LÓGICA PARA CIENCIAS DE LA COMPUTACIÓN – 1ER CUATRIMESTRE 2019

# Programación Funcional en LISP

Proyecto N°2

**Comisión 24**

Andrade Sergio – LU 114059

Zarate Tomas – LU 111365

Profesor: Dr. Falappa, Marcelo

Asistente: Dr. Gomez L., Mauro

**Consideraciones del programa:**

1. Dado que LISP no es un lenguaje *case sensitive* se toma por convención el uso de mayúsculas sobre funciones predefinidas de LISP, ej: DEFUN, REM, LIST-LENGTH, etc.; el uso de una única letra mayúscula o una letra mayúscula y un número sobre las variables, ej: L1, M, N, etc.; y finalmente el uso de letras minúsculas al comienzo de cada una de las funciones definidas por los desarrolladores, ej: isPrime, matrix, addToAll, etc.
2. Se considera como representación valida de una matriz en LISP, una lista M compuesta por listas Mi, donde cada Mi denota una fila de la matriz.
3. Se espera que las funciones auxiliares sean accedidas únicamente por las funciones del programa que las necesitan y no por el usuario. Estas funciones son consideradas como “privadas” y no poseen la robustez necesaria para ser operadas por un usuario. De esta manera las únicas funciones accesibles por el usuario son “(trans M) (sumaPrimos N) (permLex L)”
4. En este programa se priorizó la robustez por encima de la eficiencia algorítmica, si bien se buscó reducir al máximo el tiempo de ejecución de las funciones implementadas.

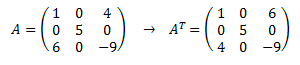
**Funcionalidad y estrategias implementadas:**

* Función 1: trans

La función recibe como argumento una matriz M­ y retorna la transpuesta de M.

Dada una matriz

Ej:



Sea Mt su matriz transpuesta, donde el elemento aij de la matriz M, se convertirá en el elemento aji de la matriz transpuesta Mt.

Para el cálculo de la matriz transpuesta se considera el siguiente planteo recursivo, sea M una representación valida de una matriz en LISP, bajo la convención adoptada para este trabajo:

* Caso Base: si M es vacía la transpuesta de una matriz vacía es una matriz vacía
* Caso Recursivo: si no, la transpuesta de M es la concatenación de la primera fila transpuesta, con la transpuesta de M', siendo M' M sin el primer elemento de cada fila

[INSERTAR FOTO CODIGO]

*Funciones Auxiliares:*

* **(compRow M):** esta función toma
* Función 2: **sumaPrimos**

La función recibe como argumento un numero entero N y retorna como resultado la suma de todos los números primos entre 0 y N.

*“Un****número primo****es un número natural mayor que 1 que tiene únicamente dos divisores distintos: él mismo y el 1.”*

La función cumple la funcionalidad del siguiente algoritmo, descrito mediante la función:

Donde la función *sumaPrimos(N)* recibe un número entero mayor o igual a 0.

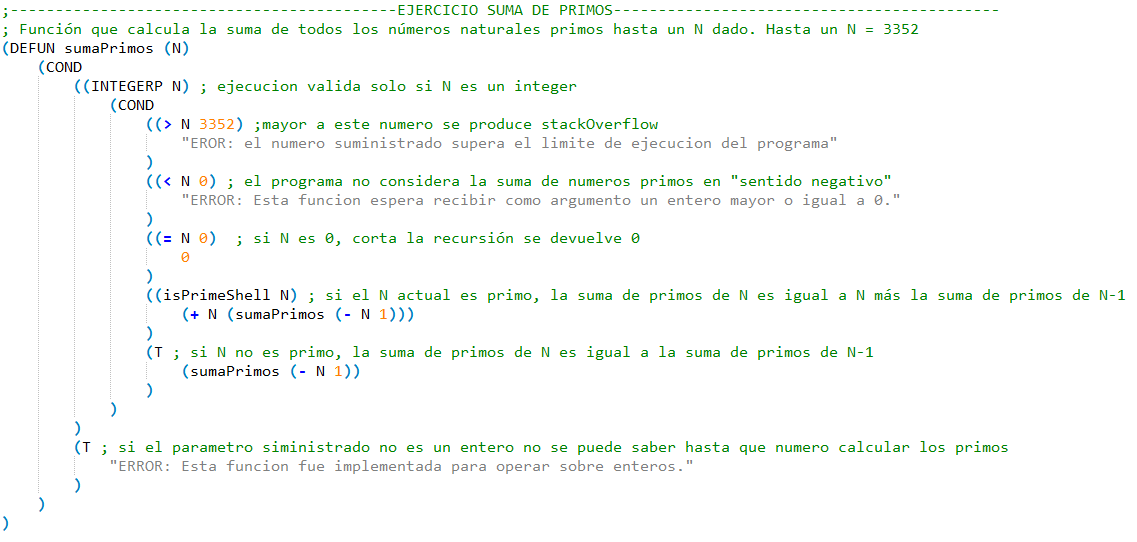
Se define la función *sumaPrimos(N)* y dos funciones auxiliares: *isPrimeShell(N)* y *isPrime(N B)*.

Explicación del algoritmo implementado en LISP:

*Función principal:*

* **sumaPrimos(N):**

Código LISP:

****

Verifica que la entrada sea un número entero entre 0 y 3352 (Si se ingresa este valor o uno mayor, se llena la pila de memoria al hacer tantas llamadas recursivas).

Estrategia:

Caso base: si **N=0**, la suma de los primos entre 0 y 0 es 0.

Caso recursivo: Si **N>0** y **N es primo**, la suma de todos los primos entre 0 y N, es N sumado a la suma de todos los primos entre 0 y N-1, de lo contrario, es la suma de todos los primos entre 0 y N.

Es decir:

Si N>0 y *isPrime(N)*,

*sumaPrimos(N)* es N + *sumaPrimos(N+1)*,

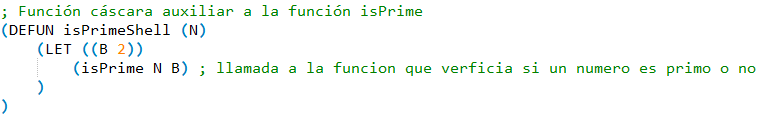
de lo contrario,

*sumaPrimos(N)* es *sumaPrimos(N-1)*.

*Funciones Auxiliares:*

* **isPrimeShell(N):**

Código LISP:

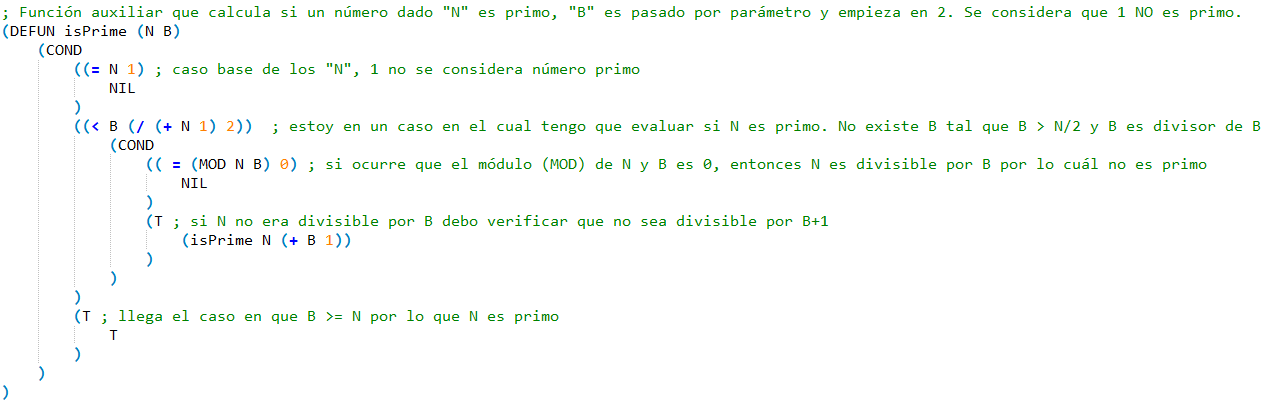
****

Recibe como argumento un entero N y devuelve como resultado si el entero ingresado es un numero primo. Su función es la de servir como función cáscara y llamar a la función *isPrime(N B)* con B = 2.

Se asume que el argumento recibido por parámetro es un número entero, es responsabilidad del usuario verificarlo.

* **isPrime(N B):**

Código LISP:

****

Recibe como argumento dos enteros N y B, y calcula si N no es divisible por algún entero entre B (Inicialmente 2) y N/2. Si N no es divisible por ningún entero en ese rango de valores, entonces N es primo, ya que N nunca puede ser divisible por un número mayor a N/2.

Primer caso: Si **N=1**, por definición, N no es primo.

Segundo caso:

Caso Base: Si B = (N + 1)/2 N tiene que ser primo, pues no encontré ningún número entero entre 0 y N/2 que pueda dividir a N sin dejar resto.

Caso recursivo: si B es **menor** a (N + 1)/2:

Si N es divisible por B, N no es primo.

De lo contrario, N es primo si *isPrime(N B+1)*, es decir, si no es divisible por B+1.

Se asume que el argumento N recibido por parámetro es un número entero, y que el argumento B se encuentra inicializado en 2 (Inicializado en la función *isPrimeShell(N))*, para el correcto funcionamiento del algoritmo, es responsabilidad del usuario verificarlo.

* Función 3: **permLex**

Dada una lista L de n elementos, la función permLex(L) retorna una lista de n! elementos con todas las permutaciones de esos n elementos, en orden lexicográfico. Se asume que la lista ingresada se encuentra ordenada lexicograficamente.

Ej.

(permLex (a b c)) = ((a b c) (a c b) (b a c) (b c a) (c a b) (c b a))

Estrategia planteada: para la resolución de este problema se decidió realizar una recursión cruzada entre la función permLex y la función permute. Sea L una lista con N elementos:

* Caso base (permLex): si N = 1, es decir, la lista posee un solo elemento, la permutación léxica de L es L misma.
* Caso Recursivo (permLex): en caso de que N > 1, la permutación léxica de L se calcula mediante la función (permute L L).

En la función (permute L1 L2) se toma la primera lista como los elementos que restan permutar y a L2 como una lista back up de la original. La permutación se realiza “dejando fijo” un elemento en la primera posición de la lista e intercambiando los lugares de los demás, así sucesivamente hasta haber realizado esta práctica con todos los elementos.

* Caso Base (permute): si L1 está vacía, no tengo más elementos que permutar y mi resultado es vacío.
* Caso Recursivo (permute): si L1 no es vacía se debe concatenar la cabeza de L1 a todas las permutaciones léxicas resultado de llamar a permLex con la lista original sin la cabeza de L1, para luego unir todas estas con las permutaciones de los restantes elementos en L1 y L2.

Para realizar lo descripto en el caso recursivo de permute, se utiliza a su vez la función reArrange, explicada en detalla más adelante, pero que, a grandes rasgos, reacomoda en L2 el elemento con el que ya se realizó la permutación léxica a modo de mantener un orden que permite seguir obteniendo las permutaciones de manera ordenada.

Explicación del algoritmo implementado en LISP:

*Función principal:*

* **permLex(L):**

Código LISP:

**[Insertar aca captura del codigo]**

Recibe como argumento una lista L, verifica que efectivamente sea una lista válida, y computa las permutaciones lexicográficas de L.

Caso Base: Si L es una lista de un único elemento, L es la única permutación posible.

Caso General: si L contiene mas de un elemento, llamo a la función auxiliar *permute(L L).*

*Funciones Auxiliares:*

* **permute(L1 L2):**

Código LISP:

**[Insertar aca captura del codigo]**

Recibe como argumentos dos listas L1 y L2, donde L1 son los elementos que restan por permutar, y L2 es la lista de la cual quiero obtener sus permutaciones.

Caso Base: Si L1 es lista vacía, no quedan elementos por permutar.

Caso General: Si L1 no es lista vacia, calculo la permutación lexica como la unión entre: el primer elemento de L1 seguido de todas las permutaciones

* **addToAll(E L):**

Código LISP:

**[Insertar aca captura del codigo]**

Recibe como argumentos un elemento E y una lista de listas L, y concatena el elemento E como cabeza de cada lista elemento de L.

Caso Base: Si L es lista vacía, no hay mas listas a las cuales unir E.

Caso Recursivo: Si L no es vacío, devuelvo la unión entre: la unión entre E y la primer lista de L, y el resto de uniones de E con L.

Es decir:

Si L es distinto de vacio

addToAll(E L)=Vacio

De lo contrario

addToAll(E L) = (E concatenado con CAR de L (Primera lista dentro de L)) concatenado con addtoAll(E CDR(L))

* **reArrange(E L I):**

Código LISP:

**[Insertar aca captura del código]**

Recibe como argumentos un elemento E, una lista L y un entero I, y retorna la lista resultante de insertar E en la posición I de la lista L. Esta función es fundamental ya que mantiene el orden lexicográfico al ir realizando las permutaciones. Si contamos al principio con la lista ( 1 2 3 4) nuestro algoritmo realiza las permutaciones con 1 en el primer lugar de la lista y luego con 2, con 3, y así sucesivamente, lo que hace esta función es reordenar la lista de manera lexicográfica considerando los elementos ya utilizados, así por ejemplo una vez utilizado el 1 la función lo “corre” para q este primero el 2 y así con los demás.

Caso Base: si I = 0, reArrange = E como cabeza de L.

Caso recursivo: Si I>0, reArrange es el resultado de insertar E en la posición I-1 del cuerpo de la lista L.