***Inteligencia Artificial: Práctica 1***

***Ejercicio 1:***

En el apartado **1.1**, hemos creado tres funciones para calcular la distancia coseno:

1. Producto-escalar que calcula el producto escalar de dos vectores:

Entrada: dos vectores x e y

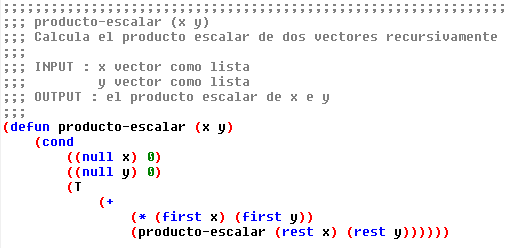
Salida: producto escalar de x e y

Producto-escalar (x, y)

Si x o y son vectores nulos

Devuelve 0

En otro caso

 Devuelve para n = tamaño de vector

1. Suma-cuadrados que calcula la norma al cuadrado de un vector:

Entrada: un vector x

Salida: la norma de x al cuadrado

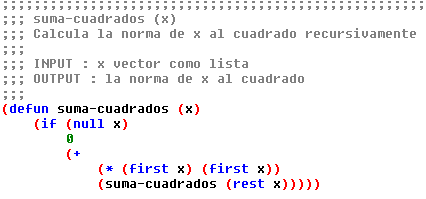
Suma-cuadrados (x)

Si x es nulo

Devuelve 0

En otro caso

Devuelve para n = tamaño de x



1. Cosine-distance-rec que calcula la distancia coseno de dos vectores:

Entrada: dos vectores x e y

Salida: la distancia coseno de x e y

Cosine-distance-rec (x, y)

Sx = suma-cuadrados (x)

Sy = suma-cuadrados (y)

Si sx = 0 y sy = 0

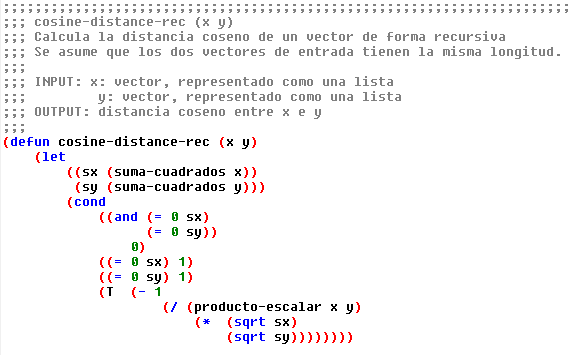
Devuelve 0

Si sx = 0 ò sy = 0

Devuelve 1

En otro caso

Devuelve 1 -



Al ejecutar el código recursivo nos ha quedado lo siguiente:

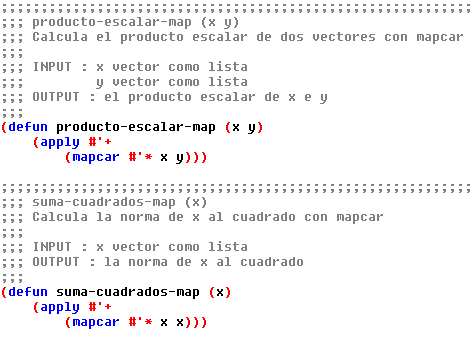
*1. (cosine-distance-rec ’(1 2) ’(1 2 3)) = 0.40238577*

*2. (cosine-distance-rec nil ’(1 2 3)) = 1*

*3. (cosine-distance-rec ’() ’()) = 0*

*4. (cosine-distance-rec ’(0 0) ’(0 0)) = 0*

Al ejecutar el mismo código con las funciones para mapcar nos dan los mismos resultados. Hemos decidido que cuando dos listas estén vacías el resultado sea 0 pues la distancia entre dos vectores vacíos debe ser nula, así como para dos vectores nulos. En caso de que solamente una de las dos listas esté vacía devuelve un 1. Los pseudocódigos son exactamente iguales, y las funciones nos quedan así:



En el apartado **1.2** nos sale lo siguiente:

*1. (order-vectors-cosine-distance ’(1 2 3) ’()) = NIL*

*2. (order-vectors-cosine-distance ’() ’((4 3 2) (1 2 3))) = NIL*

Elaboramos primeramente una función que estudia si los vectores que le pasamos tienen una semejanza (1 – distancia-coseno) mayor al nivel de confianza dado. Seguidamente, la función principal ordena esta lista usando sort, según la distancia coseno.

En el apartado **1.3**  diseñamos una función que ordena las categorías dadas según la semejanza al texto dado. Finalmente, la función principal cogerá para cada texto, usando la anterior función, la categoría que le corresponda (la primera de la lista de categorías) y devolverá la lista de pares (categoría, distancia-coseno)

Probamos esta función con distintos valores en el apartado **1.4** y obtenemos:

1. *(get-vectors-category '(()) '(()) #'cosine-distance-rec) ;;; --> NIL*
2. *(get-vectors-category '(()) '(()) #'cosine-distance-mapcar) ;;; --> NIL*
3. *(get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-rec) ;;; --> ((2 0.40238577))*
4. *(get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-mapcar) ;;; --> ((2 0.40238577))*
5. *(get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance-rec) ;;; --> NIL*
6. *(get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance-mapcar) ;;; --> NIL*

***Ejercicio 2:***

Para la resolución del apartado ***2.1*** básicamente hicimos las funciones siguientes:

1. La función *es-raíz* nos devuelve si |x| < tol, la utilizamos pasándole el valor de f(x) para que nos devuelva True si es x es raíz de f bajo una cierta tolerancia.

Entrada: x un valor, tol una tolerancia.

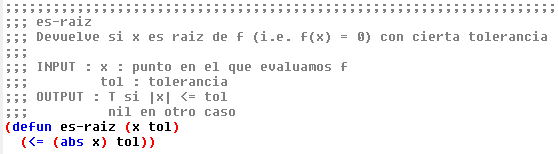
Salida: True si el valor absoluto de x es menor o igual que la tolerancia.

Otro caso devuelve False

Es-raíz (x, tol)

Si |x| <= tol devuelve True

Sino devuelve False



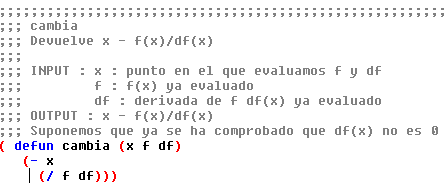
1. La función *cambia* simplemente realiza el cambio x = x – f(x)/df(x) para la llamada recursiva sin que el código quede tan sucio. Es la función superior la que se encarga de que df(x) sea distinto de 0 pues no podemos dividir por 0.

Entrada: x un valor, f una función, df la derivada de f

Salida: x = x – f(x)/df(x)

Cambia(x f df):

Devuelve x – f(x)/df(x).



1. La función newton pedida que encuentra una raíz de una función con el método de Newton de una manera recursiva.

Entrada: f función cuyo 0 queremos encontrar

Df derivada de f

Max-iter número máximo de iteraciones

X0 estimación inicial del cero

Tol tolerancia para la convergencia

Salida: estimación del 0 o nil si diverge

Newton (f df max-iter x0 tol)

Fx = f(x0)

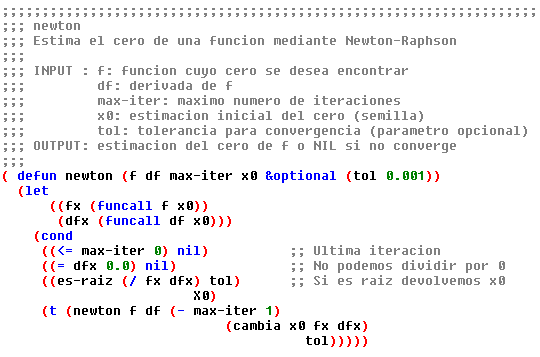
Dfx = df(x0)

Si max-iter <= 0 devuelve nil, diverge

Si dfx = 0 devuelve nil, no podemos dividir por 0

Si fx/dfx es raíz devuelve x0

Otro caso devuelve newton (f df (max-iter)-1 cambia(x0) tol)



Hemos decidido que cuando la derivada sea 0, como no podemos dividir por 0, la función devuelve *nil*. Lo mismo ocurre cuando no converge y no encuentra solución, es decir, las iteraciones llegan a 0.

Los resultados de los ejemplos nos dan lo siguiente:

1. *(newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 20 3.0) ;;---> 4.000084*

1. *(newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 20 0.6) ;;---> 0.99999946*

1. *(newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 30 -2.5) ;;---> -3.0000203*

1. *(newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 10 100) ;;---> NIL*

Podemos observar que los resultados no son exactos, pero se puede utilizar una función para redondearlos. Esto ocurre por la tolerancia.

En el apartado **2.2** hemos realizado la siguiente función:

Entrada: f, df, max-iter, tol como antes.

Semillas: una lista de semillas a probar

Salida: primer resultado que se acerque a un 0 a partir de la lista de semillas

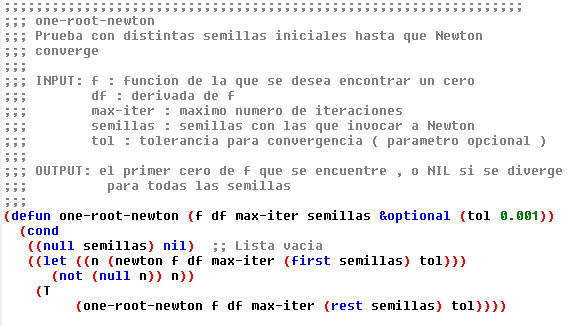
One-root-newton (f df max-iter semillas tol)

Si semillas está vacía devuelve nil

Sea n = newton con la primera semilla de la lista

Si n converge a una solución devuelve la solución

Sino hace recursivamente one-root-newton con (rest semillas)



Podemos observar que primero comprueba si la lista *semillas* es *nil,* devolviendo *nil* en caso afirmativo; sino, realiza la función *newton* del apartado anterior con el primer elemento de la lista. Si da *nil*, es decir, no encuentra resultado para esa semilla, realiza la misma función, pero con *(rest semillas)* de modo que hace recursivamente una búsqueda de una posible solución entre la lista de semillas hasta encontrar una o hasta que se acabe la lista. Los resultados obtenidos son los siguientes:

1. *(one-root-newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5)) ;;---> 0.99999946*

1. *(one-root-newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 20 '(3.0 -2.5)) ;;---> 4.000084*

1. *(one-root-newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 1 '(3.0 -2.5)) ;;---> NIL*

Se puede volver a observar que los resultados no están redondeados, pero vuelve a ser por la función *newton* y la tolerancia que utilizamos para encontrar las raíces.

Finalmente, en el apartado **2.3** hemos implementado la siguiente función, la cual comprueba que la lista *semillas* no esté vacía, en cuyo caso devuelve *nil*. A continuación, si la lista no estaba vacía, realizamos recursivamente la función del apartado anterior con *cons* para que nos devuelva una lista de soluciones.

Entrada : f, df, max-iter como antes

Semillas es una lista de semillas

Salida : una lista de soluciones para cada semilla de la lista semillas

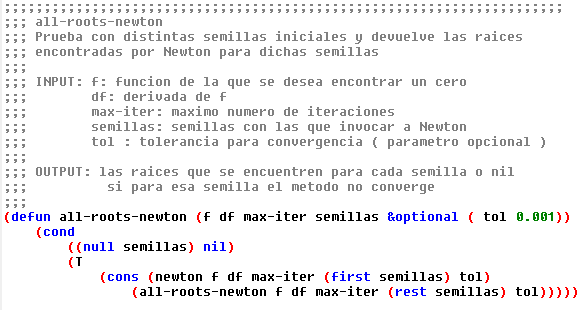
All-roots-newton(f df max-iter semillas tol)

Si semillas está vacía devolvemos nil

Sino:

Calcula newton de la primera semilla (llamémoslo n)

Concatena n con all-roots-newton de (rest semillas)



1. *(all-roots-newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5))*

*;;---> (0.99999946 4.000084 -3.0000203)*

1. *(all-roots-newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 20 '(0.6 3.0 10000))*

*;; ---> (0.99999946 4.000084 nil)*

1. *(all-roots-newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 1 '(0.6 3.0 -2.5)) ;;---> (nil nil nil)*

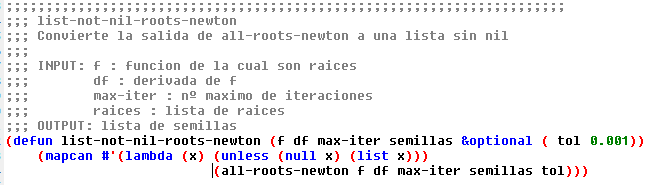
En el apartado **2.3.1** implementamos la siguiente función para eliminar los nil. En ella básicamente hacemos un (unless (null x) (list x)) para que sólo meta en la lista los que no son nil.

Entrada: f, df, max-iter, semillas y tol como antes.

Salida: una lista de soluciones para cada semilla de la lista semillas pero que no incluye los casos en los que diverge (los nil).

List-not-nil-roots-newton (f df max-iter semillas tol):

Para cada resultado de all-roots-newton los mete en una lista a menos que sea nil.



1. *(list-not-nil-roots-newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 20 '(0.6 3.0 -2.5))*

*;;---> (0.99999946 4.000084 -3.0000203)*

1. *(list-not-nil-roots-newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 20 '(0.6 3.0 10000))*

*;; ---> (0.99999946 4.000084)*

1. *(list-not-nil-roots-newton #'(lambda(x) (\* (- x 4) (- x 1) (+ x 3)))*

*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 1 '(0.6 3.0 -2.5)) ;;---> nil*

***Ejercicio 3:***

En el apartado **3.1** teníamos que combinar un elemento con una lista, dando lugar a los distintos pares con el siguiente código:

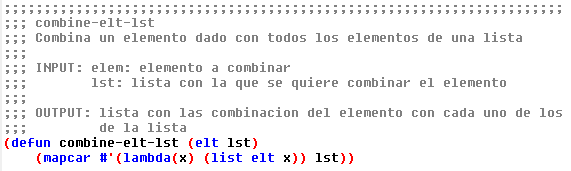
Entrada: elt elemento cualquiera

Lst lista de elementos

Salida: una combinación de elt con cada elemento de lst

Combine-elt-lst (elt lst)

Para cada elemento de lst hace una lista de él y de elt



Podemos observar que nos dan los siguientes resultados:

1. *(combine-elt-lst 'a '(1 2 3)) -> ((A 1) (A 2) (A 3))*
2. *(combine-elt-lst 'a nil) -> nil*
3. *(combine-elt-lst nil nil) -> nil*
4. *(combine-elt-lst nil '(a b) -> ((nil a) (nil b))*

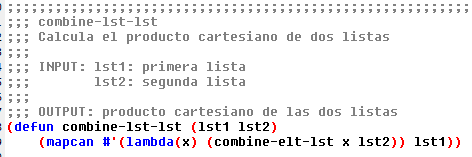
A continuación, en el apartado **3.2 s**e pedía combinar dos listas, para ello hemos utilizado la función combine-elt-lst anterior recursivamente:

Entrada: lst1 y lst2 que son dos listas de elementos

Salida: producto cartesiano de las dos listas (i.e. combinación de todos los elementos de ambas listas)

Combine-lst-lst (lst1 lst2)

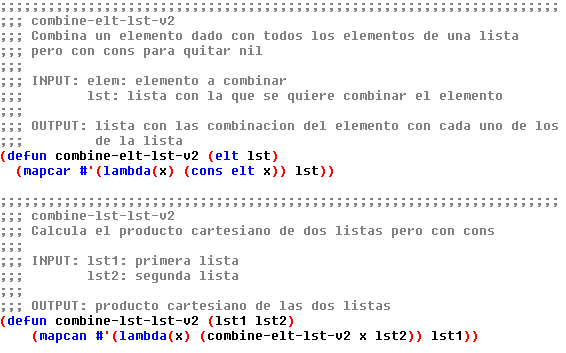
Para cada elemento de lst1 hace combine-elt-lst de él y lst2



Dándonos los siguientes resultados:

1. *(combine-lst-lst '(a b c) '(1 2)) -> ((A 1) (A 2) (B 1) (B 2) (C 1) (C 2))*
2. *(combine-lst-lst nil nil) -> nil*
3. *(combine-lst-lst '(a b c) nil) -> nil*
4. *(combine-lst-lst nil '(a b c)) -> nil*

Finalmente, en el apartado **3.3** había que combinar n listas. Puesto que *list* no excluye el caso *nil*, hemos creado dos funciones nuevas que utilizan *cons* para evitar que *nil* aparezca en las listas. El pseudocódigo es el mismo que los anteriores.



Son básicamente iguales que las iniciales y con ellas hemos creado la siguiente, la cual combina un número arbitrario de listas:

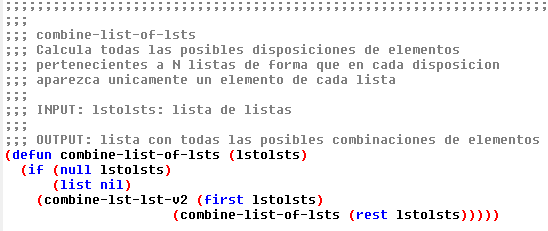
Entrada: lstolsts que es una lista de listas

Salida: lista con todas las combinaciones de las listas anteriores

Combine-list-of-lsts(lstolsts)

Si lstolsts está vacía devuelve nil listado (lidt nil)

Sino devuelve combine-lst-lst-v2 de la primera lista de listolsts y de combine-list-of-lsts del resto de lstolsts, es decir, combina todas las listas menos la primera, y luego combina la primera lista con el resultado que de lo anterior.



Los resultados obtenidos son:

1. *(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))*

1. *(combine-list-of-lsts '(() (+ -) (1 2 3 4))) ;; -> nil*
2. *(combine-list-of-lsts '((a b c) () (1 2 3 4))) ;; -> nil*
3. *(combine-list-of-lsts '((a b c) (1 2 3 4) ())) ;; -> nil*
4. *(combine-list-of-lsts '((1 2 3 4))) ;; -> ((1) (2) (3) (4))*
5. *(combine-list-of-lsts '(nil)) ;; -> nil*
6. *(combine-list-of-lsts nil) ;; -> (nil)*

***Ejercicio 4:***

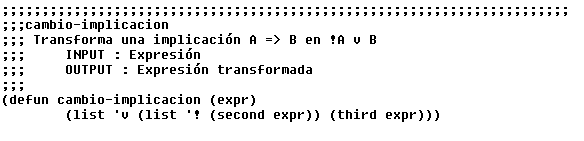
El apartado **4.1** nos pide que desarrollemos una serie de funciones necesarias para el desarrollo del árbol de verdad. Hemos implementado las siguientes:

1. cambio-implicación

Entrada: una expresión de la forma (=> A B)

Salida: la expresión de la forma (v (! A) B)

cambio-implicacion (expr)

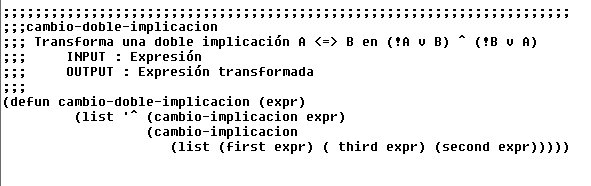
Devuelve una lista cuyo primer elemento es v, el segundo es la lista ! + segundo elemento de expr, y el tercero es el tercer elemento de expr

1. cambio-doble-implicacion

Entrada: una expresión de la forma (<=> A B)

Salida: la expresión de la forma (^ (v (! A) B) (v (! B) A ))

cambio-doble-implicacion (expr)

Devuelve una lista cuyo primer elemento es ^, el segundo es el resultado de llamar a cambio-implicacion(expr) y el tercero el resultado de llamar a cambio-implicacion, pasándole esta vez como argumento una expresión resultante de permutar el segundo elemento de expr por el tercero.

1. construye-lista-negada

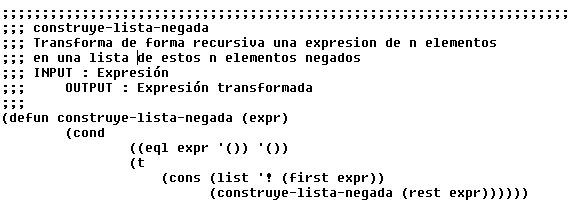
Entrada: una lista de la forma ( A B C …)

Salida: la lista de la forma ((! A) (! B) (! C)….

construye-lista-negada(expr)

Si la expresión es una lista vacia, devuelve la lista vacia

Sino, hace el cons de la lista (! (first expr)) con la llamada a construye-lista-negada(rest expr)



1. cambio-negacion

Entrada: una expresión de la forma (!(! A)) o (! (^ A B C …)) o (! (v A B C …))

o (! (=> A B)) o (! (<=> A B))

Salida: Para cada expresión , su transformación correspondiente, eliminando las negaciones

cambio-negacion(expr)

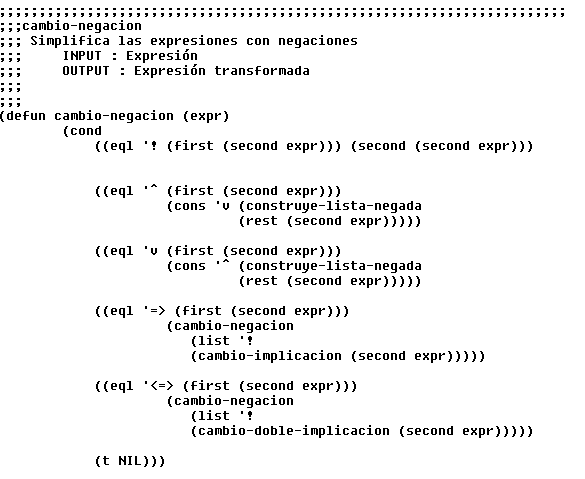
Si es una doble negación, se devuelve el elemento sin negaciones.

Si es una negación de un and, se devuelve la lista de v con la llamada a construye-lista-negada sobre los elementos del and.

Si es una negación de un or, se devuelve la lista de ^ con la llamada a construye-lista-negada sobre los elementos del or.

Si es una negación de una implicación, se llama a la función cambio-negación sobre la lista generada al llamar a cambio-implicacion(expr).

Si es una negación de una doble implicación, se llama a la función cambio-negación sobre la lista generada al llamar a cambio-doble-implicacion(expr).

En cualquier otro caso devuelve NIL.

1. transforma

Entrada: una fbf

Salida: la fbf sin las implicaciones, dobles implicaciones y con las negaciones desarrolladas correctamente

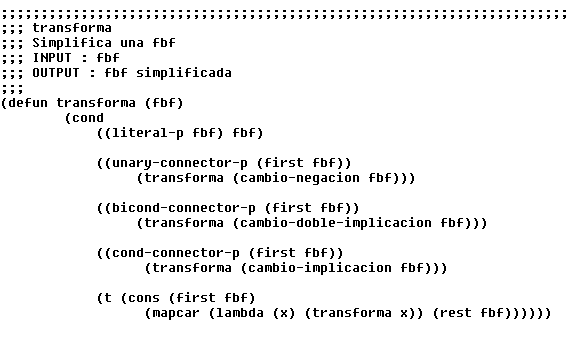
transforma(fbf)

Si la fbf es un literal, lo devuelve igual

Si la fbf es una negación, llama a transforma sobre la expresión que devuelve cambio-negacion(fbf)

Si la fbf es una doble implicación, llama a transforma sobre la expresión que devuelve cambio-doble-implicación(fbf)

Si la fbf es una implicación, llama a transforma sobre la expresión que devuelve cambio-implicación(fbf).

En cualquier otro caso( ands y ors), hace un cons de el primer elemento (^ o v ) con el resultado de aplicar transforma a todos los elementos que restan de la fbf.

***Ejercicio 5:***

En primer lugar, para el apartado **5.1** hemos realizado las siguientes BFS.

En el apartado **5.2** se nos pide adjuntar el pseudocódigo del algoritmo BFS :

BFS ( destino cola-de-caminos grafo)

Si la cola está vacía, devuelve nill (caso base)

Si no:

Establece el nodo origen

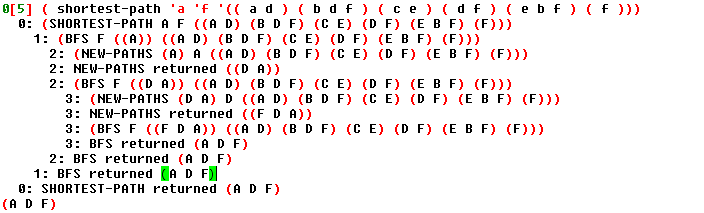
Obtiene el primer camino de la cola de caminos

Si el nodo origen es el destino, devuelve el camino

Si no:

Llama de nuevo al algoritmo, con la cola de caminos actualizada con los nodos hijo del nodo origen

En el apartado **5.5,** se nos pregunta por qué la función shortest-path encuentra el camino más corto entre dos nodos del grafo. Esto se debe a que este algoritmo llama a bfs, introduciendo como parámetro el destino, el grafo, y la cola de caminos con tan solo un elemento: el nodo origen. De esta manera, el algoritmo buscará los caminos desde ese nodo hasta todos los posibles del grafo, encontrando por lo tanto el mejor camino, ya que sabemos que bfs es óptimo para estas búsquedas.

Como pide el apartado **5.6,** ilustramos la secuencia de llamadas del código, para estudiar correctamente su funcionamiento:

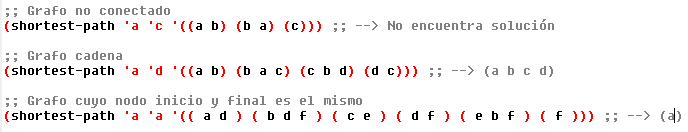
Básicamente, shortest-path llama a la función de búsqueda en anchura con el primer nodo A, el nodo final F y el grafo. Una vez en BFS, ésta llama a new-paths con el nodo A para encontrar sus vecinos, devolviendo ((D A)), pues A y F no están conectados. Volvemos a llamar a BFS pero esta vez la lista de nodos es la devuelta por new-paths ((D A)), como D tampoco es el nodo final, volvemos a llamar a new-paths para conocer los nodos de D, y nos devuelve ((F D A)); llamamos de nuevo a BFS con la nueva cola de nodos, como el primer nodo de ésta es F, ya hemos encontrado la solución, luego devolvemos la cola del revés pues es el camino que hay que realizar: A -> D -> F.

En el apartado **5.7** se pide hallar el camino más corto entre B y G. Se puede apreciar fácilmente que el camino más corto es B -> D -> G. Efectivamente es lo que nos devuelve la siguiente llamada:

*(shortest-path 'b 'g '((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))) ;; --> (b d g)*

Como primer nodo está B, como último nodo está G y como grafo, al no ser dirigido, va a haber algunas repeticiones, hemos añadido para cada nodo sus vecinos en orden alfabético. Además, hemos realizado los siguientes casos especiales:



Finalmente, el apartado **5.8** nos pide modificar el código para evitar los ciclos infinitos. Un ejemplo de ciclo es un grafo no conexo, es decir, los nodos “a” y “b” están conectados (no dirigido) y hay un tercer nodo “c” que no está conectado. Si intentamos ir de “a” a “c” entrará en una recursión infinita entre “a” y “b” hasta encontrar “c” pero nunca lo va a encontrar. Para ello hemos ejecutado la siguiente sentencia:

(shortest-path 'a 'c '((a b) (b a) (c)))

Lo que hace nuestro algoritmo BFS es, en este caso, ir devolviendo (a), (ba), (aba)… y así sucesivamente. Para ello añadimos una nueva lista donde vamos guardando los nodos explorados, si encontramos un nodo ya explorado, lo ignora.

La mecánica es la siguiente:

* shortest-path-improved llama a bfs-improved con los parámetros normales, tal y como lo hacía en el apartado 5.7
* bfs-improved llama a bfs-improved-recursive parándole como parámetro el nodo final, la cola que al principio sólo incluye en primer nodo, nil, es decir, una lista vacía con los nodos explorados, y el grafo completo.
* Finalmente bfs-improved-recursive funciona exactamente igual que bfs, la única diferencia es que el nuevo tiene una lista *explored* donde va almacenando todos los nodos que ha explorado. De este modo, si el nodo ha sido explorado, en vez de devolver la sentencia recursiva con new-paths, la devolvemos con (rest queue) para eliminar los nodos explorados y que no haya recursiones infinitas.

Bfs-improved-recursive

Función que encuentra un camino entre los nodos dados

INPUT:

end : nodo final.

Queue : cola con los nodos a explorar, al principio sólo contiene el primer nodo, por el que empezamos.

Explored : lista con los nodos que ya se han explorado.

Net : grafo completo

OUTPUT : camino entre el nodo final y el primer nodo

Bfs-improved-recursive ( end, queue, explored, net)

Si queue está vacía devuelve nil (no hay camino)

Si queue sólo contiene un elemento devuelve nil

Sea path = primer camino de queue

Sea node = primer elemento/nodo de queue

Si nodo == end devolvemos path del revés (camino encontrado)

Si nodo está en la lista de explorados

Devuelve función con end, (rest queue), explored y net

Sino añadimos nodo a la lista de explorados y devolvemos la función normal



Podemos observar que ahora las funciones van correctamente y no encuentran bucles:

