Javier Delgado del Cerro y Javier López Cano - Pareja 2

Prática 1

Ejercicio 1

Apartado 1.1

Batería de ejemplos

Comentar antes de empezar este apartado que no incluiremos en ninguna de estas baterías de ejemplos las expresiones pedidas en la memoria, que aparecen explicadas en *Comentarios sobre la impementación*, para evitar repetir información. Además, esta batería de ejemplos está siempre en el mismo orden que la ejecución del programa, y seguida en dicha ejecución por los ejempos pedidos, de forma que sea muy sencillo comprobar su verosimilitud.

En este caso específico, podemos ver que casi todos los casos conflictivos han sido pedidos en el enunciado, con lo que quedaría simplemente comprobar el simétrico de `(cosine-distance nil '(1 2 3))` y comprobar con algunos ejemplos que, efectivamente, se calcula correctamente. Como el resultado coincide en la versión recursiva y en la mapcar, lo expresamos como una única fución:

```
(cosine-distance '(1 2 3) nil) ;;; ---> NIL
(cosine-distance '(1 0) '(0 1)) ;;; ---> 1
(cosine-distance '(1 2 3) '(1 2 3)) ;;; ---> 0
(cosine-distance '(1 2 3) '(-1 -2 -3)) ;;; ---> 2
```

Como esperábamos (y como se puede ver al ejecutar el archivo), todos los resultados son correctos a excepción de la tercera prueba, donde en vez de obtener 0 obtenemos $5.96 * 10^{-8}$, que se debe únicamente a las aproximaciones del procesador (probablemente sobre todo en las raices cuadradas).

Pseudo-código

La única diferencia en el pseudocódigo de las dos funciones pedidas es la forma de calcular el producto escalar, por lo que especificaremos una única función `cosine-distance` que emplee `dot-product`, y dicho producto escalar será uno de los dos explicados.

```
cosine-distance(x, y):
    denominador = raiz(dot-product(x, x))*raiz(dot-product(y, y))
    si denominador == 0:
        devolver null
    si no:
        devolver dot-product(x, y)/denominador

dot-product-rec(x, y):
    si (x vacio) o (y vacio):
        devolver 0
    devolver (primer elem x)*(primer elem y)+dot-product-rec(resto de x, resto de y)

dot-product-mapcar(x, y):
```

```
    sumar(multiplicar el elemento xi por el yi)
```

Comentarios sobre la implementación

Para desarrollar las funciones `cosine-distance-rec` y `cosine-distance-mapcar` creamos dos funciones que nos permiten calcular el producto escalar de dos vectores de forma recursiva y usando mapcar, estas funciones son `dot-product-rec` y `dot-product-mapcar` respectivamente. Además, para evitar repeticiones de código creamos una tercera función `cosine-distance` que recibe como parámetro la función distancia a utilizar, con lo que `cosine-distance-rec` y `cosine-distance-mapcar` son simplemente llamadas a `cosine-distance`. El resultado de la evaluación de los casos indicados sería:

Expresión	Evaluación recurrente	Evaluación mapcar
`(cosine-distance '(1 2) '(1 2 3))`	0.40238577	0.40238577
`(cosine-distance nil '(1 2 3))`	NIL	NIL
`(cosine-distance '() '())`	NIL	NIL
`(cosine-distance '(0 0) '(0 0))`	NIL	NIL

Apartado 1.2

Batería de ejemplos

En este caso, la batería de ejemplos que hemos pensado (sin tener en cuenta los pedidos posteriormente, ni los dados en el enunciado como ejemplo, que también funcionan correctamente) serían:

```
(order-vectors-cosine-distance '(1 2 3) '((0 0 0))) ;; --> NIL
(order-vectors-cosine-distance '(0 0 0) '((1 2 3))) ;; --> NIL
(order-vectors-cosine-distance '(1 2 3) '((1 2 3) (2 1 1)) 0.99) ;; --> ((1 2 3))
(order-vectors-cosine-distance '(1 2 3) '((1 2 3) (2 1 1)) 1) ;; --> ((1 2 3))
(order-vectors-cosine-distance '(1 0) '((1 0) (0 1) (1 0.1) (1 0.2))) ;; --> ((1 0) (1
0.1) (1 0.2) (0 1))
(order-vectors-cosine-distance nil '((1 2 3) (2 1 1))) ;; --> NIL
(order-vectors-cosine-distance '(1 2 3) nil) ;; --> NIL
(order-vectors-cosine-distance nil nil) ;; --> NIL
```

Al igual que antes, podemos ver en el tercer caso como, por errores de aproximación, el nivel de confianza 1 no es útil, pues el resultado es NIL en vez de el esperado, al ser la distancia $5.96 * 10^{-8}$ en vez de cero.

Pseudo-código

```
order-vectors-cosine-distance (vector lst-of-vectors confidence-level):
    para cada v en lst-of-vectors:
        resultado.añadir(v, cosine-distance(vector, v))
    resultado.ordenar()
    delvolver resultado.primer-elemento-tuplas()
```

Comentarios sobre la implementación

Para codificar dicha función, es claro que necesitamos obtener la distancia coseno de cada uno de los vectores de `lst-of-vectors` a `vector`, eliminar aquellos cuya distancia sea mayor que `1 - confidence-level` y finalmente, ordenarlos respecto a dicho parámetro de menor a mayor. Para conseguir esto, desarrollamos la funcion `map-vectors-cosine-distance` que se encarga de crear una lista de tuplas (cada una de ellas con un vector y su distancia coseno respecto al vector de referencia) con los vectores cuya distancia coseno es menor o igual que que `1 - confidence-level`. Posteriormente, basta con ordenar esta lista (para lo que usamos `vector-order` y una copia de la lista, pues `sort` es una función destructiva) y coger únicamente el primer elemento de cada tupla. Como alternativa, podríamos haber programado nosotros mismos un algoritmo de ordenación como quicksort y emplear dicha función directamente, pero creemos que la implementación oficial probablemente esté mucho más optimizada que la que podemos desarrollar nosotros.

Al ejecutar los ejemplos dados en el PDF obtenemos exactamente los mismos resultados. Para los casos de prueba pedidos, obtenemos lo siguiente:

Expresión	Evaluación
`(order-vectors-cosine-distance '(1 2 3) '())`	NIL
`(order-vectors-cosine-distance '() '((4 3 2) (1 2 3)))`	NIL

Como era de esperar.

Apartado 1.3

Batería de ejemplos

De nuevo, evitamos probar las expresiones que se piden posteriormente y están explicdas en la tabla del apartado 1.4. Además, usamos la expresión `cosine-distance` por ser el resultado independiente del uso de la función recursiva o la función con mapcar. La batería de ejemplos sería entonces:

```
(get-vectors-category '((1 1 2) (2 2 1)) '((1 2 1) (2 1 2)) #'cosine-distance) ;;; -->
((2 0.0) (1 0.0))
(get-vectors-category '((1 1 2)) '((1 2 1) (2 1 2)) #'cosine-distance) ;;; --> ((1
0.19999999) (1 0.0))
(get-vectors-category '((1 0 0)) '((1 2 1) (2 1 2)) #'cosine-distance) ;;; --; (NIL NIL)
(get-vectors-category '((1 1 2) (2 2 1)) '((1 0 0)) #'cosine-distance) ;;; --; (NIL)
(print (get-vectors-category nil '((1 2 1) (2 1 2)) #'cosine-distance) ;;; --> NIL
(get-vectors-category '((1 2 1) (2 1 2)) nil #'cosine-distance) ;;; --> NIL
(get-vectors-category nil nil #'cosine-distance) ;;; --> NIL
(get-vectors-category '((1 1 2 3) ()) '((1 1 2 3) (2 2 4 6)) #'cosine-distance) ;;; -->
((1 0) (1 0.02536))
(get-vectors-category '((1 4 5 6) (2 2 1 3)) '(() (2 4 5 6)) #'cosine-distance) ;;; -->
(NIL (1 0.0))
(get-vectors-category '(() (1 4 5 6)) '(() (2 4 5 6)) #'cosine-distance) ;;; --> (NIL (1
0.0))
```

Consideramos que en el caso 3 y 4 tiene que devolver listas de NIL, pues aunque ambas listas con válidas (las te categorías y las de textos), ninguno de los textos tiene una categoría que se le pueda asociar.

Sin embargo, en los casos 5, 6 y 7, como o solo una o ninguna de las listas son válidas, devolvemos nil directamente.

En el caso 8, al igual que ha pasado anteriormente, cabe destacar que no obtenemos exactamente dicho resultado, pues dadas las aproximaciones del ordenador el par `(1 0)` se convierte en `(1 5.9604645E-8)`.

Pseudo-código

```
get-vectors-category (categories texts distance-measure):
    para cada text en texts:
        lista.crear()
        para cada cat en categories:
            lista.añadir(id(cat) distance-measure(vector(text), vector(text)))
        resultado.añadir(lista.coger_minima_distancia())
    devolver resultado
```

Comentarios sobre la implementación

En este tercer apartado, empleamos dos funciones auxiliares para llegar a la pedida. En primer lugar `get-text-category-dist` nos permite, dado un texto y una categoría, obtener una lista con una tupla con el id de la categoria y la distancia del texto a la categoría, y en segundo lugar `get-text-category` nos permite obtener la categoría de un texto. Hacemos que `get-text-category-dist` nos devuelva una lista con una tupla para, de esta forma, poder usar mapcan en `get-text-category`, y controlar los casos en los que alguno de los vectores sea, quitando el id, todo ceros.

De esta forma en `get-vectors-category` simplemente tenemos que llamar a `get-text-category` para cada uno de los textos.

Hemos decidido además que, en el caso de que alguna de las listas pasadas no tenga elementos, la función devuelva nil, pues en ese caso será imposible que la función haga su cometido.

Apartado 1.4

Usando la macro time con algunos ejemplos para medir la diferencia de rendimiento entre ambas funciones, podemos ver que la función recursiva es mucho menos eficiente, como era de esperar, pues sabemos que mapcar paraleliza la ejecución en distintos hilos. Así, por ejemplo, para dos vectores de cuatro elementos como textos y categorías, la función recursiva tarda $8.1 * 10^{-5} s$ mientras que la función mapcar tarda $3.8 * 10^{-5} s$. Si aumentamos los vectores para que sean de cinco elementos, y usamos cuatro de estos vectores para cada una de las listas, vemos que los tiempos en este caso serían $1.91 * 10^{-4} s$ y $7.6 * 10^{-5} s$ respectivamente, por lo que es claro además que el tiempo de la función recursiva crece mucho más rápido que el de la función mapcar.

Además, para los ejemplos planteados obtenemos los siguientes resultados:

Expresión	Evaluación
`(get-vectors-category '(()) '(()) #'cosine-distance)`	(NIL)
`(get-vectors-category '((1 4 2) (2 1 2)) '((1 1 2 3)) #'cosine-distance)`	`((2 0.40238577))`
`(get-vectors-category '(()) '((1 1 2 3) (2 4 5 6)) #'cosine-distance)`	(NIL NIL)

Como hemos mencionado en la parte de *batería de ejemplos* del apartado anterior, en el primer y tercer caso, devuelve una lista con NILs porque, para los textos (que pueden o no ser NIL) no hay ninguna categoría.

Ejercicio 2

Apartado 2.1

Batería de ejemplos

En este apartado, los ejemplos que se nos proporcionan en el enunciado para probar el funcionamiento son bastante completos para los casos en los que no se introduce tolerancia, para estos solo faltaría el caso en el que se introduce directamente la raíz de la función. Por tanto en esta batería incluimos este caso y todos los que introducen una tolerancia:

```
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4))
11)) 20 4.0) ;;---> 4.0
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4))
11)) 20 0.6 0.001) ;;---> 1.0
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4))
11)) 30 -2.5 0) ;;---> -3.0
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4))
11)) 10 100.0 0.005) ;;---> NIL
(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4))
11)) 20 4.0 0.007) ;;---> 4.0
```

En el primer caso la función devuelve 4.0 de forma exacta en la primera iteración ya que directamente se ha introducido como semilla la raíz, lo mismo ocurre en el último caso (igual que el primero pero introduciendo una tolerancia).

En el caso tres la función devuelva un valor muy cercano a 1, 0.99999946, en lugar de exactamente 1 pues $1 - 0.99999946 < 0.001$ que es la tolerancia introducida y por tanto el algoritmo es correcto. En el cuarto caso el resultado debe ser exacto pues la tolerancia introducida es 0.

El penúltimo caso devuelve NIL como se espera ya que no encuentra una raíz que cumpla con la tolerancia pedida en las diez iteraciones que se le permite hacer.

Pseudocódigo

```
newton (funct, deriv, iterations, x0, tol):
    si iterations <= 0:
        devolver null
    si no:
        si valor-absoluto(funct(x0)/deriv(x0)) <= tol:
            devolver x0
        si no:
            devolver newton (funct, deriv, iterations-1, x0-(funct(x0)/deriv(x0), tol)
```

Comentarios sobre la implementación

En este caso, debido a la simplicidad de la función pedida, hemos decidido no emplear funciones auxiliares en su implementación.

Hemos empleado un `let` para guardar el cociente `f(x0)/df(x0)` y así evitar repetir el cáculo varias veces en la función.

Por último, mediante el uso de varios `if` controlamos que el algoritmo (que es recursivo) no realice mas iteraciones de las permitidas (reduciendo en 1 el campo `max-iter` en cada iteración y comprobando que no es cero) , y si el valor introducido como semilla en la iteración (`x0`) cumple la tolerancia (y por tanto es la raíz que buscamos) o no y por tanto se debe realizar una iteración más.

Los resultados para los ejemplos que se nos proponen en el enunciado son los siguientes:

Expresión	Evaluación
`(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 3.0)`	4.000084
`(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 0.6)`	0.99999946
`(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 30 -2.5)`	-3.0000203
`(newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 10 100.0)`	NIL

Que, como se puede ver, son los resultados esperados teniendo en cuenta que no son exactos pero satisfacen la tolerancia.

Apartado 2.2

Batería de ejemplos

De nuevo en este caso no repetimos los casos que ya se comprueban con los ejemplos proporcionados en el enunciado para probar el funcionamiento de la función pedida.

Sabiendo esto, la batería de ejemplos propuesta es:

```
(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x
3) 4)) 11)) 7 '(50.0 0.6)) ;;---> 0.99999946
(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x
3) 4)) 11)) 7 '(3.0 50.0)) ;;---> 4.000084
(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x
3) 4)) 11)) 1 '(3.0 -2.5)) ;;---> NIL
(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) ;;---> 1.0
```

Podemos observar que en todos los ejemplos salvo el último es aceptable obtener resultados no exactos siempre que cumplan la tolerancia $0,001$ pues esta es la que se establece por defecto si no se introduce ninguna. Por el contrario en el último ejemplo el resultado debe ser exacto pues se introduce una tolerancia exacta. En este último ejemplo podemos ver que al introducir una tolerancia diferente a la que se usa por defecto la función sigue ejecutándose de forma correcta.

En el primer ejemplo vemos que si el algoritmo de newton devuelve NIL con la primera semilla, esta función continua de forma correcta con la semilla siguiente. En el segundo caso comprobamos que si la primera semilla hace que newton devuelva un valor para la raíz el algoritmo termina y devuelve ese valor, y en el tercero vemos que si no encuentra una raíz para ninguna de las semillas en las iteraciones permitidas devuelve NIL.

Pseudocódigo

```
one-root-newton (funct, deriv, iterations, semillas, tol):
    si newton(funct, derivate, iterations, first(semillas), tol) == null:
        si rest(Semillas) == null:
            devolver null
        si no:
            devolver one-root-newton (funct, deriv, iterations, rest(semillas), tol)
    si no:
        devolver newton(funct, derivate, iterations, first(semillas), tol)
```

Comentarios sobre la implementación

De nuevo, en este caso tampoco hemos utilizado funciones auxiliares pero si hemos utilizado la función `let`, en concreto para guardar en una variable local el valor de la llamada a la función `(newton f df max-iter (first semillas) tol)` y así evitar repetición de código.

En este algoritmo hemos decidido emplear varios `if` para controlar cuando debe pararse la ejecución recursiva del algoritmo (cuando la ejecución de newton para alguna semilla produce un resultado distinto de `nil` o cuando `(rest semillas)` es una lista vacía) y para determinar si el algoritmo debe devolver `nil` o el valor que ha devuelto la función newton.

Como se puede observar tanto en el pseudocódigo como en el código, para la implementación de esta función hemos empleado la función newton creada en el apartado anterior.

Al ejecutar las sentencias propuestas en el enunciado obtenemos los siguientes resultados:

Expresión	Evaluación
`(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
`(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
`(one-root-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 1 '(3.0 -2.5))`	NIL

Que, de nuevo, a pesar de no ser exactos, son correctos pues satisfacen la tolerancia.

Apartado 2.3

Batería de ejemplos

En este apartado los ejemplos que se nos proporcionan para probar el funcionamiento del algoritmo pedido ya incluyen todas las posibles variaciones que merece la pena probar (con la tolerancia establecida por defecto), por lo tanto, en esta batería de ejemplos propuesta tan solo añadimos un ejemplo sin introducir tolerancia para probar el correcto funcionamiento del algoritmo, y uno introduciendo tolerancia para ver que también funciona correctamente si se decide introducir una tolerancia específica.

De este modo obtenemos la siguiente batería de ejemplos:

```
(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(-2.5 3.0 10000.0)) ;;---> (-3.0000203 4.0 nil)
(all-roots-newton #'(lambda(x) (* (- x 4) (- x 1) (+ x 3))) #'(lambda (x) (- (* x (- (* x 3) 4)) 11)) 20 '(-2.5 3.0 10000.0) 0) ;;---> (-3.0 4.0 nil)
```

Al ejecutar estos ejemplos observamos que ambos son correctos, y tan solo cabe destacar que, al igual que en los casos anteriores, el caso con tolerancia "0" ofrece resultados exactos mientras que el otro no, pero esto es correcto ya que a pesar de no ser exactos cumplen con la tolerancia.

Pseudocódigo

```
all-roots-newton (funct, deriv, iterations, semillas, tol):
    si rest(semillas) == null:
        devolver lista.crear(newton(funct, deriv, iterations, first(semilla), tol))
    si no:
        devolver lista.añadir(newton(funct, deriv, iterations, first(semilla), tol),
all-roots-newton(funct, deriv, iterations, rest(semillas), tol))
```

Comentarios sobre la implementación

A la hora de implementar el funcionamiento pedido en este apartado, hemos empleado al igual que en el apartado anterior, llamadas a la función newton creada en el apartado 2.1 y hemos almacenado el valor que esta devuelve en una variable local mediante la función `let` para evitar repetir código.

Esta función emplea un if para ver si se trata o no de la ultima iteración del algoritmo recursivo (comprobando si `(rest semillas)` es una lista vacía o no), si lo es, devuelve el valor obtenido de la función newton para esa semilla, y si no lo es, añade el valor obtenido de newton a la lista que se obtiene al llamar de nuevo a la función `all-roots-newton` con `(rest semillas)` (es decir, la lista con los valores de la función newton para el resto de semillas obtenido en las iteraciones siguientes).

De este modo obtenemos a la salida una lista con las raíces que halla newton para las semillas pasadas como argumento (o NIL en caso de que newton no converja para alguna de ellas).

Al ejecutar las expresiones propuestas en el enunciado obtenemos los siguientes resultados:

Expresión	Evaluación
`(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)
`(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))`	(0.99999946 4.000084 NIL)
`(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 estos ejemplos se puede ver que el algoritmo es correcto en los casos en que ninguna semilla de la lista hace que el algoritmo de newton converja, en el caso en que todas las semillas hacen que converja, y en el caso en el que algunas hacen que converja y otras no. Es por esto por lo que, como hemos dicho en el apartado de la batería de ejemplos, estos casos no se han incluido ahí.

Podemos observar que los resultados son correctos pues cuando el algoritmo no converge introduce NIL en la lista, y cuando converge los resultados satisfacen la tolerancia establecida por defecto que es de 0,001.

Apartado 2.3.1

Batería de ejemplos

Para probar este apartado concreto se ha decidido emplear la siguiente batería de ejemplos:

```
(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)
(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)) ;;---> (0.99999946 4.000084)
(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
(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) 0) ;;---> (1.0 4.0)
```

En el primer ejemplo propuesto comprobamos que al emplear una lista de semillas en las que todas permiten que el algoritmo newton converja en el número de iteraciones permitido la función no elimina ninguno de los resultados de la lista.

En el segundo empleamos una lista de semillas en las que hay algunas que hacen que el algoritmo converja y otras no y observamos que, como esperábamos, el algoritmo elimina los NIL de la lista resultado.

En el tercer ejemplo vemos que si introducimos una lista de semillas en las que ninguna permite que el algoritmo converja, el resultado de la función es NIL, es decir, la lista vacía, que es correcto.

Por último comprobamos que al introducir una tolerancia diferente a la que se establece por defecto el algoritmo también funciona correctamente. Cabe mencionar que como esperábamos, en este caso con tolerancia 0 los resultados son exactos, mientras que en el resto no es necesario que lo sean siempre que cumplan la tolerancia 0,001 que se establece por defecto.

Pseudocódigo

```
list-not-nil-roots-newton (funct, deriv, iterations, semillas, tol):
    para cada elemento de all-roots-newton(funct, deriv, iterations, semillas, tol):
        si es null:
            no.añadir.elemento.lista()
        si no:
            añadir.elemento.lista()
    devolver lista
```

Comentarios sobre la implementación

En este apartado no hemos empleado recursividad para implementar el funcionamiento pedido, sino que hemos empleado la función `mapcan` para modificar todos los elementos de la lista, como se nos exigía en el enunciado.

Para la implementación del algoritmo hemos empleado llamadas a la función `all-roots-newton` creada en el apartado anterior, y, aplicando una función lambda sobre cada uno de los elementos de la lista que esta función devuelve (usando `mapcan`) eliminamos los posibles NIL de la lista, devolviendo dicha lista modificada.

Esta función lambda empleada sobre la lista obtenida de `all-roots-newton` emplea un `if` para comprobar si el elemento de la lista es NIL, si lo es, lo quita (sustituyéndolo por `'()`), y si no, lo deja como está `(list x)`.

Al probar la expresión que se nos propone en el enunciado obtenemos:

Expresión	Evaluación
`(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))`	(0.999999464.000084)

Que es, de nuevo, el resultado que esperábamos obtener.

Ejercicio 3

Apartado 3.1

Batería de ejemplos

Dada la simplicitud del apartado, además de los ejemplos proporcionados en el enunciado y evaluados en el apartado de *comentarios sobre la implementación*, solo falta comprobar su correcto funcionamiento mediante la expresión:

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

Pseudo-código

```
combine-elt-lst (elt, lst):
    para cada elemento en lst:
        resultado.añadir(lista(elt lst))
    devolver resultado
```

Comentarios sobre la implementación

En este caso, la implementación ha sido muy sencilla y lo único a destacar es la decisión que hemos tomado de, si o bien el `elt` o bien `lst` son vacíos o nil, devolver nil directamente. Por tanto, la evaluación de las expresiones pedidas es la siguiente:

Expresión	Evaluación
`(combine-elt-lst 'a nil)`	NIL
`(combine-elt-lst nil nil)`	NIL
`(combine-elt-lst nil '(a b))`	NIL

Apartado 3.2

Batería de ejemplos

Al igual que en el ejemplo anterior, todas las expresiones que pueden causar algún tipo de conflicto están evaluadas en el apartado de *comentarios sobre la implementación*, con lo que solo es necesario probar su correcto funcionamiento, para lo que es suficiente la expresión:

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

Pseudo-código

Una vez desarrollada la función del aparado anterior, esta es bastante sencilla

```
combine-lst-lst (lst1, lst2):
    para cada elt en lst1:
        resultado.concatenar(combine-elt-lst(elt lst))
    devolver resultado
```

Comentarios sobre la implementación

Como `combine-elt-lst` devuelve una lista, usamos mapcan para aplicar esta función con cada elemento de `lst1` sobre `lst2`. De esta forma, mapcan concatena las listas. Además, en el caso de ser alguno de los campos vacío o NIL, mapcan devuelve simplemente nil, pues no añade nada la lista. Por tanto, la evaluación de las expresiones pedidas es la siguiente:

Expresión	Evaluación
`(combine-lst-lst nil nil)`	NIL
`(combine-lst-lst '(a b c) nil)`	NIL
`(combine-lst-lst nil '(a b c))`	NIL

Apartado 3.3

Batería de ejemplos

De nuevo, todas las expresiones que pueden dar lugar a algún tipo de conflicto están evaluadas en el apartado de *comentarios sobre la implementación*, con lo que basta con ver que funciona correctamente para dos listas (su comportamiento tiene que ser exactamente igual que el de `combine-lst-lst`) y para tres o más listas:

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

Pseudo-código

Una vez desarrollada la función del aparado anterior, basta con utilizar una recursión cuyo caso base sea el caso en el que `lstolsts` tiene únicamente un elemento, pues en dicho caso el resultado sería, usando un ejemplo, `(combine-list-of-lsts '((a b c))) --> ((A) (B) (C))`

```
combine-list-of-lsts (lstolsts):
    si elementos(lstolsts) == 1:
        para cada item en primer elem lstolsts:
             resultado.añadir(lista(item))
        devolver resultado
    devolver combine-lst-lst(primer elem lstolsts, combine-list-of-lsts(resto lstolsts))
```

Comentarios sobre la implementación

Sin embargo, en la implementación nos encontramos con un pequeño problema, y es que la función `combine-elt-lst` crea una lista para cada par, lo que cumple las especificaciones pedidas para el apartado 3.1, sin embargo, al usar dicha función en este tercer apartado, necesitaríamos concatenar el par en otra lista, obteniento así una única lista con dos elementos, y no una lista con dos sublistas.

> Un ejemplo sería: queremos obtener `(A + 1)`, pero con la función actual obtenemos `(A (+ (1)))`. Como hemos dicho, la solución pasaría por concatenar los elementos (función `cons`) en vez de crear una lista, pero esto no cumpliría el formato de salida pedido en el enunciado para esta función, pues el resultado de evaluar `(combine-elt-lst 'a '(1 2 3))` sería `((A . 1) (A . 2) (A . 3))` en vez de `((A 1) (A 2) (A 3))`.

Por tanto, creamos dos nuevas funciones auxiliares, una `combine-elt-lst-aux` que simplemente implementa esta concatenación, y otra `combine-elt-lst-aux` cuya función es la misma que `combine-elt-lst` pero llamando en este caso a `combine-elt-lst-aux`.

Una vez programada la función, comprobamos con el ejemplo dado y con nuestra batería de ejemplos que funciona correctamente, y la evaluación sobre las expresiones pedidas sería:

Expresión	**Evaluación**
`(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) (+ -) ()))`	NIL
`(combine-list-of-lsts '((1 2 3 4)))`	`((1) (2) (3) (4))`
`(combine-list-of-lsts '(nil))`	NIL
`(combine-list-of-lsts nil)`	NIL