Facultad de Ingeniería
Universidad Nacional de Jujuy

Desarrollo Sistemático de Programas

Unidad 4

Programación Funcional

Ing. Carlos A. Afranllie

Programación Funcional

Historia:

• El primer lenguaje de programación funcional fue LISP, que también fue el segundo lenguaje de alto nivel que se implementó, precedido por Fortran que estaba orientado a cálculos numéricos.

• LISP fue creado por John McCarthy a finales de los 50's en el laboratorio de Inteligencia Artificial del MIT, para un proyecto cuyo propósito era crear “un sistema capaz de manejar oraciones tanto declarativas como imperativas y exhibir un ‘sentido común’ al llevar a cabo las instrucciones”.

• Las listas habían probado ser una estructura de datos adecuada para el procesamiento simbólico, así que McCarthy creó LISt Processing (LISP).

• Se basó en “un esquema para la representación de funciones recursivas parciales de una clase de expresiones simbólicas” que llamó Expresiones-S y las funciones Funciones-S.

Programación Funcional

Historia:

• LISP es un lenguaje muy pequeño, se construye con pocas operaciones básicas y las expresiones condicionales lo que se conoce como "LISP puro", pero también existen instrucciones que permiten la asignación, e iteración.

• LISP fue programado en la computadora IBM 704 que tenían en el MIT, pero después surgieron varios dialectos, lo único que en ese entonces se podría considerar un estándar fue la publicación de Lisp 1.5

• Es un lenguaje (matemáticamente) muy elegante, y es fácil entender los programas escritos en otros dialectos distintos al que uno domine.

• Aplicación: Inteligencia Artificial, ingeniería de software, cálculos ingenieriles, diseño asistido por computadora, manejadores de bases de datos, entre otras.

• Otros lenguajes funcionales son: FP, FFP, KRC, Miranda, ML, SML, Haskell.

LISP (LISt Processing)

Introducción:

• Una característica distintiva de LISP es su poderoso conjunto de primitivas para el procesamiento de listas.

• Existen muchos dialectos nosotros usaremos el estándar definido a mediados de los años 80 denominado Common LISP.

• Este estándar fue completamente definido en 1992, está hoy en día disponible para la mayoría de las plataformas de computación y es de hecho el estándar comercial.

LISP (LISt Processing)

Introducción:

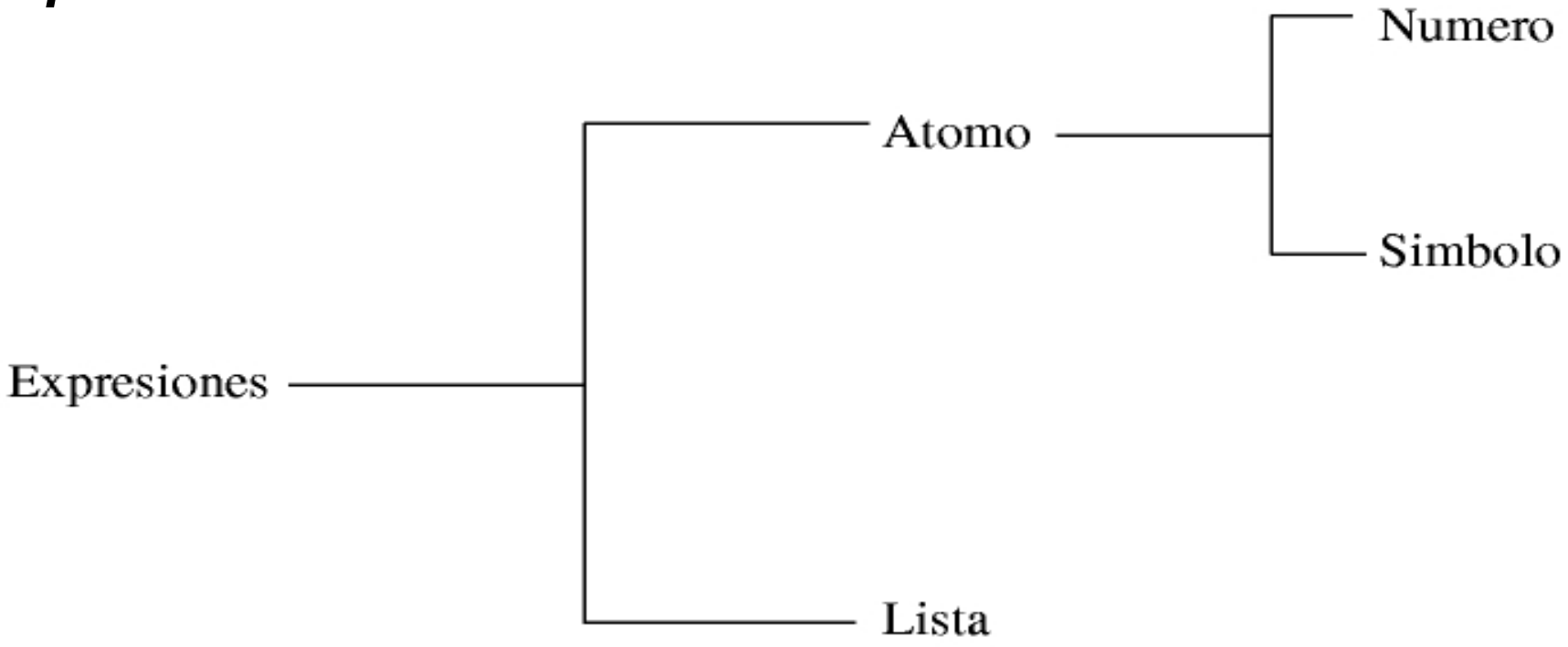
• Equivalencia del formato usado para los datos y los programas; esto permite que las estructuras de datos puedan ser ejecutadas como programas y que los programas puedan ser modificados como datos.

• Los programas no suelen estar estructurados como una secuencia de sentencias, sino como expresiones anidadas en notación prefijo.

• En lugar de la iteración clásica, la estructura de control por excelencia es la recursión.

Expresiones simbólicas – Tipos:

Las expresiones válidas en Lisp se denominan s-expressions o expresiones simbólicas, y de ahora en más las referenciaremos con el término general de expresiones.



Expresiones simbólicas – Tipos:

• Un átomo es un objeto indivisible, como por ejemplo **PEPE**, **B27**, **CUENTA-CORRIENTE**, **18**, **+**, etc.

• Los átomos como **4** y **3.14** se denominan átomos numéricos o simplemente números.

• Los átomos como **PEPE** o **B27** se denominan átomos simbólicos o simplemente símbolos.

Expresiones simbólicas – Tipos:

• Una lista es una expresión formada por la secuencia de 0 o más expresiones entre paréntesis. Ej.: **(+ 1 2 3 4)** y **(2 3 PEPE HOLA)** son 2 listas válidas en LISP de 5 y 4 elementos respectivamente.

• La definición de lista, permite tener listas cuyos elementos sean otras listas, posibilitando de esta manera múltiples niveles de anidamiento. Ej.: **(1 3 (A B) 4)** es una lista de 4 elementos, cuyo tercer elemento es una lista de 2 elementos.

• La lista vacía se denota ( ) o con el átomo NIL.

Evaluación:

• LISP ejecuta normalmente bajo control de un intérprete.

• El usuario ingresa el “programa principal” como una secuencia de expresiones a ser evaluadas.

• El sistema LISP evalúa cada expresión ingresada y muestra el resultado inmediatamente, trabajando en forma análoga a una poderosa calculadora.

Evaluación:

• Cuando el sistema LSIP está á listo para recibir una orden desde el usuario, informa esta situación mostrando un símbolo especial (como por ejemplo un asterisco * o en el caso de la versión LispStudio el símbolo >).

• Para evaluar una operación o función en LISP, la operación debe ser ingresada en forma de lista respetando la notación prefija, es decir, el primer elemento de la lista es la función a ser evaluada y los elementos restantes son sus argumentos.

Evaluación:

> (+ 2 3)	**5**
> (/ 8 4)	**2**
> (+ 2 3 8)	**13**
> (/ 8 4 2)	**1**
> (1+ 5)	**6**
> (MAX 3 8 9 5)	**9**

Evaluación:

Los argumentos de una función u operación, pueden ser no solo átomos sino también otras operaciones, permitiendo así múltiples niveles de anidamiento.

> (+ 3 (* 4 2)) **11**

> (+ (* 2 2) (/ 4 2)) **6**

> (* (+ 1 2) (+ 3 (* 2 2))) **21**

Evaluación:

Cuando LISP evalúa una lista (llamada a función), como en los ejemplos anteriores, realiza normalmente los siguientes pasos:

1. Asume que el primer elemento de la lista es la función a aplicar.
2. Evalúa (recursivamente) cada argumento, de izquierda a derecha.
3. Pasa los valores de los argumentos a la función a aplicar.
4. Ejecuta la función con los argumentos recibidos.
5. Devuelve el valor calculado por la función.

Evaluación:

Si deseamos evaluar un átomo A uno esperaría que así como LISP evalúa (+ 2 3) realizando un cálculo y retornando el resultado de la operación, si tipeamos

> A

me devuelva el resultado de la evaluación de A.

ERROR

Asociación (SETF):

Una forma de “asignar” valores a átomos es utilizando la función primitiva **SETF**. Esta función es similar a una sentencia de asignación en los lenguajes convencionales, y si quisiéramos hacer algo equivalente a la asignación a = 5 deberíamos tipear

```
> (SETF A 5)
5

> A
5
```

Asociación (SETF):

SETF no sólo asigna un valor a un átomo sino que también retorna un valor (como todas las funciones de Lisp). Con SETF se puede asignar a un átomo el resultado de la evaluación de cualquier expresión.

```
>(SETF A (* (+ 1 2) (+ 3 (* 2 2))))

> A
21
```

Asociación (SETF):

Una vez que un átomo tiene un valor asignado puede ser usado como argumento en otras expresiones.

```
> (SETF A 8)

> (SETF B 4)

> (/ A B)

2
```

Asociación:

Existen átomos que no necesitan (y en realidad tampoco se puede) asociarles ningún valor, ya que el resultado de su evaluación es el mismo átomo.

>T	**T**
> NIL	**NIL**
> 2	**2**
> 3.14	**3.14**

Deteniendo la Evaluación (Quote (‘)):

LISP tiene un símbolo especial **' (quote)**, que detiene la evaluación estándar, y su efecto es que la expresión contigua al ', sea retornada textualmente como si fuera una constante.

> (+ 2 3) **5**

> '(+ 2 3) **(+ 2 3)**

> (SETF A (+ 2 3))

> (SETF B '(+ 2 3))

Deteniendo la Evaluación (Quote (‘)):

> A	**5**
> 'A	**A**
> B	**(+ 2 3)**
> 'B	**B**
> (+ A 8)	**13**
> (+ 'A 8)	**ERROR**

Asociación (SET):

Es importante observar que la función SETF maneja sus argumentos en forma no estándar, ya que no evalúa su primer argumento.

Existe en Lisp una función equivalente al SETF, denominada **SET**, pero que trabaja en forma normal.

```
> (SETF X 'Y)
> (SET X 'Z)

> X
Y

> Y
Z
```

Notar que (SETF X 'Z) es equivalente a (SET 'X 'Z)

Evaluación adicional (EVAL):

Así como existe una forma de detener la evaluación usando el ', existe en LISP una función, denominada **EVAL**, que cumple el objetivo inverso, ya que realiza una evaluación adicional.

```
> (SETF A 'B)
> (SETF B 'C)

> 'A                    A

> A                     B
```

Evaluación adicional (EVAL):

> (EVAL A)	**C**
> (EVAL 'A)	**B**
> (SETF X '(+ 2 3))	
> X	**(+ 2 3)**
>(EVAL X)	**5**

Funciones para la manipulación de listas:

> (FIRST '(A B C))	**A**
> (REST '(A B C))	**(B C)**
> (FIRST '((A B) (C D)))	**(A B)**
> (REST '((A B) (C D)))	**((CD))**
> (FIRST (REST '(A B C)))	**B**
> (REST (FIRST (REST '((A B) (C D)))))	**(D)**

Funciones para la manipulación de listas:

> (FIRST (REST (REST '(A B C D))))	**C**
> (FIRST (REST '(REST (A B C D))))	**(A B C D)**
> (FIRST '(REST (REST (A B C D))))	**REST**
> '(FIRST (REST (REST (A B C D))))	
> (SETF A '(A B C))	
> (REST A)	**(B C)**
> (REST 'A)	**ERROR**

Funciones para la manipulación de listas:

> (REST '(A))	**NIL**
> (FIRST ())	**NIL**
> (REST ())	**NIL**
> (THIRD '(A B C D))	**C**
> (FOURTH '(A (B C) D (E F) G))	**(E F)**

LISP- Manipulación de listas

CONS, APPEND y LIST

CONS:

• Toma como argumento una expresión y una lista y retorna una lista cuyo primer elemento es la expresión y los elementos restantes son los elementos de la lista que se pasa como segundo argumento.

• Dicho de otra manera, **CONS** retorna el resultado de agregar el primer argumento a la lista del segundo argumento (en la primera posición).

LISP- Manipulación de listas

CONS:

> (CONS 'A '(B C))

(A B C)

> (CONS '(A G) '(B C))

((A G) B C)

> (SETF LISTA '(1 2 3))

> (CONS 'PEPE LISTA)

(PEPE 1 2 3)

LISP- Manipulación de listas

CONS, APPEND y LIST

APPEND: Realiza la concatenación de las lista pasadas como argumentos.

```
> (APPEND '(A B) '(C D))
```

(A B C D)

```
> (APPEND '(A (B)) '(C D) '((E)) () '(((F))))
```

(A (B) C D (E) ((F)))

LISP- Manipulación de listas

CONS, APPEND y LIST

LIST: Construye una lista cuyos elementos son sus argumentos.

```
> (LIST 'A 'B 'C)
(A B C)

> (LIST 'A '(B C))
(A (B C))
```

LISP- Manipulación de listas

CONS, APPEND y LIST (diferencias):

```
> (SETF LISAB '(A B))
> (SETF LISCD '(C D))

> (CONS LISAB LISCD)
      ((A B) C D)

> (APPEND LISAB LISCD)
      (A B C D)

> (LIST LISAB LISCD)
      ((AB) (CD))
```

LISP- Manipulación de listas

CONS, APPEND y LIST (diferencias):

> (APPEND 'LISAB LISCD)

ERROR

> (CONS 'LISAB LISCD)

(LISAB C D)

> (LIST 'LISAB LISCD)

(LISAB (CD))

LISP- Manipulación de listas

CONS, APPEND y LIST:

• Es importante observar que **CONS, APPEND** y **LIST** no alteran los valores de sus argumentos.

• Si deseamos modificar una lista asignada a un símbolo, estas funciones deberán ser combinadas con el uso del SETF.

LISP- Manipulación de listas

CONS, APPEND y LIST:

```
> (SETF L1 '(1 2 3))

> (CONS 0 L1)
   (0 1 2 3)

> L1
   (1 2 3)

> (SETF L1 (CONS 0 L1))
> L1
   (0 1 2 3)
```

LISP- Manipulación de listas

LAST, LENGTH y REVERSE:

• Uno intuitivamente podría pensar que **LAST** retorna el último elemento de la lista argumento, pero en realidad devuelve una lista con el último elemento.

• **LENGTH** y **REVERSE** por su parte realizan las funciones que sus nombres indican, pero teniendo en mente que cuando la lista argumento tiene múltiples niveles, siempre se trabaja sobre el nivel superior de la lista.

LISP- Manipulación de listas

LAST, LENGTH y REVERSE:

> (SETF LISTA '(A (B C) (D E) F G))

> (LAST LISTA)

(G)

> (LENGTH LISTA)

5

> (REVERSE LISTA)

(G F (D E) (B C) A)

PREDICADOS:

• Un predicado es una función que retorna un valor que representa verdadero o falso.

• El **NIL** siempre indica falso.

• Para indicar que algo es verdadero se suele utilizar el símbolo especial **T**, pero en realidad cualquier cosa distinta de NIL es considerada como verdadero.

LISP- Predicados

EQUAL: chequea sí los valores de sus argumentos son la misma expresión.

```
> (EQUAL 4 (+ 2 2))
   T

> (EQUAL 4 '(+ 2 2))
   NIL

> (EQUAL '(+ 2 2) '(+ 2 2))
   T
```

LISP- Predicados

MEMBER:

• Chequea la pertenencia de un elemento a una lista.

• Si el primer argumento no pertenece al segundo argumento retorna NIL, pero si pertenece no retorna T, sino la lista formada por los elementos del segundo argumento a partir de donde se encontró ó el primer argumento.

• Hay que tener en cuenta que sí tenemos listas de múltiples niveles, el MEMBER sólo trabaja sobre los elementos del nivel superior.

LISP- Predicados

MEMBER:

> (SETF LISTA '(PEPE (TITO TOTA) PEDRO CARLOS RAUL))

> (MEMBER 'JOSE LISTA)
NIL

> (MEMBER 'CARLOS LISTA)
(CARLOS RAUL)

> (MEMBER 'TITO LISTA)
NIL

LISP- Predicados

De control de tipo de expresión:

LISP cuenta con un conjunto de primitivas para controlar si un objeto pertenece a un tipo particular de dato (o tipo de expresión).

Las más importantes se detallan en la siguiente figura:

Nombre	Propósito
ATOM	¿El argumento es un átomo?
NUMBERP	¿El argumento es un número?
SYMBOLP	¿El argumento es un símbolo?
LISTP	¿El argumento es una lista?

LISP- Predicados

```
>(SETF A '(PEPE 1 (TITO)))

> (ATOM A)
   NIL
> (ATOM 'A)
   T
> (NUMBERP (SECOND A))
   T
> (LISTP A)
   T
>(LISTP 'A)
   NIL
```

LISP- Predicados

De control de tipo de expresión:

Un caso muy particular surge de la total equivalencia entre las dos expresiones utilizadas para representar la lista vacía: la lista () y el átomo simbólico NIL.

Ambas expresiones son listas y átomos (simbólicos) al mismo tiempo, por lo que tanto ATOM, como LISTP como SYMBOLP responderán T, cuando son aplicados a cualquiera de las dos expresiones.

LISP- Predicados

De control de tipo de expresión:

Existen dos predicados para chequear si el argumento es una lista vacía: **NULL** y **ENDP**.

La diferencia es que **ENDP** es más restrictiva, ya que **NULL** controla si el argumento es una lista vacía, mientras que **ENDP** controla si el argumento, el cual debe ser una lista, está vacía (no funciona con átomos).

LISP- Predicados

```
> (NULL '(ESTO ES UNA LISTA))
    NIL
 > (ENDP '(ESTO ES UNA LISTA))
    NIL
 > (NULL ())
    T
> (ENDP ())
    T
> (NULL 'A)
    NIL
> (ENDP 'A)
    ERROR
```

LISP- Predicados

Con argumentos numéricos:

LISP cuenta con una amplia variedad de predicados que trabajan sobre números.

Nombre	Propósito
ZEROP	¿El argumento es 0?
PLUSP	¿El argumento es positivo?
MINUSP	¿El argumento es negativo?
EVENP	¿El argumento es par?
ODDP	¿El argumento es impar?
>	¿Los argumentos están en orden descendente estricto?
<	¿Los argumentos están en orden ascendente estricto?

LISP- Predicados

```
> (> 4 2)
    T

> (> 4 4)
    NIL

> (> 2 4)
    NIL

> (> 8 4 2)
    T

> (> 8 2 4)
    NIL
```

LISP- Predicados

Combinando con AND y OR:

• LISP permite combinar predicados usando las conectivas **AND** y **OR**.

• La función **AND** devuelve **NIL** si alguno de sus argumentos evalúa como **NIL**, y devuelve un valor distinto de **NIL** en caso contrario.

• La función **OR** devuelve **NIL** si todos sus argumentos evalúan como **NIL** y devuelve un valor distinto de **NIL** en caso contrario.

LISP- Predicados

Combinando con AND y OR:

• **AND** evalúa sus argumentos de izquierda a derecha y en cuanto uno de sus argumentos evalúa como **NIL**, retorna **NIL** sin evaluar los argumentos restantes. (Evaluación “Perezosa”)

• Si todos los argumentos evalúan valores distintos de **NIL**, el resultado del **AND** es la evaluación del último argumento.

LISP- Predicados

Combinando con AND y OR:

• **OR** también evalúa sus argumentos de izquierda a derecha y retorna el resultado de la evaluación del primer argumento cuya evaluación es distinta de **NIL**, sin evaluar los argumentos restantes. (Evaluación “Perezosa”)

• En caso que la evaluación de todos los argumentos sea **NIL**, el **OR** retorna **NIL**.

LISP- Predicados

> (SETF LISTA '(A B C))

> (AND (EQUAL (LENGTH LISTA) 3) (MEMBER 'B LISTA))

(B C)

> (AND (> (LENGTH LISTA) 3) (MEMBER 'B LISTA))

NIL

LISP- Predicados

> (OR (EQUAL (LENGTH LISTA) 3) (MEMBER 'B LISTA))

T

> (OR (> (LENGTH LISTA) 3) (MEMBER 'B LISTA))

(B C)

> (OR (> (LENGTH LISTA) 3) (MEMBER 'D LISTA))

NIL

LISP- Predicados

Negación NOT:

• La negación en Lisp, es implementada mediante la función **NOT**.

• **NOT** devuelve **T** si su argumento es **NIL**, y **NIL** si su argumento es distinto de **NIL**.

• Observe la total equivalencia entre las funciones **NULL** y **NOT**. Si bien ambas trabajan de la misma manera el uso de una u otra servirá para indicar si uno está pensando en valores lógicos o en listas.

LISP- Predicados

```
> (NOT NIL)
    T

> (NOT T)
    NIL

> (NOT (MEMBER 'B '(A B C)))
    NIL

> (NOT (MEMBER 'D '(A B C)))
    T
```

CONDICIONALES:

• La forma de evaluación estándar de Lisp, establece que todos los argumentos de una función deben ser evaluados.

• Los condicionales son una excepción a esta regla, permitiendo que la evaluación de los argumentos dependa de la evaluación de uno o más predicados.

• El condicional más elemental es la función **IF**, que implementa la selección tradicional.

LISP - Condicionales

IF:

El formato general del **IF** es:

(**IF** <test> <expresión-verdadero> <expresión-falso>)

• Cuando se ejecuta el **IF**, primero se ejecuta el test (un predicado); si el resultado es distinto de **NIL** se evalúa <expresión-verdadero> y en caso contrario se evalúa <expresión-falso>.

• El resultado retornado por la expresión evaluada, será el valor de retorno del **IF**.

LISP- Condicionales

```
> (SETF A 5)

> (IF (AND (> A 3) (< A 8)) A (+ A 1))
   5

>(SETF A 9)

> (IF (AND (> A 3) (< A 8)) A (+ A 1))
   10
```

LISP - Condicionales

COND:

Otro condicional muy utilizado en LISP es el **COND**. Esta función es más general que el IF, ya que permite una evaluación condicional basada sobre múltiples tests.

El formato general del **COND** es:

```
(    COND (<condición 1> <consecuente 1-1> .....)
             (<condición 2> <consecuente 2-1> .....)
         ...
         ...
             (<condición m> <consecuente m-1> .....)
)
```

LISP - Condicionales

COND:

• El símbolo **COND** es seguido por un número variable de listas denominadas cláusulas.

• Cada cláusula contiene una condición seguida por 0 o más expresiones denominadas consecuentes.

• Cuando se ejecuta un **COND**, se evalúan las condiciones de las cláusulas una a una (comenzando por la primera), hasta encontrar una condición cuyo valor retornado sea distinto de **NIL**.

LISP - Condicionales

COND:

• En ese caso, se dice que la cláusula es disparada y todos los consecuentes de la cláusula son evaluados.

• El valor que retorna el **COND** es el valor de la evaluación del último consecuente de la cláusula disparada.

• Si la evaluación de las condiciones de todas las cláusulas es **NIL**, el valor de retorno del **COND** es **NIL**.

LISP- Condicionales

```
> (SETF L1 '(A B C))

> (SETF L2 '(D E F))

> (COND ((AND (ENDP L1) (ENDP L2)) 'AMBAS-VACIAS)
        ((ENDP L1) 'VACIA1)
        ((ENDP L2) 'VACIA2)
        (T (SETF SUMA (+ (LENGTH L1) (LENGTH L2))) SUMA)
)    6
```

Definición de Funciones DEFUN:

La primitiva **DEFUN** nos permite definir nuestras propias funciones, tiene el siguiente formato general:

(**DEFUN** <nombre de función> (<parámetro 1> <parámetro n>)
<expresión 1>
...
…
<expresión m>
)

El nombre de la función debe ser un átomo simbólico.

LISP – Definición de Funciones

DEFUN:

Al igual que cualquier otra función, **DEFUN** también retorna un valor, que en esta caso será el nombre de la nueva función definida.

Si por ejemplo, quisiéramos definir una función que, dado un argumento numérico, devuelva el doble del argumento, podríamos hacer:

```
> (DEFUN DOBLE (N)
        (* 2 N)
)
DOBLE
```

LISP – Definición de Funciones

```
> (DOBLE 4)
  8

> (SETF A (DOBLE 9))

> A
  18
```

LISP – Definición de Funciones

DEFUN:

1. Se reserva memoria para contener el resultado de la evaluación de los parámetros reales.

2. Se ligan los parámetros formales a éstos lugares de memoria (salvando previamente los viejos valores si fuera necesario).

3. Se evalúan las expresiones del cuerpo de la función y se retorna como resultado la evaluación de la última expresión.

4. Libera la memoria reservada para los parámetros.

5. Restaura los viejos valores de los parámetros formales si fuera necesario.

LISP – Definición de Funciones

DEFUN:

• Es importante destacar, que LISP crea una especie de "defensa virtual" de los parámetros formales definidos en una función.

• Esto significa que si uno de los símbolos utilizados como parámetro formal, ha tenido otro uso previo, LISP preserva estos valores, manteniéndolos inalterados al finalizar la ejecución de la función (inclusive en el caso en que el parámetro formal hubiera sido alterado explícitamente con un SETF dentro de la función).

LISP – Definición de Funciones

```
> (DEFUN DOBLE (N)
        (* 2 N)
)

>(SETF N 6)

> (DOBLE 4)
   8

> N
   6
```

LISP – Definición de Funciones

DEFUN - Ejemplo:

Tomar 2 listas y obtener una nueva lista de 2 elementos cuyo primer elemento es el primero de la primera lista incrementado en 1 y el segundo es el último de la segunda lista como está.

Este efecto, lo podríamos lograr haciendo directamente:

```
>(SETF L1 '(1 2 3))

>(SETF L2 '(4 5 6 7))

> (CONS (1+ (FIRST L1)) (LAST L2))
(2 7)
```

LISP – Definición de Funciones

```
> (DEFUN FUN1 (LISTA1 LISTA2)
        (CONS (1+ (FIRST LISTA)) (LAST LISTA2))
)
```

FUN1

```
> (FUN1 L1 L2)
```

(2 7)

```
> (FUN1 L2 L1)
```

(5 3)

```
> (FUN1 '(8 6) L2)
```

(9 7)

LISP – Definición de Funciones

Otra forma de escribir la función FUN1 podría ser la siguiente:

```
> (DEFUN FUN1 (LISTA1 LISTA2)
        (SETF TEMP1 (1+ (FIRST LISTA)))
        (SETF TEMP2 (LAST LISTA2))
        (CONS TEMP1 TEMP2)
)
   FUN1
```

LISP – Repetición

Recursión:

Una definición de función recursiva consta normalmente de:

- La utilización de una o más copias de la función que está siendo definida, que toman como argumentos una versión simplificada del problema original.

- Uno o más "puntos de parada", para aquellos argumentos en los que el valor de la función es conocido, y el resultado es directo y no implica nuevas invocaciones recursivas de la función.

LISP – Recursión

Un ejemplo clásico de este tipo de problemas, es el cálculo del factorial (!) de un número, que se puede definir recursivamente de la siguiente manera:

factorial(0) = 1

factorial(N) = N * (factorial (N - 1)); si N > 0

El factorial puede ser implementado sencillamente en Lisp, mediante la siguiente definición de función recursiva:

LISP – Recursión

```
 > (DEFUN FACTORIAL (N)
        (IF (ZEROP N)
              1
              (* N (FACTORIAL (1- N)))
)

FACTORIAL

> (FACTORIAL 0)
1

> (FACTORIAL 3)
6
```

LISP – Recursión

Ej.: Implementar una función MIEMBRO que trabaje como la primitiva MEMBER de Lisp.

Esta función tendrá 2 puntos de parada: el primero controlará si el segundo argumento es la lista vacía, lo cual acontecerá cuando la función ha sido aplicada directamente a NIL, o se ha terminado de recorrer completamente la lista y no se ha encontrado al primer argumento. En este caso la función debe devolver NIL.

El segundo punto de parada, es cuando el primer argumento es igual al primer elemento del segundo argumento, en cuyo caso la función debe devolver el segundo argumento
como está.

Cuando ninguno de estos 2 casos ocurra, la función MIEMBRO retornará el mismo valor que MIEMBRO aplicado a la cola de la lista del segundo argumento.

LISP – Recursión

```
> (DEFUN MIEMBRO (ELEMENTO LISTA)
        (COND ((ENDP LISTA) NIL)
        ((EQUAL ELEMENTO (FIRST LISTA)) LISTA)
        (T (MIEMBRO ELEMENTO (REST LISTA)))
        )
)
MIEMBRO

> (MIEMBRO 3 ())
NIL

> (MIEMBRO 3 '(1 2 4))
NIL

> (MIEMBRO 3 '(1 2 3 4))
(3 4)
```

LISP – Recursión

Uno puede ver a una lista como una estructura formada por su primer elemento, y la cola, la cual a su vez es una lista.

Esta idea puede ser utilizada cuando debemos definir una función que realice algún tipo de procesamiento sobre todos los elementos de una lista.

En este caso, es común definir la función recursivamente de manera tal de procesar el primer elemento, e invocar la función recursivamente sobre la lista restante, y así sucesivamente hasta llegar al final de la cola (NIL).

LISP – Recursión

Ej.: definir una función que incrementa en 1 todas las componentes de una lista de números.

```
>(DEFUN INCREMENTAR1 (LISTA)
     (IF (ENDP LISTA)
          NIL
          (CONS (1+ (FIRST LISTA)) (INCREMENTAR1 (REST LISTA));
    )
)
```

INCREMENTAR1

```
> (INCREMENTAR1 '(1 3 6 8))
```

(2 4 7 9)

LISP – Recursión

Cuando se trabaja con expresiones anidadas (listas de múltiples niveles), la situación se complica ya que al tomar el FIRST de la lista, este elemento puede ser una lista que debe a su vez será procesada recursivamente.

Es por este motivo, que cuando se deben procesar expresiones anidadas en Lisp, usualmente la función es doblemente recursiva ya que la función se debe llamar a sí mismo dos veces: una vez para analizar el primer elemento y otra para analizar los elementos restantes.

Como se puede ver, esta forma de descomponer y procesar la lista conducirá eventualmente a que la función sea aplicada a la lista vacía o a un átomo, siendo éstos los puntos de parada más usuales en estas situaciones.

LISP – Recursión

```
> (DEFUN CONTAR-CEROS (EXPRESION)
        (COND ((NULL EXPRESION) 0)
              ((EQUAL EXPRESION 0) 1)
              ((ATOM EXPRESION) 0)
              (T (+ (CONTAR-CEROS (FIRST EXPRESION))
                    (CONTAR-CEROS (REST EXPRESION))))
        )
)
CONTAR-CEROS

> (CONTAR-CEROS 0)
1

> (CONTAR-CEROS 'A)
0

> (CONTAR-CEROS '((0) 3 0 (((0 A)))))
3
```

LISP – Repetición

Iteración:

Otra estructura de control que permite repetir una acción sin utilizar la recursión es la iteración.
Lisp provee una primitiva denominada **MAPCAR** que permite repetir una acción sobre todos los elementos de una lista en forma iterativa (sin usar la recursión).

La función MAPCAR, tiene el siguiente formato general:

(MAPCAR #'<nombre-de-función> <lista>)

LISP – Iteración

La función MAPCAR toma como primer argumento un nombre de función (o para ser más precisos un objeto función y una lista, y retorna como resultado una lista cuyos elementos son los resultados de aplicar la función con cada elemento de la lista como argumento.

> (MAPCAR #'ODDP '(1 2 3))

(T NIL T)

LISP – Iteración

Usando MAPCAR, la función INCREMENTAR1 se podría haber definido en forma iterativa de la siguiente manera:

```
> (DEFUN INCREMENTAR1 (LISTA)
(MAPCAR #'1+ LISTA)
)
```

INCREMENTAR1

LISP – Iteración

MAPCAR puede tomar como argumento una función de más de un argumento.

En este caso, el MAPCAR deberá ser invocado con tantas listas como argumentos requiera la función, y el MAPCAR será el encargado de aplicar la función del primer argumento, suministrándole los argumentos requeridos, tomando uno de cada lista argumento.

> (MAPCAR #'EQUAL '(1 2 3) '(3 2 1))
(NIL T NIL)

LISP – Iteración

Otra dos primitivas importantes que también usan una función como argumento son **FUNCALL** y **APPLY**.

FUNCALL aplica su primer argumento (una función) a los argumentos restantes.

El formato general del FUNCALL, se presenta a continuación:

(FUNCALL #'<nombre de función> <argumento 1> ... <argumento n>)

LISP – Iteración

Algunos ejemplos del uso del FUNCALL son los siguientes:

```
> (FUNCALL #'LENGTH '(1 2 3))
3

> (FUNCALL #'APPEND '(1 2 3) '(A B))
(1 2 3 A B)
```

Una forma fácil de entender el efecto del uso del FUNCALL es considerar que el resultado es el mismo que si tapáramos la palabra FUNCALL con el dedo.

LISP – Iteración

FUNCALL es realmente útil cuando debemos definir nuestros propios procedimientos que toman funciones como argumentos.

Consideremos la siguiente función:

```
> (DEFUN POSICION (FUNCION LISTA)
(FUNCALL FUNCION LISTA)
)
```

POSICION

LISP – Iteración

> (POSICION #'FIRST '(A B C))
A

> (POSICION #'THIRD '(A B C))
C

La función POSICION retorna el elemento del segundo argumento que se obtiene aplicando la función del primer argumento. Si en la definición de POSICION, hubiéramos intentado llamar directamente a (FUNCION LISTA), Lisp hubiera indicado un error, dado que la función FUNCION no está definida.

LISP – Iteración

La primitiva APPLY por su parte, también toma una función como primer argumento pero su segundo argumento es una lista.

Su efecto es el mismo que si invocáramos la función no con la lista, sino con los elementos de la lista.

El formato general del APPLY, se presenta a continuación:

(APPLY #'<nombre de función> <lista de argumentos>)

LISP – Iteración

Para observar la necesidad del APPLY consideremos el siguiente ejemplo:

```
> (+ '(3 5 7))
ERROR

> (APPLY #'+ '(3 5 7))
15
```

LISP – Repetición

Combinando MAPCAR y recursión:

```
> (DEFUN CONTAR-CEROS (EXPRESION)
        (COND ((NULL EXPRESION) 0)
                ((EQUAL EXPRESION 0) 1)
                ((ATOM EXPRESION) 0)
                (T (APPLY #'+ (MAPCAR #'CONTAR-CEROS EXPRESION)))
        )
)
```