

Proyecto N°2

Lógica para Ciencias de la Computación

Lisp

Carlos Ignacio Loza
Javier Fernández Tierno

Indice

E1: Matriz Transpuesta.....	3
Funcionalidad.....	3
Estrategia de resolución.....	3
El código.....	4
Ejemplos de ejecuciones.....	6
E2: Suma de Primos.....	7
Funcionalidad.....	7
Estrategia de resolución.....	7
El código.....	7
Ejemplos de ejecuciones.....	9
E3: Permutaciones en orden lexicográfico.....	10
Funcionalidad.....	10
Estrategia de resolución.....	10
El código.....	11
Ejemplos de ejecuciones.....	14

E1: Matriz Transpuesta

Funcionalidad

Supongamos la matriz M de n filas y m columnas, notada genéricamente como:

$$M = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix}$$

Se representará en notación de LISP como lista de lista, donde la lista principal representa la matriz y cada “sublista” corresponderá a cada fila de la matriz; esto se reescribe como:

$$M = ((L1_1, L1_2 \dots L1_n) (L2_1, L2_2 \dots L2_n) \dots (LM_1, LM_2 \dots LM_n))$$

Luego el resultado M^t de hacer la transpuesta de la matriz M es:

$$M^t = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

Y resultara en notación de LISP como:

$$M^t = ((L1_1, L2_1 \dots LM_1) (L1_2, L2_2 \dots LM_2) \dots (L1_n, L2_n \dots LM_n))$$

Estrategia de resolución

La resolución de este ejercicio es simple y su comprensión es igual de fácil.

Partiendo de la idea de que en LISP notaremos la matriz como lista de lista, y como dijimos, cada sublista sera una fila de la matriz; crearemos una nueva lista de lista, donde tomaremos las cabezas de las sublistas originales, es decir una columna, y formaremos con estos elementos una nueva lista que será una fila de la matriz transpuesta, luego con las colas restantes y siendo estas las nuevas listas, llamaremos nuevamente a este procesos para formar una nueva fila con las nuevas cabezas; así el proceso se repite recursivamente hasta que las colas resulten vacías.

Llegado el punto en que se cumple el caso base, habremos desglosado la matriz original formando todas las listas que representan filas de la matriz deseada, luego pondremos estas listas en una lista y obtendremos la matriz transpuesta.

El código

Para la resolución del problema de obtener la matriz traspuesta y en consecuencia al algoritmo planteado, tenemos 3 funciones principales que dan la funcionalidad para obtener la matriz en cuestión.

En primer lugar mencionaremos las funciones auxiliares. por un lado la función “cars”, que, análoga que “CAR” que ya viene predefinida en LISP que extrae la cabeza de una lista, esta función “cars” retorna una lista con las cabezas de las listas, para nuestro caso sera una columna de la matriz

```
(DEFUN cars (M)
  (COND
    ((NULL M) NIL)
    (T(CONS (CAR (CAR M)) (cars (CDR M)))))
)
```

La segunda función auxiliar es “cdrs”, similar al caso anterior, acá hacemos otra analogía con “CAR” pero en este caso con “CDR”, para retornar una lista de listas, que en particular corresponden a las colas de las listas, es decir las listas sin las cabezas, y que estas listas de listas corresponden a la matriz sin la primer columna

```
(DEFUN cdrs (M)
  (COND
    ((NULL M) NIL)
    (T(CONS (CDR (CAR M)) (cdrs (CDR M)))))
)
```

Luego tenemos la función “trans”, que es la función principal, recursiva y de aridad 1 que recibe la matriz:

```
(DEFUN transC (M)
  (COND
    ((NULL M) NIL)
    ((NULL (CAR M)) NIL)
    (T (CONS (cars M) (transC (cdrs M)))))
)
```

Esta función es la que va uniendo la lista de cabezas que obtiene de la función “cars”, es decir obtiene la primer columna de la matriz, y llama recursivamente a “trans” con las colas que obtiene de la función “cdrs”, es decir aplica “trans” al resto de la matriz. Notar que la función se llama “transC” ya que posee una cascara llamada “trans” que realiza algunos chequeos y lo hace abstraer al cliente de dichos controles:

Hablando ahora de “trans” mencionamos que esta cascara realiza algunos controles, como ver que el parámetro ingresado sea una matriz, esto es:

- Que sea una lista (usando “listp”)
- Que sea una lista de listas (aplicando “listp” a “CAR” de M)
- Y que efectivamente sea una matriz, es decir todas listas del mismo tamaño (usando “esMatrix”)

En el caso que estos requerimientos no se cumplan lanza un cartel de error que advierte al usuario de un eventual error. Definimos la cascara como:

```
(DEFUN trans (M)
  (COND
    ((AND (listp M) (listp(CAR M)) (esMatrix M))
      (transC M)
    )
    (T
      (print "El parametro ingresado no tiene forma de Matriz.")
    )
  )
)
```

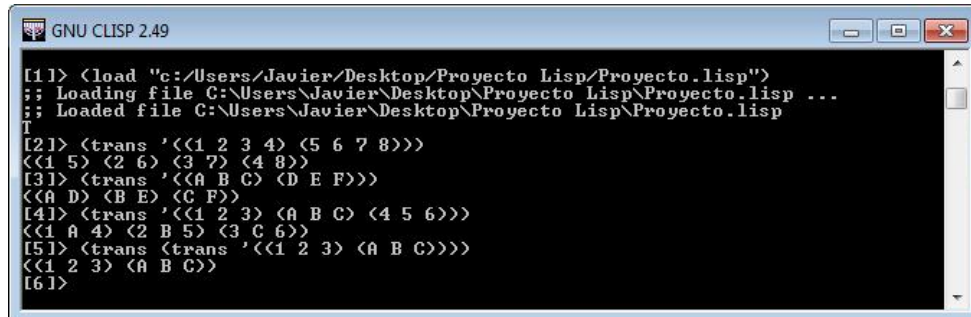
Si bien “listp” y “CAR” son funciones predefinida, hemos definido nuestra función de control “esMatrix” que simplemente calcula el largo de la primer lista y luego corrobora que el resto de las listas sean del mismo largo:

```
;Funcion esMatrix Cascara
(DEFUN esMatrix(M)
  (esMatrixC (Longitud(CAR M)) M)
)

;Funcion esMatrix
(DEFUN esMatrixC (N M)
  (COND
    ((NULL M)
      T
    )
    ((AND (= N (Longitud (CAR M))) (esMatrixC N (CDR M)))
      T
    )
    (T NIL)
  )
)
```

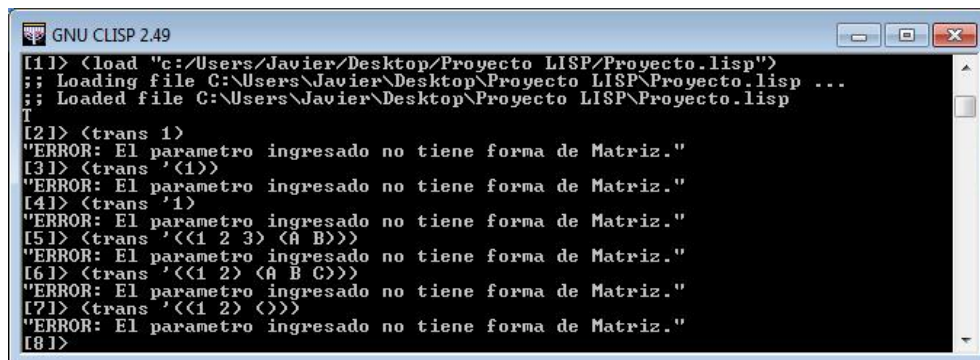
Ejemplos de ejecuciones

1. Ejemplos de una correcta ejecución:



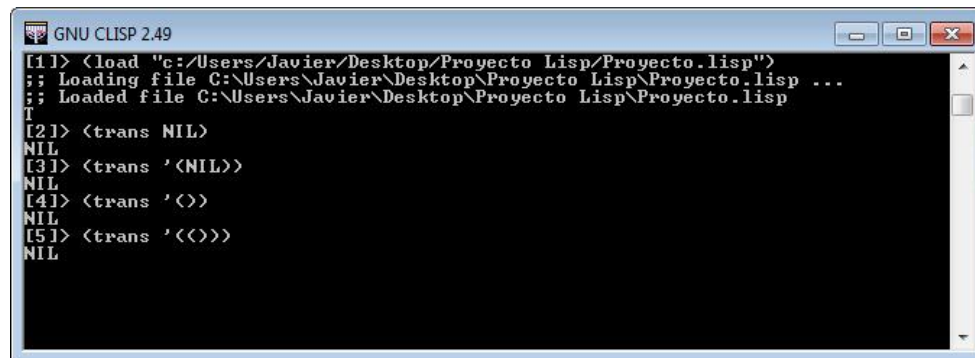
```
GNU CLISP 2.49
[1]> (load "c:/Users/Javier/Desktop/Proyecto Lisp/Proyecto.lisp")
;; Loading file C:\Users\Javier\Desktop\Proyecto Lisp\Proyecto.lisp ...
;; Loaded file C:\Users\Javier\Desktop\Proyecto Lisp\Proyecto.lisp
T
[2]> (trans '(1 2 3 4) (5 6 7 8)))
<(1 5) (2 6) (3 7) (4 8)>
[3]> (trans '(A B C) (D E F)))
<(A D) (B E) (C F)>
[4]> (trans '(1 2 3) (A B C) (4 5 6)))
<(1 A 4) (2 B 5) (3 C 6)>
[5]> (trans (trans '(1 2 3) (A B C)))
<(1 2 3) (A B C)>
[6]>
```

2. Ejemplos de finalización con entradas incorrectas:



```
GNU CLISP 2.49
[1]> (load "c:/Users/Javier/Desktop/Proyecto LISP/Proyecto.lisp")
;; Loading file C:\Users\Javier\Desktop\Proyecto LISP\Proyecto.lisp ...
;; Loaded file C:\Users\Javier\Desktop\Proyecto LISP\Proyecto.lisp
T
[2]> (trans 1)
"ERROR: El parametro ingresado no tiene forma de Matriz."
[3]> (trans '1)
"ERROR: El parametro ingresado no tiene forma de Matriz."
[4]> (trans '1)
"ERROR: El parametro ingresado no tiene forma de Matriz."
[5]> (trans '(1 2 3) (A B)))
"ERROR: El parametro ingresado no tiene forma de Matriz."
[6]> (trans '(1 2) (A B C)))
"ERROR: El parametro ingresado no tiene forma de Matriz."
[7]> (trans '(1 2) ()))
"ERROR: El parametro ingresado no tiene forma de Matriz."
[8]>
```

3. Ejemplos de casos base:



```
GNU CLISP 2.49
[1]> (load "c:/Users/Javier/Desktop/Proyecto Lisp/Proyecto.lisp")
;; Loading file C:\Users\Javier\Desktop\Proyecto Lisp\Proyecto.lisp ...
;; Loaded file C:\Users\Javier\Desktop\Proyecto Lisp\Proyecto.lisp
T
[2]> (trans NIL)
NIL
[3]> (trans '(NIL))
NIL
[4]> (trans '())
NIL
[5]> (trans '(<>>))
NIL
```

E2: Suma de Primos

Funcionalidad

Sea un numero natural N , se define la suma de primos como:

$$\sum e_i, \quad 1 < e_i < n : e_i \text{ es primo.}$$

Donde e_i es un número primo entre 1 y N ; por lo que la sumatoria toma todos estos números primos menor a N y retorna la sumatoria de ellos

Estrategia de resolución

Suma de primos es una función simple, y en simples palabras verifica si N es primo, si lo es, suma N a la sumatoria de primos para $N-1$, caso contrario sumará 0 a la sumatoria. El proceso termina cuando se calcula la sumatoria de primos para uno lo cual es cero.

El código

La sumatoria de primos es una función sencilla que consta de dos funciones, por un lado una función “sumaPrimos” de aridad 1 que calcula la suma de primos para un número N ingresado por parámetro.

En primer lugar tenemos la función compuesta “esPrimo” que retorna N (ingresado por parámetro) si es primo y 0 en caso de que N tenga divisores. Para ello “esPrimo” hace uso de la función “RecPrimo” que comienza con un numero de control M , desde 2, a buscar los divisores hasta llegar a N , en caso de encontrar un divisor de entre 2 y N retorna cero, caso contrario no encontrará divisores y eventualmente llegara el punto que M es igual N , en este caso se tiene certeza de que n es primo y sera retornado

```

(DEFUN esPrimo (N)
  (COND((= N 1) 0)
        ((> N 1) (RecPrimo N 2))
  )
)

;Funcion RecPrimo
(DEFUN RecPrimo (N M)
  (COND ((= M N) N)
        ((< M N) (
          COND((= 0 (MOD N M)) 0)
              (T
               (RecPrimo N (+ 1 M))
              )
        )
  )
)

```

Luego hacemos la función principal “sumaPrimos” que comienza desde N, le aplica la función “esPrimo” y su resultado lo suma a “sumaPrimos” para N - 1, como ya se tiene certeza de que “esPrimo” retorna N si N es primo o 0 en caso que no lo sea, la suma de “esPrimo” par N y “sumaPrimos” para N-1 es valida.

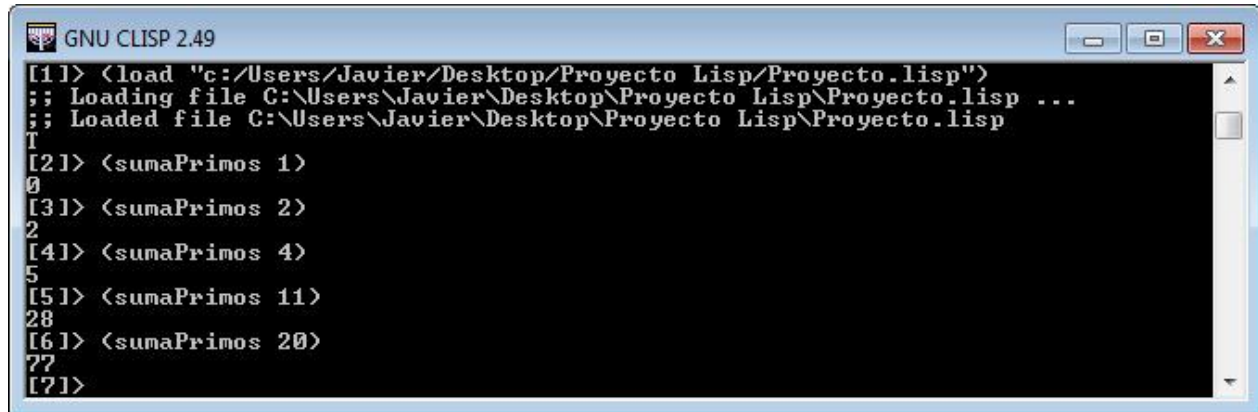
```

(DEFUN sumaPrimos (N)
  (COND
    ((numberp N)(COND
      ((< N 1) (print "N no es un numero positivo."))
      ((= N 1) 0)
      ((> N 1) (+ (esPrimo N) (sumaPrimos(- N 1)) ))
    )
  )
  (T (print "N no es un numero natural.") )
)
]

```

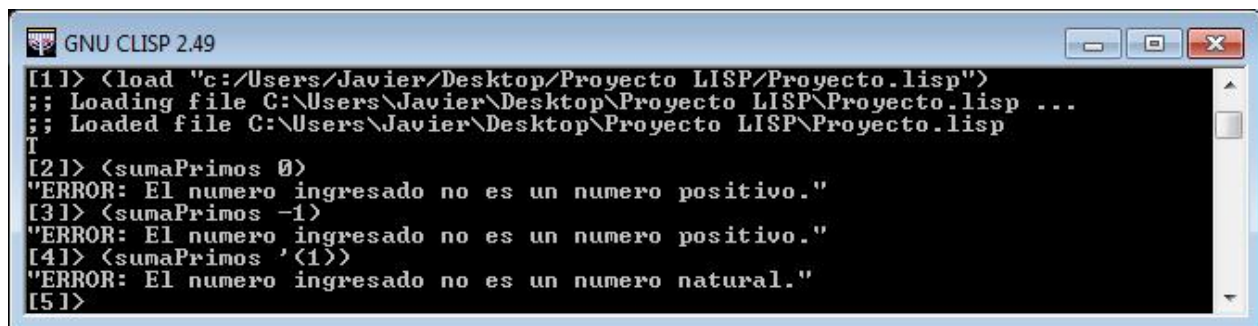

Ejemplos de ejecuciones

1. Ejemplos de una correcta ejecución:



```
GNU CLISP 2.49
[1]> <load "c:/Users/Javier/Desktop/Proyecto Lisp/Proyecto.lisp">
;; Loading file C:\Users\Javier\Desktop\Proyecto Lisp\Proyecto.lisp ...
;; Loaded file C:\Users\Javier\Desktop\Proyecto Lisp\Proyecto.lisp
T
[2]> <sumaPrimos 1>
0
[3]> <sumaPrimos 2>
2
[4]> <sumaPrimos 4>
5
[5]> <sumaPrimos 11>
28
[6]> <sumaPrimos 20>
77
[7]>
```

2. Ejemplos de finalización ante una entrada incorrecta:



```
GNU CLISP 2.49
[1]> <load "c:/Users/Javier/Desktop/Proyecto LISP/Proyecto.lisp">
;; Loading file C:\Users\Javier\Desktop\Proyecto LISP\Proyecto.lisp ...
;; Loaded file C:\Users\Javier\Desktop\Proyecto LISP\Proyecto.lisp
T
[2]> <sumaPrimos 0>
"ERROR: El numero ingresado no es un numero positivo."
[3]> <sumaPrimos -1>
"ERROR: El numero ingresado no es un numero positivo."
[4]> <sumaPrimos '(1)>
"ERROR: El numero ingresado no es un numero natural."
[5]>
```

E3: Permutaciones en orden lexicográfico

Funcionalidad

Sea una lista L de letras E_i denotada en LISP como:

$$L = (E_1, E_2, E_3, \dots, E_n)$$

Donde el resultado, a modo simplificado, de hacer las permutaciones en orden lexicográfico de una lista con tres elementos será:

$$L = ((E_1, E_2, E_3), (E_1, E_3, E_2), (E_2, E_1, E_3), (E_2, E_3, E_1), (E_3, E_1, E_2), (E_3, E_2, E_1))$$

Estrategia de resolución

La idea de resolución de estas permutaciones lexicográficas a priori parece un poco rebuscado, pero en el fondo es fácil de comprender si prestamos atención a la secuencia de pasos a seguir.

Para realizar todas las permutaciones debemos de tener presente la lista que ingresa por parámetro, tomaremos uno de sus elementos (por una cuestión de orden comenzaremos por el primero elemento) lo retiramos y obtenemos las permutaciones con los elementos restantes, es decir, la lista luego de eliminar el elemento seleccionado. Obtenidas estas permutaciones colocamos el elemento antes apartado como cabeza de todas estas permutaciones.

Volvemos a la lista original que ingresó inicialmente (notar que en este paso tomamos la lista original, es decir el primer elemento no fue apartado), esta vez tomamos el segundo elemento y pedimos las permutaciones del nuevo resto. Nuevamente tomamos el elemento apartado, en este caso el segundo, lo concatenamos a este nuevo conjunto de permutaciones obtenido y el resultado lo juntamos con las permutaciones que obtuvimos en el paso anterior.

El procesos de apartar un elemento, obtener las permutaciones del resto, y concatenar el elemento como cabeza, se repite hasta haberlo hecho para todos los elementos de la lista original. Este sera el primer caso base, cuando no queden elementos para apartar.

Notar ademas, que luego de apartar un elemento y pedir las permutaciones con el resto, la lista que ingresa en ese instante, tiene un elemento menos que la lista original ingresada, es decir suponiendo que la lista ingresada tiene N elementos, o una longitud de N , en la llamada recursiva la lista tendrá $N-1$. Esto nos lleva a que eventualmente la lista tendrá dos elementos, este será nuestro segundo caso base

ya que las permutaciones posibles para una lista con dos elementos no es mas que la lista original y su inverso.

Dejar bien en claro que los dos casos base no son excluyentes, sino mas bien, complementarios. Es decir, para obtener todas las permutaciones se llegara a ambos casos

El código

Para resolver el algoritmo antes planteado contamos con una serie de funciones, pero siendo principal la función “permuta2”, pero antes de explicarla, quizás sea conveniente explicar las funciones auxiliares de las cuales se hace uso.

La funciones auxiliares son sencillas, quizás un poco triviales, pero no así menos importantes, en primer lugar tenemos la función “Longitud” que recibe una lista y recursivamente va contando los elementos y retorna la cantidad de elementos que posee la lista:

```
(DEFUN Longitud (L)
  (COND
    ((NULL L) 0)
    (T (+ 1 (Longitud (CDR L)))))
)
```

En segundo lugar, tenemos la función “BorrarElemento” que recibe un elemento E y una lista L, y borrar ese elemento de la lista. Su funcionamiento es sencillo, compara la cabeza, si es el elemento E ingresado, retorno la cola de la lista, sino concateno esa cabeza con la lista obtenida de eliminar E en la cola de L:

```
(DEFUN BorrarElemento(E L)
  (COND
    ((= (Longitud L) 0)
     NIL)
    ((EQUAL (CAR L) E)
     (CDR L))
    (T (CONS (CAR L) (BorrarElemento E (CDR L)))))
)
```

Otra función trivial es “derechoInverso” que recibe una lista L y retorna una lista con dos listas, la primera sera la lista original L, la segunda será la lista con la cola de L

sucedida de la cabeza de L, esta función tiene sentido si ingresa una lista con dos elementos, por lo que retornaría dos listas, una al derecho y la otra al inverso:

```
(DEFUN derechoInverso(L)
  (LIST L (APPEND (CDR L) (LIST (CAR L))))
)
```

Por ultima función auxiliar se define “ponerPrimero” que recibe un elemento E y una lista de listas L, y agrega a E como cabeza de todas las sublistas de L, esto es, recursivamente concatena E con L y luego agrega a E como cabeza al resto de listas:

```
(DEFUN ponerPrimero(E L)
  (COND
    ((> (Longitud L) 0)
     (APPEND (LIST (CONS E (CAR L))) (ponerPrimero E (CDR L)) )
    )
    (T NIL)
  )
)
```

Volviendo a la función estrella, describimos a “permuta2”, de aridad 2 y que recibe 2 listas, la primera sera una lista de control, y la segunda sera la lista de la que desearemos obtener las permutaciones. En otras palabras mas claras, la primer lista sera de donde obtendremos los “elementos a apartar”, y en la segunda lista eliminaremos este elemento apartado para obtener el “resto”. Esta función tiene sentido si inicialmente recibe la misma lista en ambos parámetros. Como podemos ver en el siguiente código el caso base consulta si la longitud es igual a 2 (este seria el segundo caso base antes mencionado), en esta instancia hará uso de la función auxiliar “derechoInverso”, y en el caso general, concatena las permutaciones con cabeza del primer elemento y las permutaciones con cabezas los siguientes elementos:

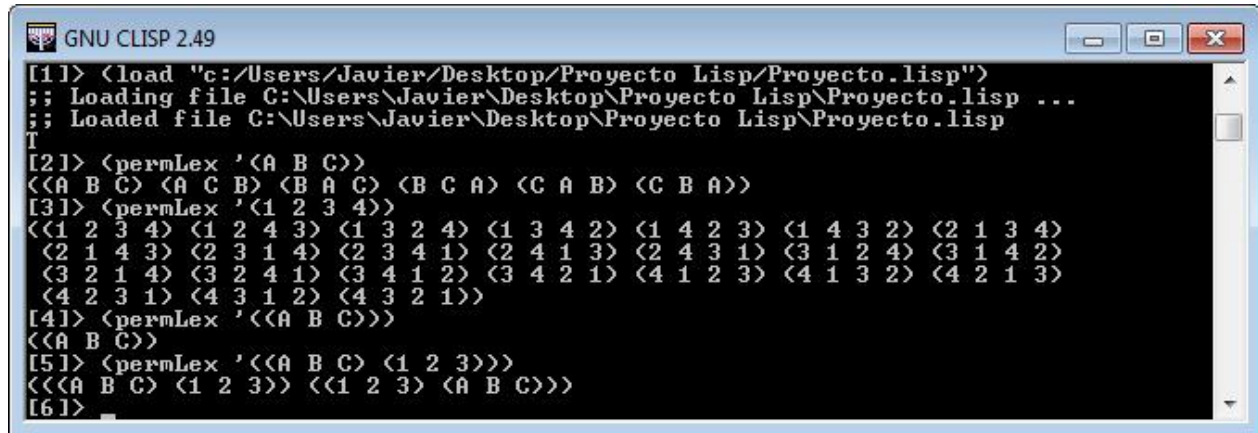
```
(DEFUN permuta2(L1 L2)
  (COND
    ((=(Longitud L2) 2)
     (derechoInverso L2)
    )
    (T
     (COND
       ((> (Longitud L1) 0)
        (APPEND (ponerPrimero (CAR L1)(permLex (BorrarElemento (CAR L1) L2))) (permuta2 (CDR L1) L2))
       )
       (T NIL)
     )
    )
  )
)
```

Como detalle en esta implementación, se puede notar que en la llamada recursiva se llama a la función “permLex”, esta llamada es la que recibe la lista de longitud N-1 que mencionamos anteriormente en el algoritmo, esta función “permLex” es una cascara que abstrae al usuario de llamar con la lista L por duplicado, además se encargada de controlar que el parámetro ingresado sea una lista, y atrapa dos casos particulares donde no hay calculo alguno, como ser la lista vacía y la lista con un elemento, que las únicas permutaciones posibles son resulta en la misma lista siendo este el resultado que devuelve

```
(DEFUN permLex (L)
  (COND
    ((listp L)
      (COND
        (((<(Longitud L) 2)
          L
        )
        (T
          (permuta2 L L)
        )
      )
    )
    (T
      (print "L no es una Lista de elementos.")
    )
  )
)
```

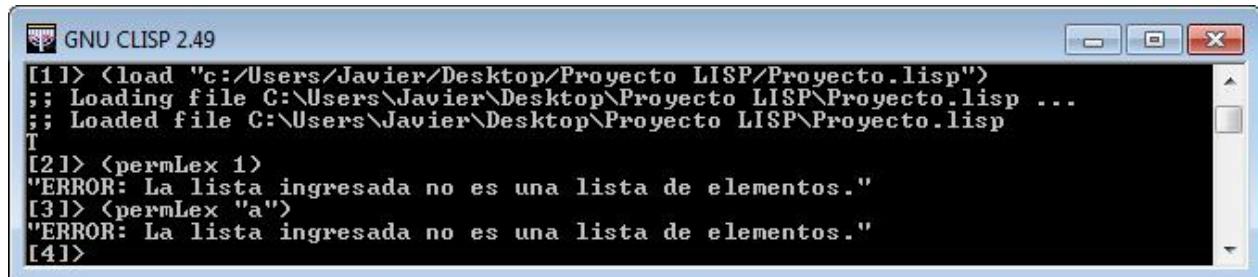
Ejemplos de ejecuciones

1. Ejemplos de una correcta ejecución:



```
GNU CLISP 2.49
[1]> <load "c:/Users/Javier/Desktop/Proyecto LISP/Proyecto.lisp">
;; Loading file C:\Users\Javier\Desktop\Proyecto LISP\Proyecto.lisp ...
;; Loaded file C:\Users\Javier\Desktop\Proyecto LISP\Proyecto.lisp
T
[2]> <permLex '<A B C>>
<<A B C> <A C B> <B A C> <B C A> <C A B> <C B A>>
[3]> <permLex '<1 2 3 4>>
<<1 2 3 4> <1 2 4 3> <1 3 2 4> <1 3 4 2> <1 4 2 3> <1 4 3 2> <2 1 3 4>
<2 1 4 3> <2 3 1 4> <2 3 4 1> <2 4 1 3> <2 4 3 1> <3 1 2 4> <3 1 4 2>
<3 2 1 4> <3 2 4 1> <3 4 1 2> <3 4 2 1> <4 1 2 3> <4 1 3 2> <4 2 1 3>
<4 2 3 1> <4 3 1 2> <4 3 2 1>>
[4]> <permLex '<<A B C>>
<<A B C>>
[5]> <permLex '<<A B C> <1 2 3>>>
<<<A B C> <1 2 3>> <<1 2 3> <A B C>>>
[6]>
```

2. Ejemplos de finalización ante una entrada incorrecta:



```
GNU CLISP 2.49
[1]> <load "c:/Users/Javier/Desktop/Proyecto LISP/Proyecto.lisp">
;; Loading file C:\Users\Javier\Desktop\Proyecto LISP\Proyecto.lisp ...
;; Loaded file C:\Users\Javier\Desktop\Proyecto LISP\Proyecto.lisp
T
[2]> <permLex 1>
"ERROR: La lista ingresada no es una lista de elementos."
[3]> <permLex "a">
"ERROR: La lista ingresada no es una lista de elementos."
[4]>
```