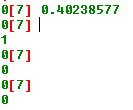
***Inteligencia Artificial: Práctica 1***

***Ejercicio 1:***

En el apartado **1.1**, 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.

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)

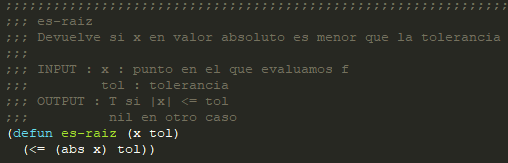
Probamos esta función con distintos valores en el apartado **1.4** y obtenemos:

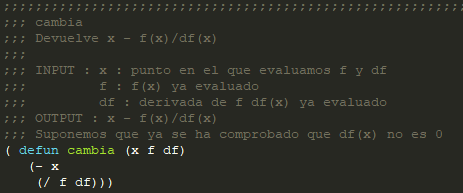
1. *(get-vectors-category '(()) '(()) #'cosine-distance-rec) ;;; --> NIL*
2. *(get-vectors-category '(()) '(()) #'cosine-distance-mapcar) ;;; --> NIL*
3. *(get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance-rec) ;;; --> ((2 0.40238577))*
4. *(get-vectors-category '((1 4 2) (2 1 2))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

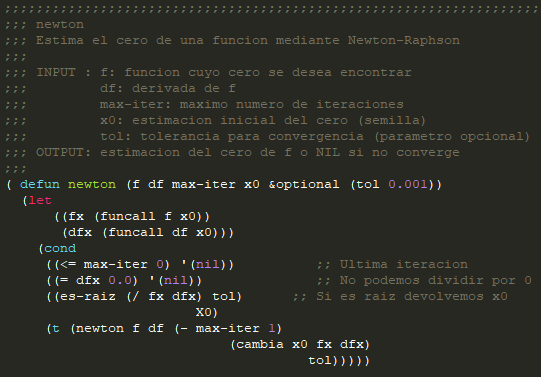
***Ejercicio 2:***

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.
2. La función *cambia* simplemente realiza el cambio x = x – f(x)/df(x) para la llamada recursiva sin que el código quede tan sucio.
3. La función newton pedida que encuentra una raíz de una función con el método de Newton de una manera recursiva.







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)))*

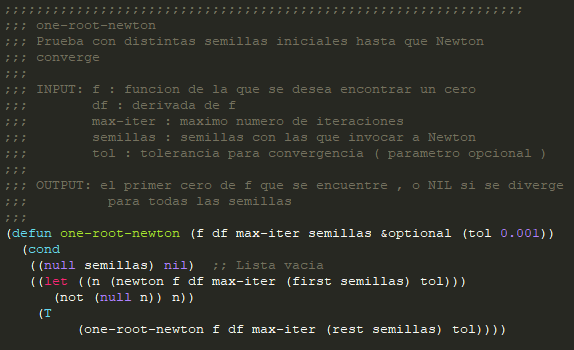
*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 30 -2.5) ;;---> -3.0000203*

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:



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)))*

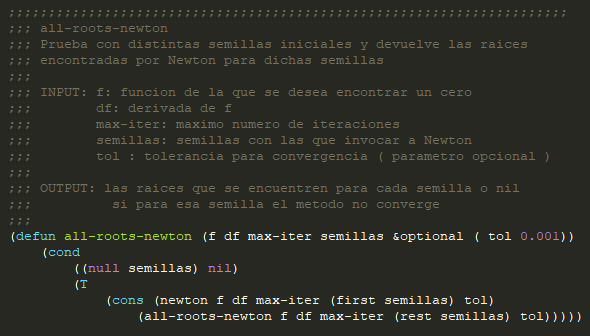
*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 20 '(3.0 -2.5)) ;;---> 4.000084*

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.



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)))*

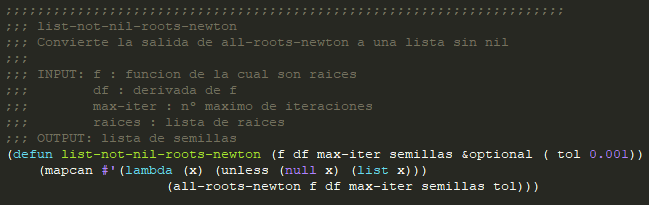
*#'(lambda (x) (- (\* x (- (\* x 3) 4)) 11)) 20 '(0.6 3.0 10000))*

*;; ---> (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:



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))*

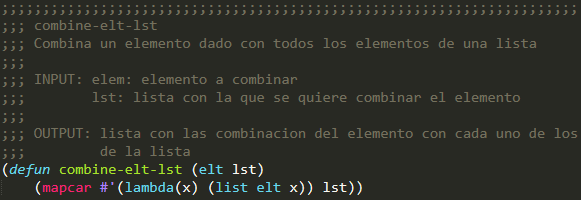
*;; ---> (0.99999946 4.000084)*

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:***

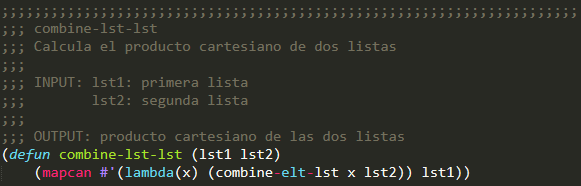
En el apartado **3.1** teníamos que combinar un elemento con una lista, dando lugar a los distintos pares con el siguiente código:



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))*

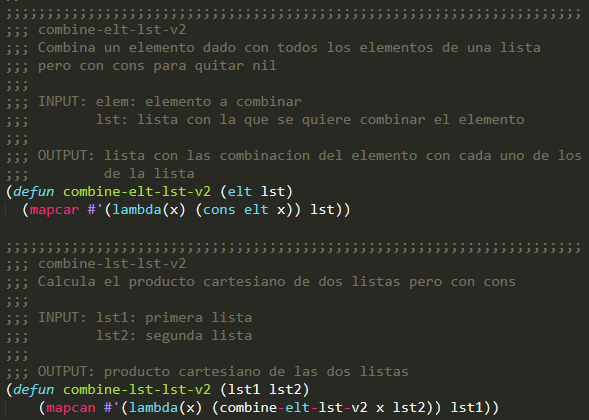
A continuación, en el apartado **3.2 s**e pedía combinar dos listas, para ello hemos utilizado la función combine-elt-lst anterior recursivamente:



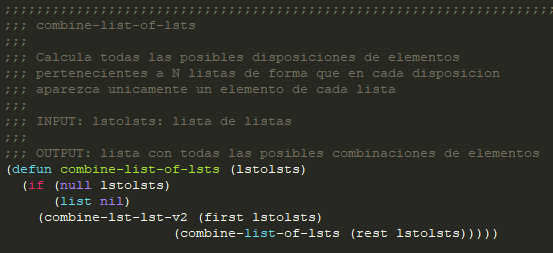
Dándonos los siguientes resultados:

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:



Son básicamente iguales que las iniciales y con ellas hemos creado la siguiente:



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)*