Programación Funcional en LISP

Proyecto N°2

Comisión 24

Andrade Sergio – LU 114059
Zarate Tomas – LU 111365

Profesor: Dr. Falappa, Marcelo
Asistente: Dr. Gómez L., Mauro

Table of Contents

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 M_i, donde cada M_i 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) Si bien se buscó reducir al máximo el tiempo de ejecución de las funciones implementadas, en este programa se priorizó la robustez por encima de la eficiencia algorítmica.

Funcionalidad y estrategias implementadas:

Función 1: trans

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

Dada una matriz $M = \begin{pmatrix} a11 & \cdots & a1j \\ \vdots & \ddots & \vdots \\ ai1 & \cdots & aij \end{pmatrix}$

Ej:

$$A = \begin{pmatrix} 1 & 0 & 4 \\ 0 & 5 & 0 \\ 6 & 0 & -9 \end{pmatrix} \rightarrow A^T = \begin{pmatrix} 1 & 0 & 6 \\ 0 & 5 & 0 \\ 4 & 0 & -9 \end{pmatrix}$$

Sea M^t su matriz transpuesta, donde el elemento a_{ij} de la matriz M, se convertirá en el elemento a_{ji} de la matriz transpuesta M^t.

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 M 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.

Casos de prueba considerados importantes:

trans de una Matriz
> (trans '((1 2 3 4) (5 6 7 8))
((1 5) (2 6) (3 7) (4 8))

trans de un vector
> (trans '((1 2 3 4))
((1) (2) (3) (4))

trans de una lista que no constituye una matriz
> (trans '((1 2 3 4) (5 6))
"ERROR: La lista ingresada no es una matriz."

trans de una lista vacía
> (trans '(()))
NIL

trans de un argumento que no constituye una matriz
> (trans 7)
"ERROR: Esta funcion fue implementada para operar sobre matrices representadas como lista de listas."

Función Principal:

- **trans(M):**

```
;-----------------------------------------EJERCICIO MATRIZ TRASPUESTA-----------------------------------------
; Funcion que computa la traspuesta de una matriz representada como una lista de listas
(DEFUN trans (M)
    (COND
        ((LISTP M) ; primero se verifica que M sea efectivamente una lista
            (COND
                ((OR (NOT (LISTP (CAR M)))(NOT (matrix M (LIST-LENGTH (CAR M))))) ; luego, que M sea una representacion valida de una matriz
                    "ERROR: La lista ingresada no es una matriz."
                )
                (T ; si es una matriz valida comienza la ejecucion
                    (COND
                        ((NULL (CAR M)) ; si M es vacia, la traspuesta de una matriz vacia es una matriz vacia
                            NIL
                        )
                        (T ; si no, la transpuesta de M es la concatenacion de la primer fila transpuesta con la transpuesta de M',
                            ;siendo M' M sin el primer elemento de cada fila
                            (CONS (compRow M) (trans (reduceMatrix M)))
                        )
                    )
                )
            )
        )
        (T ; en caso de que M no fuera una lista esta funcion no tiene sentido alguno
            "ERROR: Esta funcion fue implementada para operar sobre matrices representadas como lista de listas."
        )
    )
)
```

Dada una matriz M ingresada por parámetro, se computa la transpuesta de M.

La función verifica que la matriz haya sido ingresada en notación de LISP, esto es, una lista de listas, y verifica que cumpla la definición de matriz, llamando a la función matrix(M).

Estrategia:
Caso Base:
Si M es la matriz vacía, la transpuesta de M es la matriz vacía.
Caso Recursivo:
Si M no es vacía, la transpuesta de M es la primer columna de M, como primer fila de M^t, unido a la transpuesta de M', siendo M' M sin su primer columna. Obteniendo la primer columna mediante la llamada a la función compRow(M).

Esto es:
Si M=Vacio
trans(M) = Vacio
De lo contrario
trans(M) = compRow(M) concatenado con trans(CDR de M)

Funciones Auxiliares:

- **compRow(M):**

```
; Funcion auxiliar que permite obtener, a partir de una matriz M representada como lista de listas,
; una fila resultado de concatenar el primer elemento de cada fila de la matriz
(DEFUN compRow (M)
    (COND
        ((NULL M) ; si M es vacia, una fila traspuesta de una matriz vacia es una fila vacia
            NIL
        )
        (T ; si no, se concatena el primer elemento de la primer fila M,
            ;con la fila fila resultante de aplicar compRow sobre M', siendo M' el cuerpo de M
            (CONS (CAR (CAR M)) (compRow (CDR M)))
        )
    )
)
```

Esta función toma como argumento una matriz M y retorna una fila con el primer elemento de cada fila de M, esto es, la primer columna de M. Esto será, la primer fila de la matriz M^t.

Estrategia:

Caso Base:

Si M es la matriz vacía, su primer columna es la lista vacía.

Caso Recursivo:

Si M no es vacía, su primer columna es el primer elemento de la primera fila, unido a la primer columna de M' siendo M', M sin su primer fila.

Esto es:

Si M=Vacio

compRow(M) = vacio

De lo contrario

compRow(M) = CAR de CAR de M, unido a compRow(CDR de M).

- **reduceMatrix(M):**

```
; Funcion auxiliar que, dada una matriz M representada como lista de listas, elimina el primer elemento de cada fila
(DEFUN reduceMatrix (M)
    (COND
        ((NULL M) ; si M es vacia su reduccion es vacio
            NIL
        )
        (T ; si no, la reduccion de M es el resultado de concatenar la primer fila de M sin su primer elemento,
            ;con la reduccion de M', siendo M' el cuerpo de M
            (CONS (CDR (CAR M)) (reduceMatrix (CDR M)))
        )
    )
)
```

Dada una matriz M, retorna la matriz reducida M' siendo M' M sin el primer elemento de cada fila.

Estrategia:

Caso Base:

Si M es vacía, su matriz reducida es vacío.

Caso Recursivo:

Si M no es vacío, su reducción es la concatenación entre la primer fila de M sin su primer elemento, con la reducción de M sin su primer fila.

Es decir:

Si M = vacío

reduceMatrix(M) = Vacío

De lo contrario

reduceMatrix(M) =concatenar ((CDR de (CAR de M)) con reduceMatrix(CDR de M))

Donde CAR representa el primer elemento de una lista, y CDR el resto de elementos.

- **matrix(M I):**

```
; Funcion auxiliar que verifica que la matriz pasado por parametro cumple con la condicion de tener siempre
; el mismo numero de columnas en todas sus filas
; M es la matriz que se desea verificar
; I es la cantidad de columnas en la primer fila de M
(DEFUN matrix (M I)
    (COND
        ((= (LIST-LENGTH M) 1) ; si resta un unico elemento de en la matriz
            (COND
                ((= (LIST-LENGTH (CAR M)) I) ; se verifica que el largo de ese elemento coincide con el inicial
                    T
                )
                (T ; si no lo es, M no era una matriz en un principio
                    NIL
                )
            )
        )
        (T ; si hay mas de un unico elemento en M
            (COND
                ((= (LIST-LENGTH (CAR M)) I) ; en caso de que coincidan el largo de la primer fila y el largo de la cabeza,
                                             ;M sera una matriz <=> M', siendo M' M sin su primer fila, es una matriz
                    (matrix (CDR M) I)
                )
                (T ; si no, M no era una matriz
                    NIL
                )
            )
        )
    )
)
```

Función utilizada para verificar que una matriz M sea una matriz válida.
Recibe como argumento una matriz M y un entero I, y verifica que todas las filas de M tengan la longitud I.

Estrategia:

Caso Base:

Si M es una matriz de una única fila, M es una matriz valida de longitud de fila I, si la longitud de M es I.

Caso recursivo:

Si M tiene más de una fila, M es una matriz válida si la primera fila de M tiene logitud I, y M’ es una matriz válida, siendo M’ M sin su primer fila.

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:

$$sumaPrimos(N) = \begin{cases} 0 \, si \, N = 0 \\ N + sumaPrimos(N-1) \, si \, N > 0 \, y \, N \, es \, primo \\ sumaPrimos(N-1) \, si \, N > 0 \, y \, N \, no \, es \, primo \end{cases}$$

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)*.

Casos de prueba considerados importantes:

sumaPrimos de un entero primo
> (sumaPrimos 7)
17 ---> 7+5+3+2

sumaPrimos de un entero no primo
> (sumaPrimos 6)
10 ---> 5+3+2

sumaPrimos de 1
> (sumaPrimos 1)
0 pues 1 no se considera primo

sumaPrimos de 0
> (sumaPrimos 0)
0

sumaPrimos de un número negativo
> (sumaPrimos -7)
"ERROR: Esta funcion espera recibir como argumento un entero mayor o igual a 0."

sumaPrimos de un número racional
> (sumaPrimos 1.2)
"ERROR: Esta función fue implementada para operar sobre enteros."

sumaPrimos de un símbolo
> (sumaPrimos "$")
"ERROR: Esta función fue implementada para operar sobre enteros."

Explicación del algoritmo implementado en LISP:

Función principal:

- **sumaPrimos(N):**

  Código LISP:

```
;-----------------------------------------EJERCICIO SUMA DE PRIMOS-----------------------------------------
; Función que calcula la suma de todos los números naturales primos hasta un N dado. Hasta un N = 3352
(DEFUN sumaPrimos (N)
    (COND
        ((INTEGERP N) ; ejecucion valida solo si N es un integer
            (COND
                ((< N 0) ; el programa no considera la suma de numeros primos en "sentido negativo"
                    "ERROR: Esta funcion espera recibir como argumento un entero mayor o igual a 0."
                )
                ((= N 0)  ; si N es 0, corta la recursión se devuelve 0
                    0
                )
                ((isPrimeShell N) ; si el N actual es primo, la suma de primos de N es igual a N más la suma de primos de N-1
                    (+ N (sumaPrimos (- N 1)))
                )
                (T ; si N no es primo, la suma de primos de N es igual a la suma de primos de N-1
                    (sumaPrimos (- N 1))
                )
            )
        )
        (T ; si el parametro siministrado no es un entero no se puede saber hasta que numero calcular los primos
            "ERROR: Esta funcion fue implementada para operar sobre enteros."
        )
    )
)
```

  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:

```
; Función cáscara auxiliar a la función isPrime
(DEFUN isPrimeShell (N)
    (LET ((B 2))
        (isPrime N B) ; llamada a la funcion que verficia si un numero es primo o no
    )
)
```

  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:

```
; Función auxiliar que calcula si un número dado "N" es primo, "B" es pasado por parámetro y empieza en 2. Se considera que 1 NO es primo.
(DEFUN isPrime (N B)
    (COND
        ((= N 1) ; caso base de los "N", 1 no se considera número primo
            NIL
        )
        ((< B (/ (+ N 1) 2))  ; estoy en un caso en el cual tengo que evaluar si N es primo. No existe B tal que B > N/2 y B es divisor de B
            (COND
                (( = (MOD N B) 0) ; si ocurre que el módulo (MOD) de N y B es 0, entonces N es divisible por B por lo cuál no es primo
                    NIL
                )
                (T ; si N no era divisible por B debo verificar que no sea divisible por B+1
                    (isPrime N (+ B 1))
                )
            )
        )
        (T ; llega el caso en que B >= N por lo que N es primo
            T
        )
    )
)
```

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 conoce como una **permutación lexicográfica** al conjunto de permutaciones enlistadas numérica o alfabéticamente.

Se asume que la lista ingresada ya se encuentra ordenada lexicográficamente.

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.

Casos de prueba considerados importantes:

permLex de una lista con un único elemento
> (permLex ‘(a))
((a))

permLex de una lista con mas de un elemento
> (permLex ‘(a b))
((A B C) (A C B) (B A C) (B C A) (C A B) (C B A))

permLex de un argumento que no es una lista
> (permLex a)
"ERROR: Esta funcion espera recibir por parametro una lista."

Explicación del algoritmo implementado en LISP:

Función principal:

- **permLex(L):**

Código LISP:

```
;----------------------------------------EJERCICIO PERMUTACION LEXICA----------------------------------------
; Funcion que calcula la permutacion lexica de una lista L suministrada por parametro.
; Se asume que L se encuentra ordenada lexicograficamente
(DEFUN permLex (L)
    (COND
        ((LISTP L) ; verifico que L sea efectivamente una lista
            (COND
                ((= 1 (LIST-LENGTH L)) ; la unica permutacion posible de una lista con un unico elemento es esa misma lista
                    (LIST L)
                )
                (T
                    (permute L L) ; llamada a la funcion auxiliar que realiza la permutacion
                )
            )
        )
        (T
            "ERROR: Esta funcion espera recibir por parametro una lista."
        )
    )
```

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:

```
; Funcion que realiza la permutacion de una lista, llevando registro en otra de que elementos faltan permutar
; L1 es la lista de elementos que restan permutar
; L2 es la lista real con todos los elementos
(DEFUN permute (L1 L2)
    (COND
        ((= (LIST-LENGTH L1) 0) ; si no restan elementos por permutar el resultado de la permutacion es vacio
            NIL
        )
        (T  ; caso contrario la permutacion lexica de una lista L es el resultado de concatenar la cabeza de L
            ;a las permutaciones lexicas del cuerpo
            ;y formar una lista entre esto y las permutaciones lexicas del resto de L
            (APPEND
                (addToALL (CAR L1) (permLex (DELETE (CAR L1) L2)))
                (permute (CDR L1) (reArrange (CAR L1) (DELETE (CAR L1) L2) (- (LIST-LENGTH L2) (- (LIST-LENGTH L1) 1))))
            )
        )
    )
)
```

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 vacía, calculo la permutación léxica como la unión entre: el primer elemento de L1 seguido de todas las permutaciones

- **addToAll(E L):**

Código LISP:

```
; Funcion que, dada una lista de listas L, inserta el parametro E como cabeza de la misma
; devuelve una lista con las listas modificadas
(DEFUN addToALL (E L)
    (COND
        ((= (LIST-LENGTH L) 0) ; si no hay mas elementos en la lista finaliza la recursion
            NIL
        )
        (T ; se realiza la concatenacion de E a la primer lista, luego se llama recursivamente

            (CONS (CONS E (CAR L)) (addToALL E (CDR L)))
        )
    )
)
```

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:

```
; Funcion que, dado un elemento E y una lista L inserta el elemento en la posicion I de L
(DEFUN reArrange (E L I)
    (COND
        ((= I 0) ; si la I vale 0 encontre la posicion donde debo insertar E
            (CONS E L)
        )
        (T ; caso contrario, tomo por separado el cuerpo de L, reduzco I en 1, llamo recursivamente y vuelvo a concatenar
            (CONS (CAR L) (reArrange E (CDR L) (- I 1)))
        )
    )
)
```

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.