Lenguaje de Programación LISP

LISP (**Lis**t **P**rocessing) es un lenguaje diseñado para la manipulación de fórmulas simbólicas. Más adelante, nació su aplicación a la Inteligencia Artificial. La principal característica de LISP es su habilidad de expresar algoritmos recursivos que manipulen estructuras de datos dinámicos.
En LISP existen dos tipos básicos de datos, los átomos y las listas. Todas las estructuras definidas posteriormente son basadas en estos.

Átomo: es un elemento indivisible que tiene significado propio.

Ej.: `A , 54 , 3.14 , + , lunes , sol`
Los átomos `54` y `3.14` son átomos numéricos y los otros son átomos simbólicos.

Lista: consta de un paréntesis izquierdo, seguida por cero o más átomos o listas, y un paréntesis derecho. Cada componente de la lista se denomina elemento.

Ej.: `(a 23 (8 hola ( ) ) 3.1)` lista con 4 elementos

La lista nula tiene la característica de ser lista y átomo a la vez , y se representa por:
`( )` o bien `NIL`.

Los valores de verdad están representados por el átomo `T` y el átomo `NIL` que representan el verdadero y falso respectivamente.

LISP también tiene otros tipos de datos como ser caracteres, arreglos, cadenas y estructuras.

La notación matemática en LISP es prefija. Ej.: `(+ 2 3)` para sumar 2 + 3

Funciones primitivas

- Selectores de elementos de una lista

1) **`CAR`** o bien **`FIRST`** aplicado a una lista devuelve el primer elemento de la lista

```
Ej.:  (CAR '(1 2 3) )    →    1

      (FIRST '((a b) 8 (6)) )    →    (a b)
```

2) **`CDR`** o bien **`REST`** aplicado a una lista devuelve la lista original sin el primer elemento

```
Ej.:  (CDR '(1 2 3) )    →    (2 3)

      (REST '((a b) 8 (6)) )    →    (8 (6))
```

Como en las matemáticas las funciones pueden agruparse y una función puede ser parámetro de otra función, aplicándose desde la función mas interna hacia afuera.

Ej.: `(CAR (CDR (CDR '(a b c d) ) ) )` → `c`

Ej.: `(CADDR '(a b c d) )` → `c`

3) **NTH** aplicado a una lista y un número devuelve el elemento ubicado en la posición indicada por el número

Ej.: `(NTH 3 '(a b c d) )` → `d`

4) **NTHCDR** devuelve el n–ésimo CDR de una lista a partir del numero indicado

Ej.: `(NTHCDR 1 '(a b c) )` → `(b c)`

- Función QUOTE

La función **QUOTE** evita la evaluación de su argumento devolviéndolo sin evaluar.

Ej.: `(QUOTE (a b c) )` → `(a b c)`

Se puede usar el apóstrofo como abreviatura para **QUOTE** . Por lo tanto la expresión que sigue es equivalente a la anterior.

Ej.: `'(a b c)` → `(a b c)`

El hecho de proporcionar el apóstrofo como una abreviatura para **QUOTE**, se considera una manera de *refinamiento sintáctico*.

- Constructores de listas

1) **(CONS** <obj> <lista>**)** devuelve una nueva lista donde el primer elemento es el objeto y los elementos restantes son los de la lista original

Ej.: `(CONS 'a '(1 2 3) )` → `(a 1 2 3)`

2) **(APPEND** <lista> <lista>**)** devuelve una nueva lista uniendo los elementos de de las listas argumento

Ej.: `(APPEND '(a b) '(1 2 3) )` → `(a b 1 2 3)`

3) **(LIST** <obj1> <obj2> ... <objn>**)** devuelve una nueva lista cuyos elementos son los argumentos

Ej.: `(LIST 'a 'b '(1 2 3) )` → `(a b (1 2 3))`

Ejemplo que muestra la diferencia entre las tres:

```
(CONS '(a b) '(1 2 3) )    →    ((a b) 1 2 3)

(APPEND '(a b) '(1 2 3) )    →    (a b 1 2 3)

(LIST '(a b) '(1 2 3) )    →    ((a b) (1 2 3))
```

- Reconocedores de objetos

1) **ATOM** predicado que verifica si su argumento es un átomo

Ej.: `(ATOM 'a )  →  T` `(ATOM '(a b) )  →  NIL`

2) **SYMBOLP** predicado que verifica si su argumento es un átomo no numérico

Ej.: `(SYMBOLP 'a )  →  T` `(SYMBOLP 5 )  →  NIL`

3) **NUMBERP** predicado que verifica si su argumento es un átomo numérico

Ej.: `(NUMBERP 'a )  →  NIL` `(NUMBERP 5 )  →  T`

4) **LISTP** predicado que verifica si su argumento es una lista

Ej.: `(LISTP 'a )  →  NIL` `(LISTP '(a b) )  →  T`

5) **NULL** predicado que verifica si su argumento es la lista nula

Ej.: `(NULL '() )  →  T` `(NULL '(a b) )  →  NIL`

6) **CONSP** predicado que verifica si una lista no es nula

Ej.: `(CONSP '() )  →  NIL` `(CONSP '(a b) )  →  T`

7) **LENGTH** cuenta el número de elementos del nivel superior que hay en la lista

Ej.: `(LENGTH '() )  →  0` `(LENGTH '(a b (X Y)) )  →  3`

- Funciones aritméticas : **+ , - , * , / , expt** (potencia) **, rem** (resto) **, sqrt** (raíz cuadrada)

- Funciones booleanas : **AND , OR , NOT**

- Funciones relacionales: **eq , < , > , <= , >=**

Expresiones condicionales

1) **(IF** <expr1> <expr2> <expr3>**)**

El valor del condicional **IF** es el valor de la <expr2> si el valor de <expr1> es T o bien el valor de la <expr3> si el valor de la <expr1> es NIL.

Ej.: `(IF (> a b) (CAR X) (+ a b) )`

2)
```
(COND    (<expr11>  <expr12>)
         (<expr21>  <expr22>)
                ...
         (<exprn1>  <exprn2>)
)
```

El valor del condicional **COND** será:
el valor de la <expr12> si el valor de la <expr11> es T
el valor de la <expr22> si el valor de la <expr21> es T y <expr11> es NIL
...
el valor de la <exprn2> si el valor de la <exprn1> es T y todas las <exprx1> anteriores son NIL

Ej.:
```
(COND ((> a b) (CAR L) )
      ((AND (EQ a 5) (< b 3)) (CDR L) )
      (T (+ a b) )
)
```

3)
```
(CASE  <expr>
   (<cte1>  <expr1>)
   (<cte2>  <expr2>)
          ...
   [(otherwise  <exprN>)]
)
```

El valor de la <expr> siempre debe ser un átomo y las constantes deben ser átomos o listas.
El valor del **CASE** será:
el valor de la <expr1> si el valor de la <expr> es igual (eq) a la <cte1> o bien el valor de la <expr> pertenece a la lista <cte1>
el valor de la <expr2> si el valor de la <expr> es igual (eq) a la <cte2> o bien el valor de la <expr> pertenece a la lista <cte2> y <expr> no es igual o pertenece a <cte1>
...
el valor de la <exprN> si la última constante es otherwise o bien T

Ej.:
```
(CASE 3
   (1  'a)
   (2  'b)
   (3  'c)
   (t  'z) )    →    c
```

```
Ej.:   (CASE 'hola
          ((uno hola dos) 8)
          ((tres cuatro) 9)
          ((cinco seis siete) 10)
          (otherwise 22)
       )    →    8

Ej.:   (DEFUN PP (N)
          (CASE N
             (1 'PRIMERO)
             (2 'SEGUNDO)
             ((3 4 5) 'POSTERIOR)
          ) )

> (PP 2)    →    SEGUNDO
> (PP 4)    →    POSTERIOR
```

Objetos funcionales

- Funciones definidas por el programador

```
(DEFUN  <nombre>  (parámetros)  <cuerpo>)

Ej.:   (DEFUN Sig (N)
          (+ N 1) )

Ej.:   (DE Fact (N)
          (IF (EQ N 0)
             1
             (* N (Fact (- N 1)))
          )
       )
```

Existe la posibilidad de usar parámetros opcionales con valores por defecto

```
(DEFUN  <nombre>  (<parámetros>  &optional (<parám op> <valor>)...(<parám op>
<valor>))
    <cuerpo>
)

   Ej.:   (DEFUN lista_num (x &optional (z NIL) )
             (COND ((NULL x) z)
                   ((NUMBERP (CAR x)) (lista_num (CDR x) (CONS (CAR x) z)) )
                   (T (lista_num (CDR x) z) )
             )
          )

   > (lista_num '(a b 3 y 8 9 d) )    →    (3 8 9)
```

Aquí el segundo parámetro es opcional, o sea que la función lista_num se puede invocar con uno o con dos argumentos. Si se la invoca con un solo argumento, el valor del segundo parámetro es el valor establecido en la definición o sea que para este caso sería z = NIL

- Funciones de asignación

1) **(SET** 'simb s**)** asigna, destruyendo cualquier asignación previa, el valor de la expresión s al símbolo simb. Devuelve el valor de s.

```
Ej.:   (SET 'X (+ 1 5) )    →    6
       > X    →    6

Ej.:   (SET (IF (= 1 2) 'A 'B) 3 )    →    3
       > A    →    ERROR
       > B    →    3
```

2) **(SETQ** simb1 s1 simb2 s2 ... simbN sN**)** es como **SET**, pero se permiten asignaciones múltiples y no se evalúan los símbolos. Devuelve el valor de sN. **SETQ** viene a ser como **SET QUOTE**.

```
Ej.:   (SETQ X 3 Y (+ X 2) )    →    5
       > Y    →    5
       > X    →    3
```

3) **(MAKUNBOUND** simb**)** borra el valor asignado a simb.

```
Ej.:   (SETQ A '(x y z) )    →    (x y z)
       > A    →    (x y z)
       > (MAKUNBOUND 'A)    →    A
       > A    →    ERROR
```

- Funciones anónimas

((LAMBDA <lista parámetros> <cuerpo>) <valore parámetros>**)** No reciben un nombre, se evalúan solamente cuando se encuentran en el programa y no quedan registradas como funciones en la tabla de símbolos.

```
Ej.:   ((LAMBDA (x) (+ x 1) ) 5 )    →    6

Ej.:   ((LAMBDA (x y) (+ x y) ) 2 3 )    →    5

Ej.:   ((LAMBDA (n) (LIST n n)) 2)    →    (2 2)

Ej.:   ((LAMBDA (X Y) (CONS Y X)) '(B) 'A)    →    (A B)
```

- Formas funcionales

Las formas funcionales son funciones que tienen como parámetro otra/s función/es.

1) **(EVAL** <lista>**)** Devuelve el resultado de evaluar lo que se encuentre en la lista.

```
Ej.:   (EVAL '(+ 2 3) )    →   5

Ej.:   (CONS '+ '(2 3) )    →   (+ 2 3)
       (EVAL (CONS '+ '(2 3)) )    →   5
```

2) **(MAPCAR** <función> <lista>**)** Consiste en aplicarle la función a todos los elementos de la lista obteniendo una nueva lista compuesta por los resultados de esta aplicación en cada elemento.

```
Ej.:   (MAPCAR 'ATOM '(a 1 (1 2)) )    →   (T T NIL)

Ej.:   (MAPCAR '+ '(1 2 3) '(4 5 6))    →   (5 7 9)

Ej.:   (MAPCAR 'CONS '(a (b) c) '((m p) NIL (f)) ) → ((a m p) ((b)) (c f))

Ej.:   (MAPCAR 'CDR '((a b) ((p)) (x (y))) )    →   ((b) NIL ((y)))

Ej.:   (DEFUN sum_vect (v1 v2)          Suma de vectores
          (MAPCAR '+ v1 v2) )

Ej.:   (MAPCAR (LAMBDA (x) (* x 2) ) '(4 5 6) )    →   (8 10 12)
```

Según el interprete de LISP con que se trabaje esta función se la puede encontrar con el nombre de **APPLY-TO-ALL**

3) **(APPLY** <función> <lista>**)** Si la función que se especifica es monádica, la lista debe contener un solo elemento, y el resultado es el valor que devuelve la función aplicada a ese elemento.
Si la función que se especifica es diádica, la lista debe contener solo dos elementos, y el resultado es el valor que devuelve la función aplicada a esos dos elementos, etc...

```
Ej.:   (APPLY 'car '((a b)) )    →   a

Ej.:   (APPLY 'cons '(a (1 2)) )    →   (a 1 2)
```

Ejemplo del uso de **APPLY** dentro de una función:

Vamos a escribir una función que aplicada a una lista dato y una lista que contiene nombres de funciones vaya modificando la lista dato con las funciones de la segunda lista.

(Sería como una especie de robot, al cual le damos la materia prima y una lista de ordenes para que aplique sobre la materia prima)

```
(DEFUN robot (objeto ordenes)
   (IF (NULL ordenes) objeto
      (robot (APPLY  (CAR ordenes) (LIST objeto)) (CDR ordenes) )
   )
)
> (robot '(a (b c) d) '(CDR CAR CDR) )    →   (c)
```

Otra versión que solicita al usuario ingresar las ordenes:

```
(DEFUN robot (objeto ordenes)
   (IF (NULL ordenes) objeto
      (robot (APPLY ordenes (LIST objeto)) (READ))
   )
)
```

```
> (robot '(a (b c) d)  (READ)  )
> CDR
> CAR
> CDR
> NIL
    →   (C)

> (robot '(a (b c) d)  (READ)  )
> (LAMBDA (x) (CONS '1 x))
> (LAMBDA (x) (CONS '2 x))
> (LAMBDA (x) (CONS '3 x))
> NIL
    →   (3 2 1 A (B C) D)
```

4) **(REDUCE** <función> <lista>**)** Es una forma funcional que recibe una función como argumento, en lo posible diádica la función, y la aplica el 1ero con el 2do, el resultado con el 3ero, etc... Reduce la función.

Ej.: `(REDUCE '+ '(1 2 3 4) )  →  ( + ( + (+  1 2) 3) 4)`

Ej.: `(REDUCE 'CONS '(a (b c) ((y))) )  →  ((a b c) (y))`

Ej.: `(REDUCE 'LIST '(a (b c) ((y))) )  →  ((a (b c)) ((y)))`

Ej.: producto escalar

```
(DEFUN prod_escalar (v1 v2)
  (REDUCE '+ (MAPCAR '* v1 v2)) )
```

Lectura y escritura

1) **(PRINT** <expresión>**)** Evalúa su argumento y lo imprime en una nueva línea, seguida de un espacio en blanco y el valor devuelto por el **PRINT** es el valor de su argumento.

```
Ej.:  (PRINT '(a b c) )   →   (a b c)      ;Evaluación de argumento
                              (a b c)      ;valor devuelto por PRINT
```

```
Ej.:   (PRINT “hoy es martes” )     →   “hoy es martes”
                                        “hoy es martes”

Ej.:   (+ (PRINT 6) (PRINT 8) )     →   6
                                        8
                                       14
```

2) **(FORMAT** t <cadena>**)** Imprime la <cadena> en la terminal t y devuelve NIL. En lugar de t puede haber un símbolo que conecte **FORMAT** con un archivo de salida. La <cadena> puede contener caracteres de control. Alguno de ellos son los siguientes:

- ~% nueva línea
- ~D si el argumento es un número decimal
- ~A si el argumento es un carácter ASCII
- ~B si el argumento es un número binario
- ~O si el argumento es un número octal
- ~X si el argumento es un número hexadecimal

```
Ej.:  (FORMAT T “~%LINEA 1 ~%LINEA 2”)     →   LINEA 1
                                               LINEA 2
                                               NIL

Ej.:  (FORMAT T “~%EL CUADRADO DE ~D ES ~D” 3 (* 3 3))
          →       EL CUADRADO DE 3 ES 9
                  NIL

Ej.:  (SETQ L '(A B C))     →       (A B D)
      (FORMAT T “~%LA LONGITUD DE LA LISTA ~A ES ~D” L (LENGTH L))
          →       LA LONGITUD DE LA LISTA '(A B C) ES 3
                  NIL

Ej.:  (FORMAT T “~%10 EN BINARIO ES ~B Y EN OCTAL ES ~O” 10 10)
          →       10 EN BINARIO ES 1010 Y EN OCTAL ES 12
                  NIL

Ej.:  (LIST (FORMAT T “hola “) (FORMAT T “que tal”) )
          →       hola que tal
                  (NIL NIL)

Ej.:  (LIST (PRINT “hola “) (PRINT “que tal”) )
          →       "hola "
                  "que tal"
                  ("hola " "que tal")
```

La forma de imprimir en columnas tabuladas es insertar un número de ancho de columna dentro de la directiva A. Ej. 10A le indica a **FORMAT** que va a alinear la información a izquierda dentro de un espacio de 10 posiciones.

```
Ej.:   (FORMAT T "~5a*~5a=~10a*.*.*" 3 4 (* 3 4) )
          →       3    *4    =12        *.*.*
                  NIL
```

3) **(READ)** para ingreso de datos por teclado. El valor que devuelve **READ** es lo ingresado por teclado

```
Ej.:   (read)
                  10          ;  valor ingresado por teclado
       →          10          ;  valor devuelto por el read

Ej.:   (+ (read) (read) )
                  2
                  3
          →       5

Ej.:   (DEFUN sumavec (&OPTIONAL (v1 (APPEND (FORMAT T "ingrese el 1
                                                vector") (READ)))
                                 (v2 (APPEND (FORMAT T "ingrese el 2
                                                vector") (READ))) )
             (APPEND (FORMAT T "la suma de los 2 vectores es: ") (MAPCAR '+
                                                v1 v2) )
       )
```

ARCHIVOS

Para grabar en un archivo primero hay que abrir el archivo con **OPEN** en modo escritura y luego asociar el stream, que devuelve el **OPEN**, con un nombre utilizando **SETQ**.
Luego se usa la función **PRINT** para grabar los valores en dicho archivo. Al finalizar la grabación de los datos se debe cerrar el archivo con la función **CLOSE**.

Ej.: Las siguientes funciones graban en un archivo los números 1, 2 y 3

```
> (SETQ arch (OPEN "archivo" :direction :output :if-exists
:supersede) )
> (PRINT 1 arch)
> (PRINT 2 arch)
> (PRINT 3 arch)
> (CLOSE arch)
```

Ej.: En el archivo que sigue se graban: la lista (A B (C D E) F G), la cadena "hola que tal" y el átomo A y luego se leen los datos grabados en el archivo

```
> (SETQ arch (OPEN "archivo" :direction :output :if-exists
:supersede))
> (PRINT '(a b (c d e) f g) arch)
> (PRINT "hola que tal" arch)
> (PRINT 'a arch)
> (CLOSE arch)

> (SETQ arch (OPEN "archivo" :direction :input ))
> (READ arch)
→     (A B (C D E) F G)
```

```
        > (READ arch)
→       "hola que tal"
        > (READ arch)
→       A
        > (CLOSE arch)
```

Ayudas

1) **(TRACE** fn1 ... fnN**)** Permite rastrear las funciones fn1,..., fnN mostrando la acción de las funciones fn1,..., fnN cada vez que actúan.

para desviar la salida por pantalla del **TRACE** a un archivo se pone:

```
    (SETQ TRACE-OUT (OPEN "test.txt" :direction :output :if-exists
:supersede )
```

El **TRACE** es acumulativo. Cada vez que invoco **TRACE** con funciones, éstas se agregan a las otras que ya estaban listadas. Para sacar un función puesta en el **TRACE** se pone:

```
    (UNTRACE fn1 ... fnN)
```

Si quiero vaciar la lista del trace pongo : (**TRACE** NIL) o bien (**UNTRACE**)

2) **(STEP** <expresión>**)** evalúa la expresión s paso a paso, de forma que se puede controlar el proceso a voluntad.

```
> (DEFUN FACTORIAL (N)
      (COND ((<= N 1) 1)
      (T (* (FACTORIAL (- N 1)) N))))
          →       FACTORIAL
> (STEP (FACTORIAL 2))
      (FACTORIAL 2)
      2
      2 = 2
      (COND ((<= N 1) 1) (T (* (FACTORIAL #) N)))
      (<= N 1)
      N
      N = 2
      1
      1 = 1
      (<= N 1) = NIL
      T
      T = T
      (* (FACTORIAL (- N 1)) N)
      (FACTORIAL (- N 1))
      (- N 1)
      N
      N = 2
      (- N 1) = 1
      (COND ((<= N 1) 1) (T (* (FACTORIAL #) N)))
      (<= N 1)
      N
      N = 1
```

```
1
1 = 1
(<= N 1) = T
1
1 = 1
(COND ((<= N 1) 1) (T (* (FACTORIAL #) N))) = 1
(FACTORIAL (- N 1)) = 1
N
N = 2
(* (FACTORIAL (- N 1)) N) = 2
(COND ((<= N 1) 1) (T (* (FACTORIAL #) N))) = 2
```

Webgrafía y Licencia

- Textos tomados, corregidos y modificados de diferentes páginas de Internet, tutoriales y documentos.
- Este documento se encuentra bajo Licencia Creative Commons 2.5 Argentina (BY-NC-SA), por la cual se permite su exhibición, distribución, copia y posibilita hacer obras derivadas a partir de la misma, siempre y cuando se cite la autoría del **Prof. Matías E. García** y sólo podrá distribuir la obra derivada resultante bajo una licencia idéntica a ésta.
- Autor:

Matías E. García
Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar