recursivas, una anidada en la otra, vamos generando el producto cartesiano de ambas listas, básicamente, uniendo, elemento a elemento de la primera lista, a cada uno de los elementos de la segunda.

# 4. Inferencia lógica proposicional

## Apartado 4.1

Apartado 4.1.1

Para comprobar si un literal es positivo, la función positive-literal-p (x) comprueba que x sea un átomo, nunca NIL, T ni cualquier tipo de conector.

Apartado 4.1.2

Para comprobar si un literal es negativo, la función negative-literal-p (x) comprueba que x sea una lista con un conector ~ y un literal positivo x.

Es decir, (~ x).

Apartado 4.1.3

Para comprobar si el argumento es un literal, la función literal-p (x) comprueba que x sea literal positivo o negativo.

Apartado 4.1.4

Para comprobar si una expresión está en formato prefijo (~ FBF) o (FBF <conector> FBF), la función wff-infix-p (wff) sigue una serie de pasos.

Primero comprueba que wff no sea NIL, ya que NIL no es una FBF en formato prefijo. Si wff no es NIL, o bien es un literal o bien es una lista de estos tipos:

* **Lista (~ FBF).** Si tras FBF no hay más elementos, comprueba si FBF está en formato infijo de forma recursiva.
* **Lista (FBF1 <conector-bicondicional> FBF2).** Si después de FBF2 no hay más elementos, comprueba si FBF1 y FBF2 están en formato prefijo de forma recursiva.
* **Lista (<conector-nario>).** Comprueba si se trata de una conjunción o disyunción vacías (‘(v) o ‘(^)).
* **Lista (FBF1 <conector-nario> FBF2 <conector-nario> ... FBFn).** Si la lista es de tipo (FBF1 <conector-nario> FBF2), comprueba si FBF1 y FBF2 están en formato prefijo de forma recursiva.

Si la lista tiene más FBFs, comprueba si tanto FBF1 como (FBF2 ... FBFn) están en formato prefijo de forma recursiva.

Apartado 4.1.5

Para transformar una FBF en formato infijo a una FBF en formato prefijo, la función infix-to-prefix-p (wff) sigue una serie de pasos.

Mientras que wff sea una FBF en formato infijo, comprueba si es un literal. Si lo es, devuelve su valor.

Si wff tiene más elementos (es decir, es una lista de FBFs unidas por conectores), identifica el tipo de conector que las une.

* **Conector unario ~.** Crea una lista con el conector y la FBF convertida a formato infijo de forma recursiva.
* **Conector binario => o <=>.** Crea una lista de tipo (<conector-binario> FBF1 FBF2).
* **Conector n-ario ^ o v.** Si encuentra una lista de tipo ‘(v) o ‘(^), devuelve a conjunción o disyunción vacías.

Si se trata de una lista de tipo (FBF1 <conector-nario> FBF2 <conector-nario> FBFn), crea una nueva lista con el conector n-ario seguido del resto de FBFs. Todas ellas se han convertido a formato prefijo mediante la función:

(mapcar #'infix-to-prefix (remove (second wff) wff)

La llamada a remove elimina los conectores sobrantes de la lista de FBFs, dando como resultado (FBF2 FBF3 … FBFn), lista que se evaluará elemento a elemento.

Apartado 4.1.5

La función clause-p (wff) determina si una FBF es una cláusula mediante estos criterios:

* El argumento wff es una lista precedida por el conector v.
* La lista contiene una disyunción vacía ‘(v), una disyunción con un literal ‘(v lit) o una disyunción con más de un literal. En este último caso se comprueba si cada elemento es un literal de forma recursiva.

Apartado 4.1.6

La función cnf-p (wff) determina si una FBF es una FNC mediante estos criterios:

* El argumento wff es una lista precedida por el conector ^.
* La lista contiene una disyunción vacía ‘(^), una conjunción de tipo ‘(^ (v)) o ‘(^ (v lit1 lit2 ... litn) , o una conjunción con más de una cláusula. En este último caso se comprueba si cada elemento es una cláusula de forma recursiva.

## Apartado 4.2

Apartado 4.2.1

Para convertir la FBF pasada como argumento a una FBF equivalente sin el conector <=>, la función eliminate-biconditional (wff) sigue una serie de pasos.

En primer lugar, si wff es NIL o un literal devuelve su valor. Si se trata de una lista, tendrá la estructura (<conector> FBF1 FBF2).

Si el conector es bicondicional, trasforma FBF1 y FBF2 a FBFs equivalentes sin <=> de forma recursiva, y crea una lista de tipo (^ (=> FBF1 FBF2) (=> FBF2 FBF1)).

Si el conector es unario o enario, devuelve la misma estructura de FBFs (~ FBF) o (FBF1 <conector-nario> FBF2 <conector-nario> ... FBFn), sin el conector <=>. Cada FBF se evalúa con la función mapcar.

Apartado 4.2.2

Para convertir la FBF pasada como argumento a una FBF equivalente sin el conector =>, la función eliminate-conditional (wff) sigue una serie de pasos.

Si wff es NIL o un literal, devuelve su valor. Si se trata de una lista, tendrá la estructura (<conector> FBF1 FBF2).

Si el conector es condicional, trasforma FBF1 y FBF2 a FBFs equivalentes sin => de forma recursiva, y crea una lista de tipo (v (~ FBF1) FBF2).

Si el conector es unario o enario, devuelve la misma estructura de FBFs (~ FBF) o (FBF1 <conector-nario> FBF2 <conector-nario> ... FBFn), sin el conector =>. Cada FBF se evalúa con la función mapcar.

Apartado 4.2.3

La función reduce-scope-of-negation (wff) convierte una FBF sin conectores bicondicionales a una FBF equivalente donde la negación se realiza a nivel de literal.

Primero, la función comprueba si wff es NIL o un literal. Si lo es, devuelve su valor.

En caso de que wff sea una lista, la función considerará los siguientes casos:

* **Lista de tipo (~ (<conector-nario> FBFs).** Niega cada FBF de la lista mediante la función:

(mapcar #'(lambda(x) (list +not+ x)) (rest elems))

Y construye una lista con estos elementos:

1. Conector n-ario ^ cambiado por v (o al revés) llamando a la función auxiliar exchange-and-or (connector).
2. FBFs negadas previamente con el ámbito de negación reducido. El ámbito de la negación de cada una de ellas se reduce mediante la función mapcar.

* **Lista de tipo (~ (~ FBF)).** Devuelve la FBF con el ámbito de negación reducido. Para ello utiliza una llamada recursiva a reduce-scope-of-negation.
* **Lista de tipo (<conector-nario> FBFs).** Crea una lista con el conector n-ario y cada una de las FBFs reducidas en ámbito de negación mediante la función mapcar.

Apartado 4.2.5

Para eliminar los conectores de una FNC, la función eliminate-connectors (cnf) comprueba primero si la FNC pasada como argumento es un literal. En ese caso devolverá su valor.

Sino, cada elemento de la FNC que contenga un conector n-ario se evaluará mediante la función:

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

Y se convertirá a una lista de literales sin conectores.

Apartado 4.2.6

Para convertir una FBF en formato infijo a una FNC en formato prefijo que no tenga conectores, la función wff-infix-to-cnf (wff) realiza llamadas a funciones auxiliares que van transformando la FNC paso a paso como se indica en los apartados anteriores.

## Apartado 4.3

Apartado 4.3.1

La función eliminate-repeated-literals (k) transforma la cláusula K a una cláusula equivalente sin literales repetidos siguiendo una serie de pasos.

En primer lugar, comprueba si K sólo tiene un elemento. Si se cumple esta condición, devuelve el literal de K (ya sea positivo o negativo). En cambio, si K tiene más de un elemento, la función comprueba cuántas ocurrencias de este elemento encuentra en K.

* **Sólo hay una ocurrencia.** Crea una lista con el único literal y el resto de los elementos de K, filtrados de forma recursiva.
* **Hay más de una ocurrencia.** El elemento se ignora y se evalúa el resto de los literales de K de forma recursiva.

Apartado 4.3.2

Para eliminar las cláusulas repetidas de una FNC, la función eliminate-repeated-clauses (cnf) elimina los literales repetidos de cada cláusula de la FNC y utiliza una serie de funciones auxiliares.

1. **check-if-equal (cls target-cls).** Comprueba si dos cláusulas son iguales, o bien mediante equal (misma representación textual) o bien porque la diferencia de conjuntos entre cls y target-cls (y viceversa) es NIL.

En este último caso, el valor NIL en las operaciones set-difference indica que ambas cláusulas son iguales, aunque se representen de forma distinta (por ejemplo, ‘(a b c) y ‘( b c a)).

1. **check-if-repeated-in-cnf (clause cnf).** Comprueba si la cláusula clause está repetida en la FNC pasada como argumento.

Para ello, comprueba si clause es igual que alguno de los elementos de la FNC de forma recursiva. Si la función llega al final de la FNC (null cnf), no hay coincidencias.

1. **get-cnf-without-repeated-clauses (cnf).** Obtiene una FNC sin cláusulas repetidas de forma recursiva.

Si la función llega al final de la FNC, no hay más cláusulas que evaluar y devuelve NIL. Sino, comprueba si la cláusula tiene repeticiones en la FNC.

* Tiene repeticiones. La función ignora la cláusula repetida y evalúa el resto de los elementos de la FNC.
* No tiene repeticiones. La función construye una lista con la cláusula única y el resto de los elementos de la FNC, con sus duplicados eliminados de forma recursiva.

Apartado 4.3.3

Para comprobar si la cláusula K1 subsume a K2, la función subsume (K1 K2) comprueba si K1 es un subconjunto de K2 (es decir, si todos los literales de K1 se encuentran en K2).

Si se cumple la condición, devuelve una lista con K1 como elemento.

Apartado 4.3.4

Para eliminar las clausulas subumidas de una fcn utilizamos una serie de funciones auxiliares para, fundamentalmente, recrear una lista que sólo contiene los elementos no subsumidos en ningún otro elemento de la clausa.

En esta tarea, nos servimos de las siguientes funciones auxiliares:

1. **eliminate-subsumed-clauses (cnf cnf-original)** Para aplicar, de forma transparente, la recursividad sobre nuestra función. Requiere de mantener a lo largo de las llamadas recursivas la lista de clausulas original de cara a que se pueda aplicar la intersección para unir los resultados de funciones auxiliares más internas.

Así mismo, dada la naturaleza de los bucles internos, como se explicará en unos instantes, que devuelven todas las cápsulas que no tienen la primera clausula de la iteración subsumida, excluyendo así, por supuesto, a la primera clausula en dicha devolución. La adicción de esta al resultado final dependerá de otra función, muy similar a la previamente mencionada para reformar formar la lista sin clausulas subsumidas respecto a la primera clausula de la iteración, sólo que esta simplemente devolverá T si esta primera clausula está subsumida en alguna otra o nil en caso contrario. Así, si recibe un nil, realizará un append de esta primera clausula sobre la lista devuelta por la función interna para mantener el elemento o, en caso contrario, no lo hará para eliminarlo.

1. **first-needs-removal? (first-fbf rest-cnf)** Función auxiliar que, mediante la recursividad de la función *or*, discierne si la clausula first-fbf está subsumida en alguna otra clausula de rest -cnf y, por tanto, debe ser ‘eliminada’, devolviendo T en ese caso o nil en el contrario.
2. **remove-subsumed-clauses (first-fbf rest-cnf)** Función auxiliar que reforma las listas ‘eliminando’ las clausulas que tienen la clausula first-fbf subsumida, iterando una de las clausulas de rest-cnf en cada iteración y sólo añadiendo a la lista reformada las clausulas que pasan el criterio.

Apartado 4.3.5

Para comprobar si la cláusula K es una tautología (es decir, si contiene los literales p y (~ p)), la función tautology-p (K) se sirve de varias funciones auxiliares.

1. **literals-with-same-component (lit1 lit2).** Comprueba si lit1 y lit2 (literales o bien en forma positiva y negativa, o bien en forma negativa y positiva) tienen el mismo literal. Por ejemplo, ‘p y ‘(~ p) tendrían el mismo componente p.

Para ello, comprueba si lit1 es un literal positivo. Si lo es, comprueba si (~ lit1) es igual que lit2, literal negativo pasado como argumento.

En caso de que lit1 sea negativo (~ lit1), comprueba si lit1 es igual que lit2, literal positivo pasado como argumento.

Si lit1 y lit2 no tienen el mismo componente, la función devolverá NIL.

1. **check-if-pos-and-neg (lit K).** Comprueba si en la cláusula K hay un literal positivo lit y otro negativo (~ lit). Para ello, recorre cada literal de K.

* Si el literal es positivo, comprueba si existe otro negativo con el mismo componente.
* Si el literal es negativo, comprueba si existe otro positivo con el mismo componente.

Si la función llega al final de K, ello implica que no ha encontrado dos literales con el mismo componente.

A partir de estas funciones, tautology-p (K) comprueba si entre alguno de sus literales hay uno positivo y otro negativo con el mismo componente. Si detecta una cláusula vacía o con un único elemento, devolverá NIL.

Apartado 4.3.6

Para eliminar las cláusulas de una FNC que son tautología, eliminate-tautologies (cnf) comprueba, por cada cláusula, si alguna es tautología.

Si lo es, ignora la cláusula y evalúa el resto de elementos de la FNC de forma recursiva. En caso contrario, construye una lista con la cláusula que no es tautología y el resto de cláusulas de la FNC, también evaluadas de forma recursiva.

Apartado 4.3.7

Para simplificar una FNC, la función simplify-cnf (cnf) realiza llamadas a funciones auxiliares que van aplicando la eliminación de literales y cláusulas repetidos, la eliminación de tautologías y la eliminación de cláusulas subsumidas como se indica en los apartados anteriores.

## Apartado 4.4

Apartado 4.4.1

Para extraer el conjunto de cláusulas lambda-neutras () de una FNC, extract-neutral-clauses (lambda cnf) se sirve de la función auxiliar contains-pos-or-neg-literal (lit clause).

Esta función auxiliar comprueba si el literal lit se encuentra en forma positiva o negativa (~ lit) dentro de la cláusula pasada como argumento. Si la cláusula es NIL, la función devolverá NIL (el literal no está).

En cuanto a extract-neutral-clauses (lambda cnf), la función comprueba, cláusula a cláusula, si alguna de ellas tiene algún literal λ positivo o negativo. Las que contengan estos literales se ignorarán, mientras que las que no cumplan la condición se devolverán como una lista de cláusulas neutras.

Apartado 4.4.2

Para extraer el conjunto de cláusulas lambda-positivas () de una FNC, extract-positive-clauses (lambda cnf) se sirve de la función auxiliar contains-positive-literal (lit clause).

Esta función auxiliar comprueba si el literal positivo lit se encuentra dentro de la cláusula pasada como argumento. Si la cláusula es NIL, la función devolverá NIL (el literal no está).

En cuanto a extract-positive-clauses (lambda cnf), la función comprueba, cláusula a cláusula, si alguna de ellas tiene algún literal λ positivo. Todas las cláusulas que contengan estos literales se devolverán como una lista de cláusulas positivas; el resto se ignorará.

Apartado 4.4.3

Para extraer el conjunto de cláusulas lambda-negativas () de una FNC, extract-negative-clauses (lambda cnf) se sirve de la función auxiliar contains-negative-literal (lit clause).

Esta función auxiliar comprueba si el literal negativo (~ lit) se encuentra dentro de la cláusula pasada como argumento. Si la cláusula es NIL, la función devolverá NIL (el literal no está).

En cuanto a extract-negative-clauses (lambda cnf), la función comprueba, cláusula a cláusula, si alguna de ellas tiene algún literal negativo (~ λ). Todas las cláusulas que contengan estos literales se devolverán como una lista de cláusulas negativas; el resto se ignorará.

Apartado 4.4.4

Para calcular el resolvente entre dos cláusulas , la función resolve-on (lambda K1 K2) utiliza dos funciones auxiliares.

1. **resolvable-clauses-p (lambda K1 K2).** Comprueba si K1 y K2 se pueden resolver. Para ello, K1 debe contener un literal positivo λ y K2 un literal negativo (~ λ), o viceversa.
2. **get-resolved-clause (lambda K).** Devuelve la cláusula K resuelta sobre λ.

Si K contiene al literal positivo λ, se eliminarán todas sus ocurrencias de K. Por otro lado, si K contiene al literal negativo (~ λ), también se eliminarán todas sus ocurrencias de K.

La función resolve-on (lambda K1 K2) comprueba primero si K1 o K2 son NIL. En ese caso, no hay nada que resolver.

Si K1 y K2 son distintas de NIL y además son resolubles, la función obtiene las cláusulas resueltas. Si en el proceso de resolución de K1 y K2 la función get-resolved-clause (lambda K) devuelve NIL (porque K1 y K2 tenían sólo un literal λ), la función devolverá (NIL).

En caso contrario, devolverá o bien , o bien ; es decir, la unión de las cláusulas resueltas sobre λ.

Apartado 4.4.5

Para calcular el conjunto de cláusulas para una FNC α, la función build-RES (lambda cnf) utiliza dos funciones auxiliares.

1. **get-list-of-resolved-clauses (lambda K clause-set).** Devuelve una lista de resolventes , siendo K la cláusula pasada como argumento y Ki cada una de las cláusulas del conjunto clause-set. Todas las resoluciones son sobre λ.

Para ello, resuelve K y la primera cláusula del conjunto clause-set s sobre λ. Si obtiene la cláusula vacía (NIL), devuelve dicha cláusula (no hay más cláusulas que resolver).

Por el contrario, si la cláusula resuelta no es (NIL), construye una lista con esta cláusula y el resto de resoluciones entre K y cada cláusula de clause-set. Todo ello se realiza de forma recursiva.

1. **get-all-resolved-clauses (lambda positive-clauses negative-clauses).** Devuelve una lista de resolventes entre cada una de las cláusulas del conjunto () y las cláusulas del conjunto ().

Para ello, obtiene de forma recursiva la lista de cláusulas resueltas entre cada cláusula del conjunto lambda-positivo y las cláusulas del conjunto lambda-negativo.

Una vez obtenidos los distintos conjuntos de cláusulas resueltas por cada cláusula del conjunto (), realiza una unión sobre ellos y elimina las posibles cláusulas repetidas.

Apoyándose en estas funciones, build-RES (lambda cnf) extrae los conjuntos de cláusulas lambda-positivos, lambda-neutros y lambda-negativos, y realiza la unión entre las cláusulas del conjunto () y las cláusulas resueltas entre () y () obtenidas a partir de get-all-resolved-clauses (lambda positive-clauses negative-clauses).

## Apartado 4.5

Para discernir si una cnf es SAT o UNSAT nos hemos enfrentado a un único problema, que es es de formar una lista con todos los literales (en estado positivo) presentes en una función.

Esto se ha llevado a cabo mediante la función auxiliar **make-literal-list**, a la cual, al introducir una cnf, primero reduce mediante la función *union*, lo cual te devuelve una lista con todos los literales positivos y negativos presentes en la lista. Seguidamente, aplicamos la función auxiliar **happiness,** que, a su vez, mediante la aplicación recursiva de la función **make-positive**, devuelve la lista con todos sus literales en estado positivo. Por último, mediante la función previamente codificada de **eliminate-repeated-literals**, la lista se queda con una única aparición de cada uno de sus clausulas como una clausula positiva.

Una vez llegados a este punto, simplemente hubo que reflejar los casos particulares de una cnf sin clausulas (en cuyo caso devuelve directamente *T*, dado que la cnf sería SAT) y el caso de que hubiera alguna clausula nula (comprobado mediante la función auxiliar **check-4-empty-clause** y devolviendo *nil* en dicho caso, dado que la cnf sería UNSAT ) y aplicar de manera recursiva la función de resolución con cada una de las clausulas previamente obtenidas sobre la cnf, devolviendo *T* si se resuelve (sólo queda nilen la cnf) o *nil* en caso contrario.

## Apartado 4.6

Este último apartado del ejercicio 4 ha sido implementado siguiendo la fórmula propuesta en el guión de la práctica, pasando tando *wff* y ¬*w* (este último generado a partir de listar *+not+* y *w)* a FNC mediante la previamente implementada **wwf-infix-to-cnf** , uniéndolas mediante *union* y resolviendo mediante **RES-SAT-p**, realizada en el apartado enterior.

Si esto resulta *nil*, quiere decir que *w* es consecuencia lógica de *wff*, y por tanto, la función devolverá *T*. En el caso contrario, devolverá *nil*, dado que habremos discernido que no es consecuencia lógica.

Cabe destacar como reflexión que, aunque este apartado, en si, es terriblemente simple en comparación con el resto de apartados de este ejercicio, ha sacado a luz una serie de errores que habíamos arrastrado a lo largo de los apartados previos que no habían salido a luz en los casos de prueba propuesto, lo que ha derivado en que hayamos tenido que dedicar una mayor cantidad de tiempo de la que cabría esperar, pero ha mejorado en buena medida la calidad de nuestro código.

# 5. Búsqueda en anchura

## Apartado 5.1

Nada que reportar.

## Apartado 5.2

BFS(grafo, camino\_recorrido, nodo\_actual, nodo\_final)

if nodo\_actual == nodo final:

return camino\_recorrido + nodo actual;

else :

adyacentes = get\_nodos\_adyacentes(grafo, nodo\_actual);

for each nuevo\_nodo in adyacentes:

if no\_existe(nuevo\_nodo, camino\_recorrido)

BFS(grafo, camino\_recorrido+nodo\_actual, nuevo nodo, nodo final)

## Apartado 5.3

Nada que reportar.

## Apartado 5.4

Los comentarios han sido añadidos al código del ejercicio, presente en el fichero de código de la práctica.

Resumiendo, el código funciona mediante una cola en la que guarda cada camino realizado, siendo el primer nodo de estos el último nodo explorado del camino, y del cual se valorarán nuevos caminos, añadiendo a la cola ese camino con, ahora, el primer elemento siendo un nuevo nodo alcanzable desde el último nodo previo, una vez por cada elemento descubrible, a la cola.

Cabe añadir que, dada esta disposición de nodos en los caminos, antes de devolverlo se aplica un *reserse* sobre el camino para que vaya de nodo inicial a nodo objetivo.

## Apartado 5.5

La función ha sido añadida al fichero de código y comentada en consecuencia.

Básicamente encuentra el mejor camino dado que, al comenzar el algoritmo, únicamente metes en la cola el nodo del comienzo y, dada la propia naturaleza del algoritmo de búsqueda por anchura, su ejecución siempre devolverá uno de los caminos más cortos (o el más corto si no hay otro similar) de los nodos que introduzcas en su primera ejecución hasta el nodo final.

## Apartado 5.6

(shortest-path ’a ’f ’((a d) (b d f) (c e) (d f) (e b f) (f)))

1. bfs( f ((a)) ((a d) (b d f) (c e) (d f) (e b f) (f))
2. bfs( f ((d a)) ((a d) (b d f) (c e) (d f) (e b f) (f))
3. bfs( f ((f d a)) ((a d) (b d f) (c e) (d f) (e b f) (f))
4. bfs( f ((f d a)) ((a d) (b d f) (c e) (d f) (e b f) (f))
5. (a d f)

## Apartado 5.7

La llamada que hay que hacer incluye la correcta representación del grafo como una lista de nodos y sus caminos y la señalización del nodo inicial y el final.

(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)))

## Apartado 5.8

Añadiendo un simple contador de iteraciones, integrado en **bfs-ls** (loop safe), se impide, con un gasto extra mínimo, tanto de memoria como de ejecución, que la función se ejecute indefinidamente hasta el stack overflow de llamada a funciones por no haber un camino válido hasta el nodo final y, sin embargo, sí haber un circuito en el camino del nodo inicial.

Como comentario, este último contador podría nombrase como la heurística del problema; el punto a partir del cual el problema no puede alcanzar una solución que se pueda considerar aceptable.