PRÁCTICA 1: LISP

1. **Vector más cercano a un vector dado:**
   1. **Norma p de un vector:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: x (vector del que se calcula la norma)

p (orden de la norma)

Salida:  n (norma p de x)

Procesamiento:

Si x es vacio,

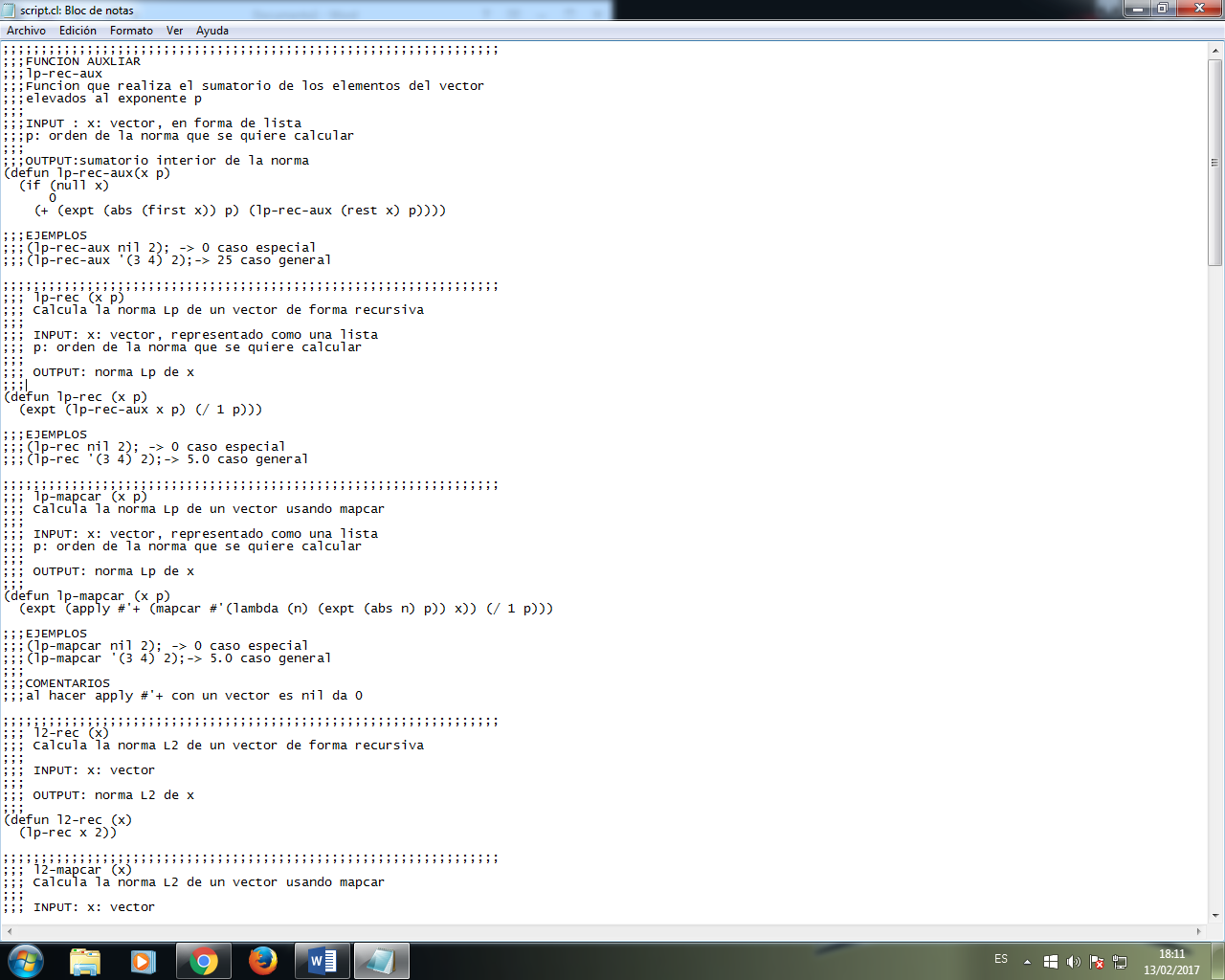
      evalúa a 0

en caso contrario

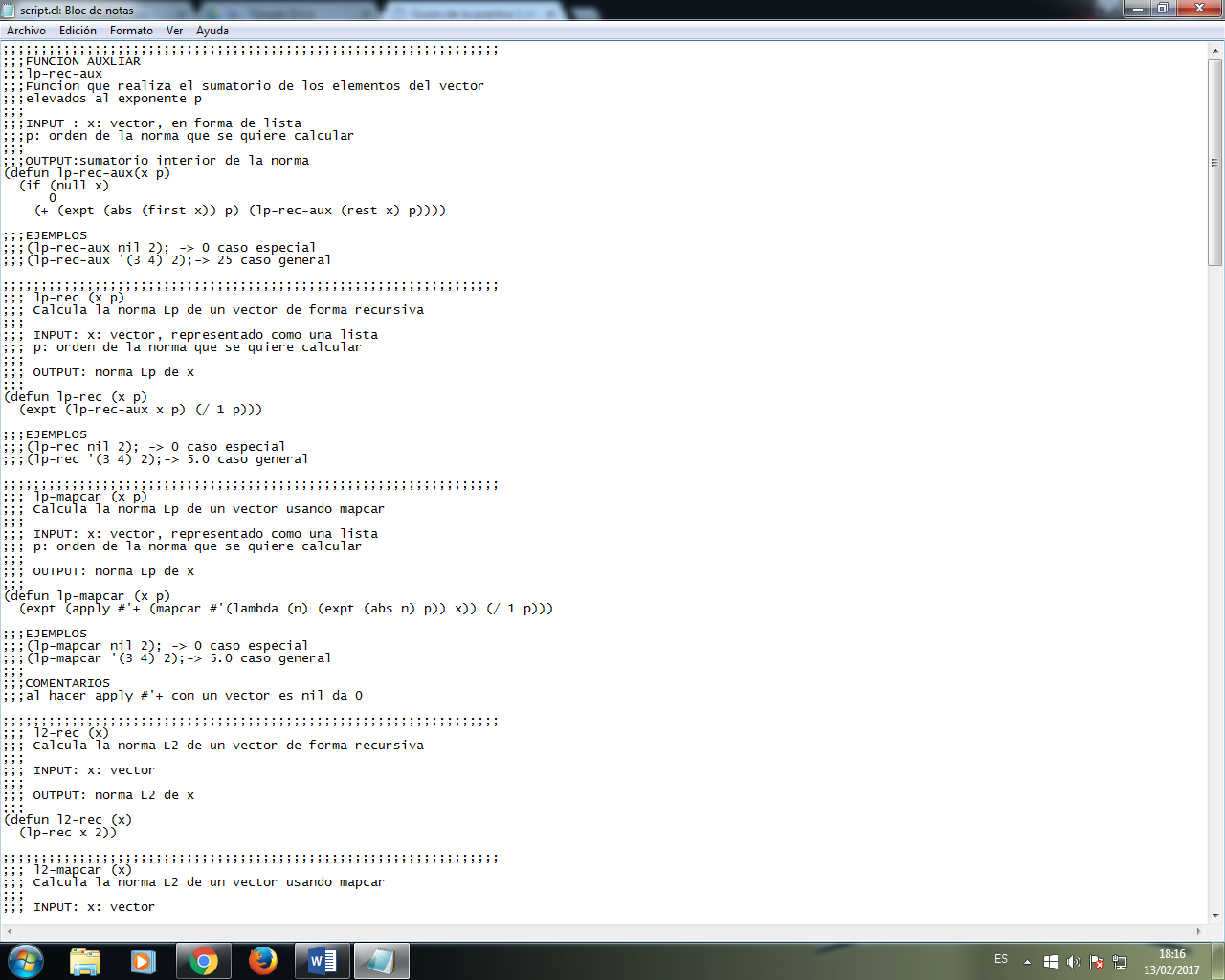
evalúa a (|primer-elemento(x)|^p + normap(siguientes-elementos(x))^(1/p)

1. **Usando recursión**

**CÓDIGO**



1. **Usando mapcar**



**COMENTARIOS**

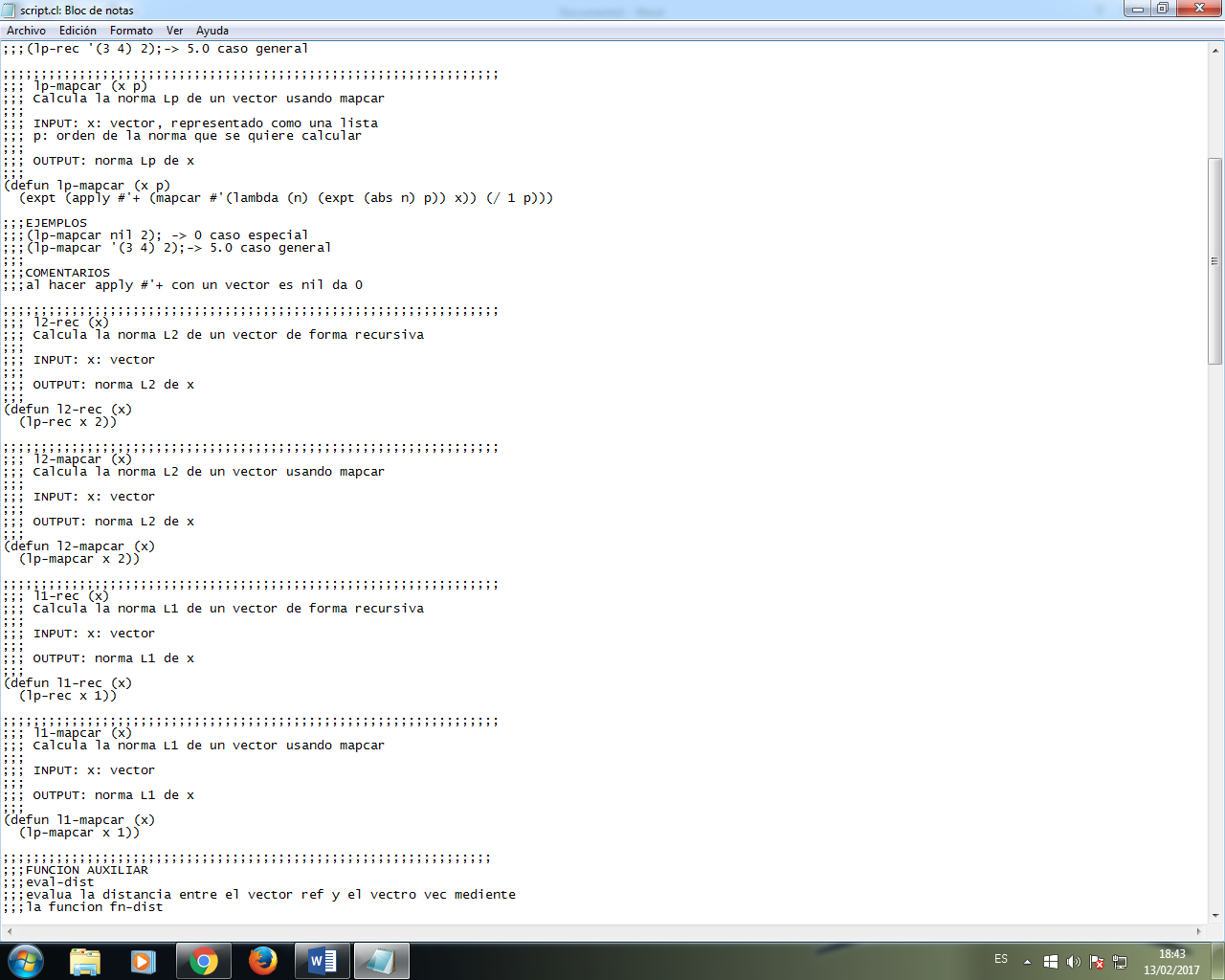
El código con mapcar es mucho más claro y sencillo que el de recursión, aunque el último es más intuitivo puesto que se trata de un sumatorio. Asimismo, cuando usamos mapcar, Lisp se encarga de comprobar los casos en los que la lista sea vacía y por tanto ahorramos comprobaciones.

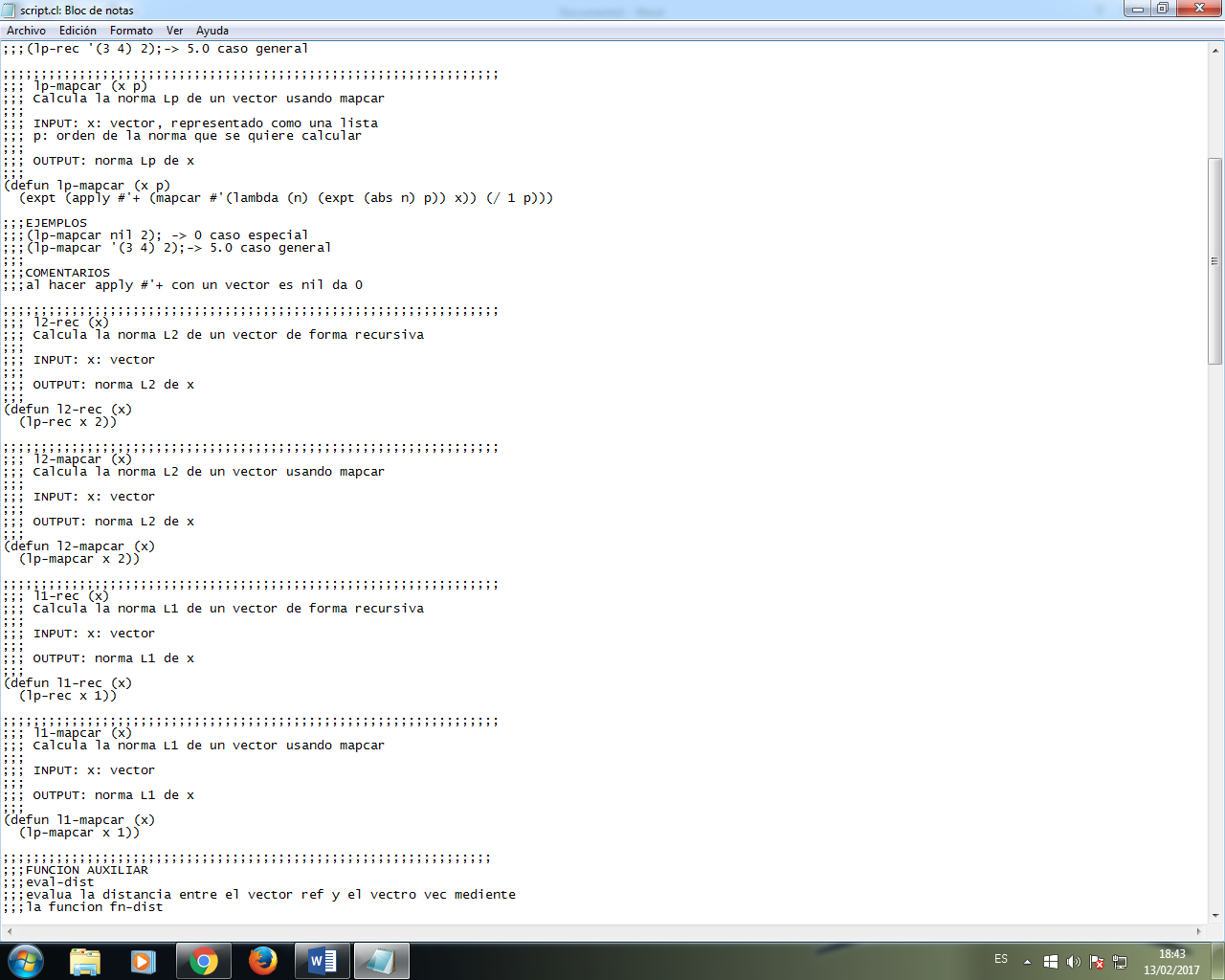
En la función recursiva hemos necesitado una función auxiliar para calcular el sumatorio. En la función principal ya podemos calcular la raíz una vez hemos llamado a la auxiliar.

* 1. **Norma euclídea y norma 1**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Una vez definidas las dos funciones anteriores, tan solo tenemos que llamarlas utilizando el valor de p que corresponda.





**COMENTARIOS**

Para probar la norma infinito podemos introducir vectores con valores negativos y ver que el valor absoluto se aplica correctamente. También podemos introducir vectores distintos cuya norma euclídea es igual pero la norma infinito cambia, por ejemplo, (1 1) tiene norma euclídea y (1 0), 1, pero ambos tienen norma infinito 1.

* 1. **Función nearest**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**PSEUDOCÓDIGO**

Entrada: lst-vectors(lista de vectores a comparar)

vector (vector referencia contra el que se compara)

fn-dist: referencia a función para medir distancias

Salida:   vector de entre los de lst-vectors más cercano al de referencia

Procesamiento:

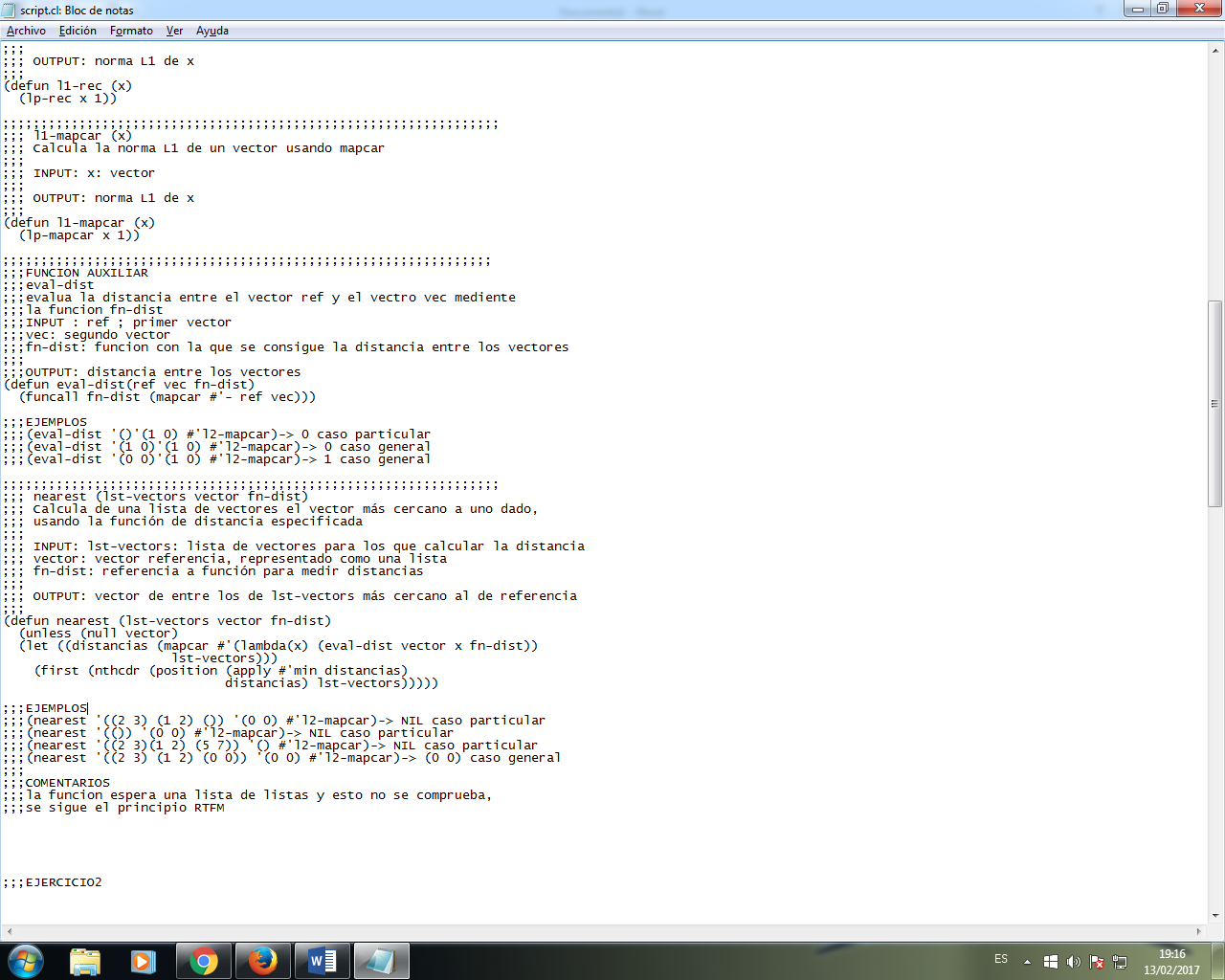
si no( vacío(vector) o vacío(lst-vectors) )

lista-distancias= aplicar (calcular-distancia(lst-vectores))

n=posición(minimo(lista-distancias))

devolver (lst-vectors(n))

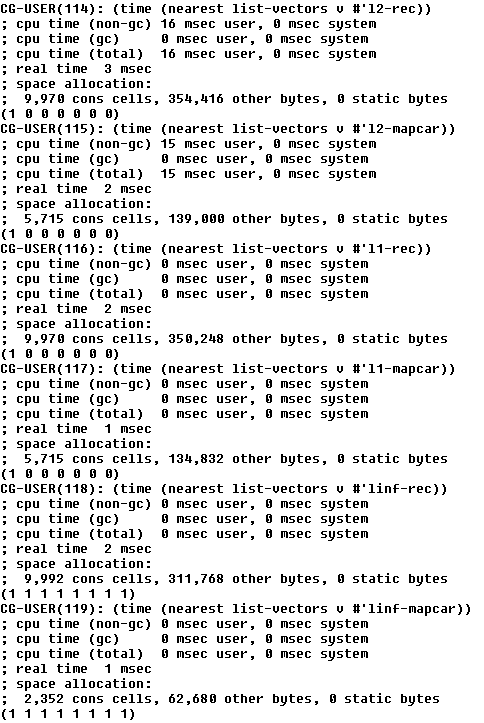
**CÓDIGO**

**COMENTARIOS**

Se ha utilizado mapcar para la implementación debido a que nos parecía más sencillo que una opción recursiva y por posibles ahorros de memoria (como al final se confirma con las pruebas realizadas a continuación)

Hemos necesitado una función auxiliar que compara un único vector contra el vector referencia.

* 1. **Pruebas función nearest con diferentes normas**

****

**COMENTARIOS**

Hemos medido los tiempos con veinticinco vectores de dimensión nueve y vemos la distancia al origen. Observamos que la norma euclídea es la más costosa, ya que se ha obtenido un tiempo de 3 ms. Asimismo, en general el mapcar es más eficiente que la recursión, tanto temporal como espacialmente (podemos ver como el número de bloques cons utilizados en todas las funciones recursivas es casi 10.000 mientras que en el mapcar se reduce aproximadamente a la mitad).

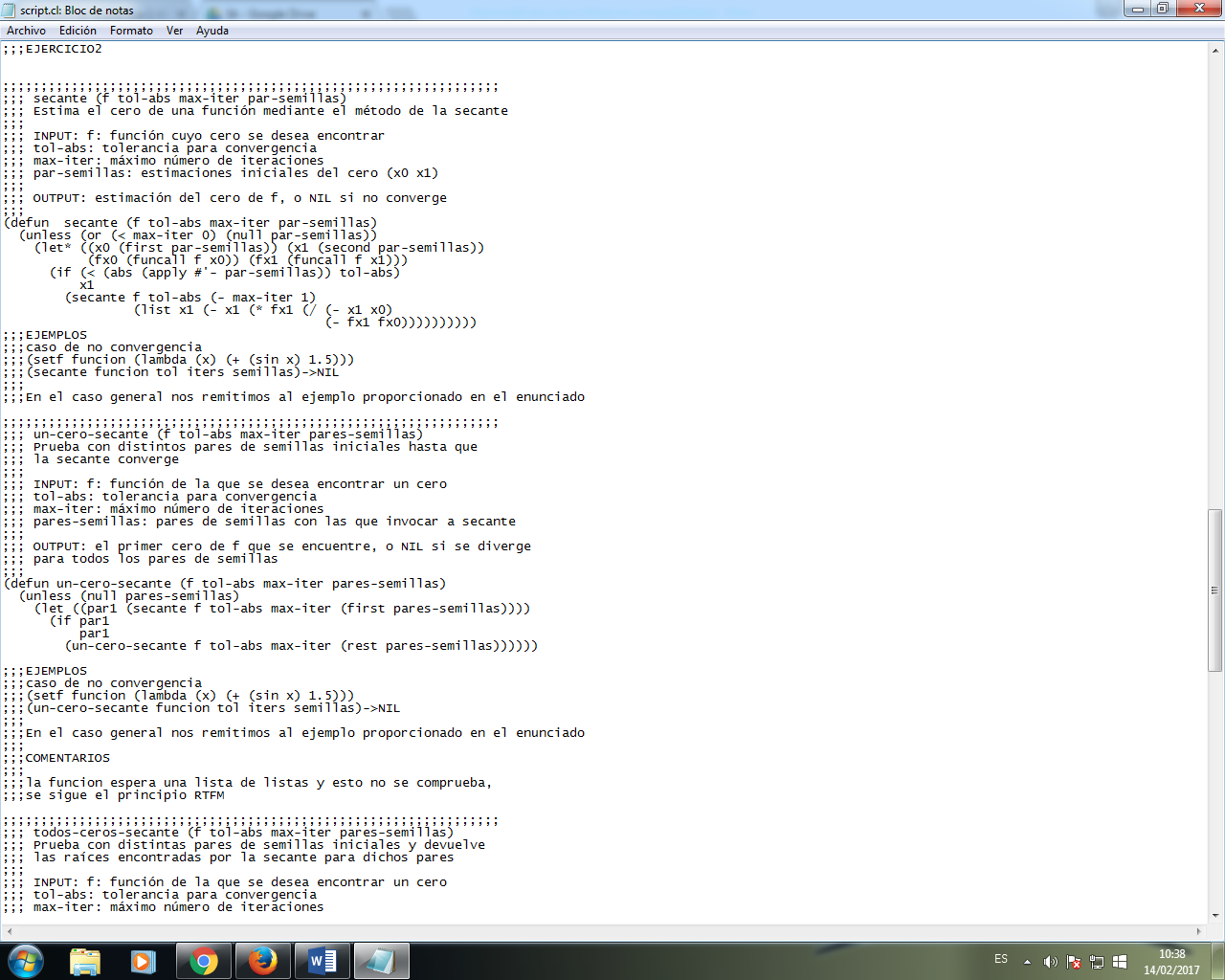
1. **Ceros de una función**
   1. **Función : secante**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Siguiendo el pseudocódigo del enunciado se ha implementado el código que se presenta más abajo.

**CÓDIGO**



**COMENTARIOS**

Sobre la implementación elegida destacamos que en cada llamada recursiva a la función creamos una nueva lista donde X1 pasa a ser X0 y el nuevo X1 se calcula mediante la fórmula que se nos ha proporcionado.

Hemos probado funciones que no tienen ceros para ver que ocurría. Una de ellas es sin(x)+1.5 que cumple el máximo de iteraciones y acaba con salida NIL. Otra de ellas es x2+1 que no llega al máximo de iteraciones puesto que las semillas crecen rápidamente y se desborda la memoria.

* 1. **Función : un-cero-secante**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: f: función

Tol-abs: tolerancia

max-iter: máximo iteraciones

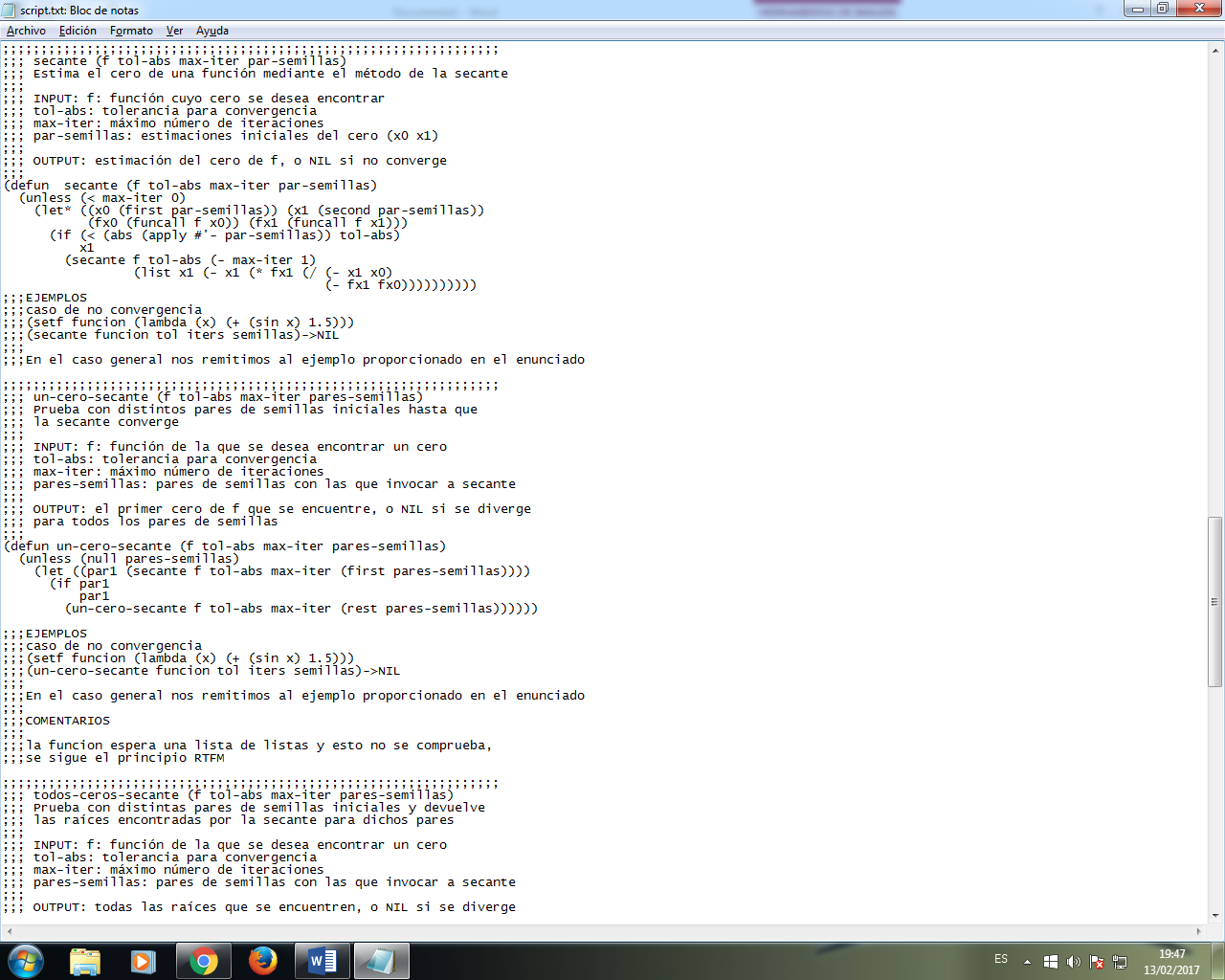
pares-semillas: lista-pares-semillas

Salida: El primer cero encontrado

Procesamiento:

si no (secante (primer-elemento(pares-semillas)))

entonces un-cero-secante(siguientes-elementos(pares-semillas))



**COMENTARIOS**

La función solo admite una lista de listas, en caso contrario se produce un error.

De nuevo hemos probado el caso de no convergencia con sin(x)+1.5 cumple el máximo de iteraciones con cada par de semillas.

* 1. **Función: todos-ceros-secante**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: f: función de la que se desea encontrar un cero

tol-abs: tolerancia para convergencia

max-iter: máximo número de iteraciones

pares-semillas: pares de semillas con las que invocar a secante

Salida: Lista con todos los ceros encontrados a partir de las semillas

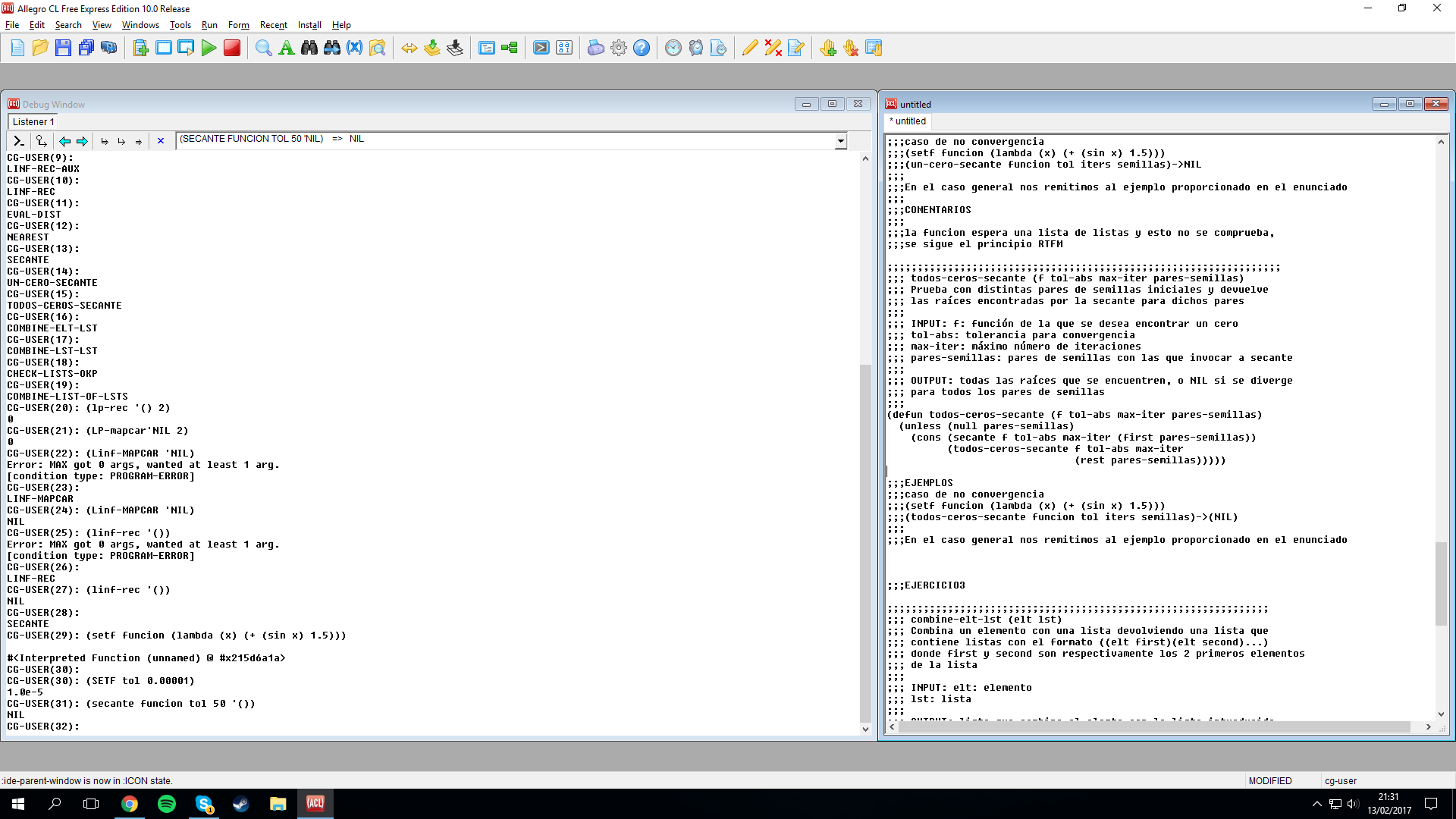
Procesamiento:

si no (vacío (pares-semillas)

secante(primer-elemento(pares-semillas))

todos-ceros-secante(siguientes-elementos(pares-semillas))

**CÓDIGO**



**COMENTARIOS**

Se espera que se conozcan las semillas previamente, ya que la función no calcula las semillas ni los intervalos donde se encuentran las raíces, simplemente las evalúa y devuelve la lista de resultados.

1. **Combinación de listas**
   1. **Combinación elemento-lista**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: elt: elemento

lst: lista

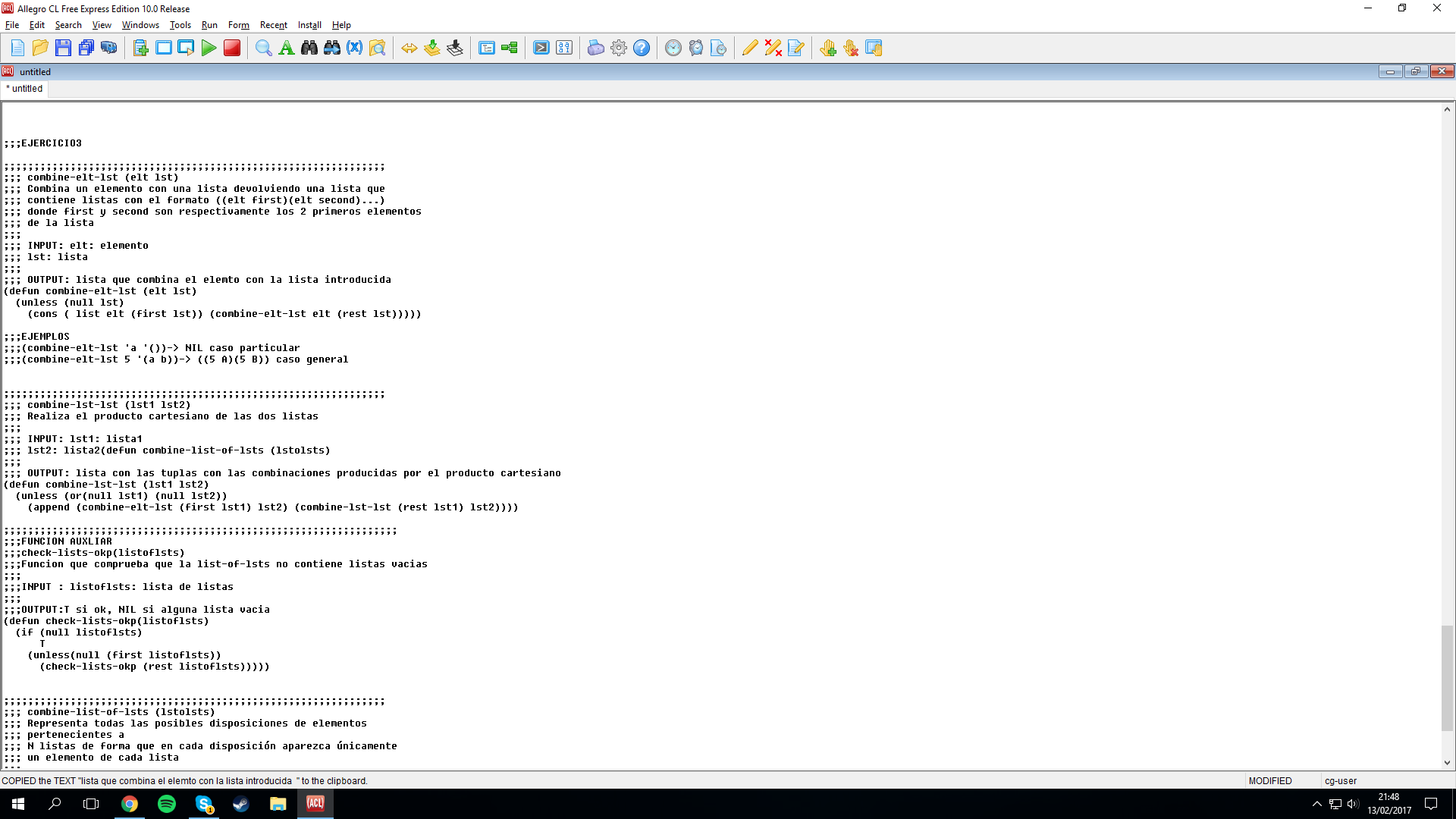
Salida: lista que combina el elemento con la lista introducida

Procesamiento:

Si no ( null(lst))

par( lista(elt, primer-elemento(lst)), combine-elt-lst(siguientes-elementos(lst))

**CÓDIGO**



**COMENTARIOS**

Debido a que no se especifica en el enunciado se admite el caso en el que el elemento es la lista vacía.

* 1. **Combinación lista-lista**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: lst1: lista1

lst2: lista2

Salida: lista que realiza el producto vectorial de las dos listas entrantes

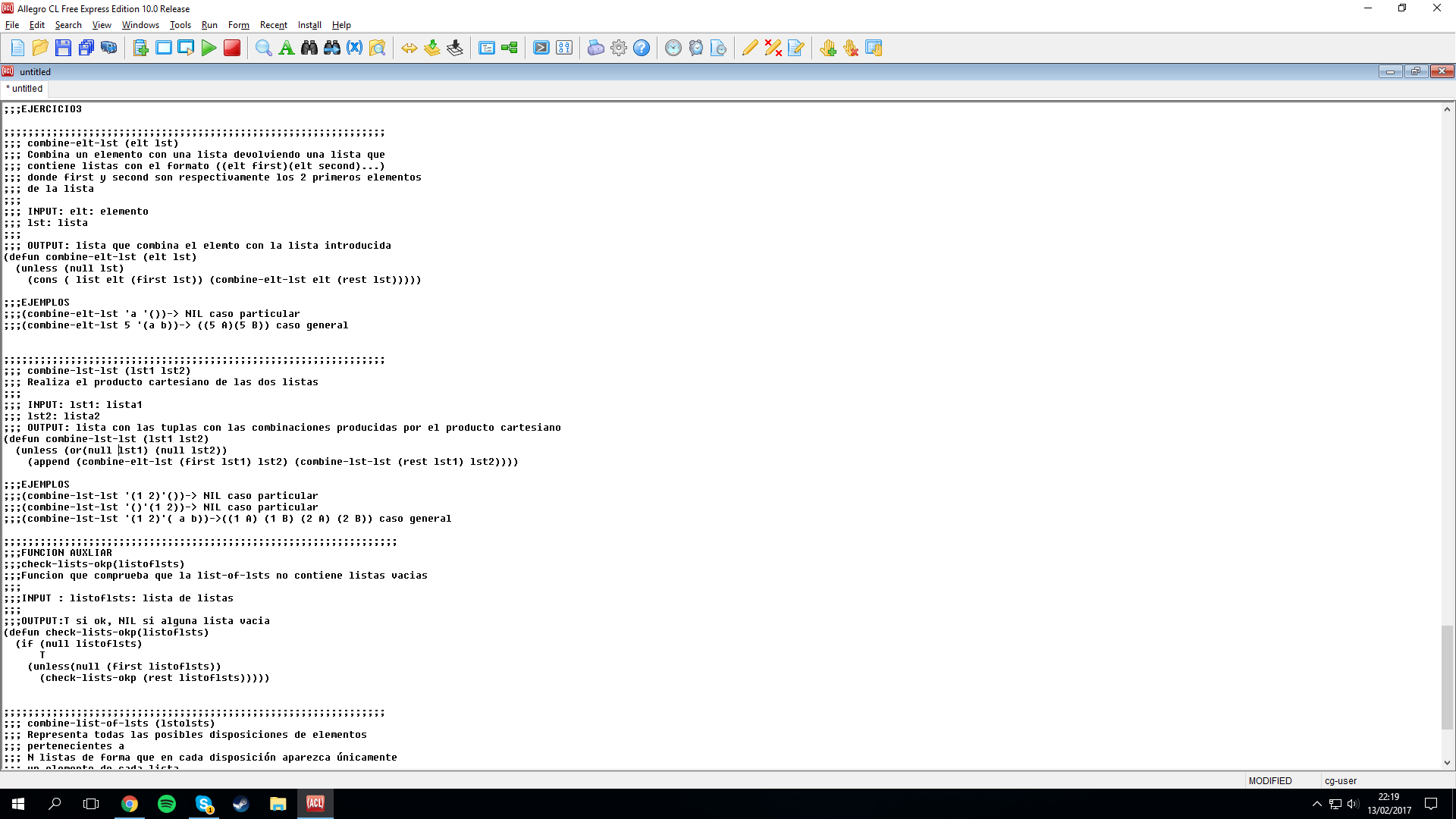
Procesamiento:

sino( null(lst1) o null(lst2))

une((combine-elt-lst(primer-elemento(lst1), lst2),

combine-elt-lst(siguientes-elementos(lst1),lst2))

**CÓDIGO**



* 1. **Combinación lista de listas**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**PSEUDOCÓDIGO**

Entrada: lstolsts: lista de listas con las que realizaremos las disposiciones

Salida: lista con las diferentes disposiciones producidas por la función

Procesamiento:

(si formato-okp(lstolsts))

l1=primer-elemento(lstolsts)

lr= siguientes-elementos(lstolsts)

primero= primer-elemento(l1)

combinar-elemento-lista(primero, lr)

combinar-lst-of-lsts(lista(siguientes-elementos(l1),lr))

**CÓDIGO**

