Paradigmas y Lenguajes

Msc. Ricardo Monzón

PROGRAMACION FUNCIONAL
INTRODUCCION. LISP

INTRODUCCION A LA PROGRAMACION FUNCIONAL - LISP

- Lisp fué propuesto por John McCarthy a finales de los 50.
- Es un lenguaje que implementa el modelo de las funciones recursivas las cuales proporcionan una definición sintáctica y semánticamente clara.
- Las **listas** son la base tanto de los programas como de los datos en LISP. Programas y Datos son equivalentes.
- LISP: es un acrónimo de LISt Processing.
- Lisp proporciona un conjunto potente de funciones que manipulan listas, implementadas internamente como punteros.
- Originalmente fue un lenguaje simple y pequeño, consistente en funciones de construcción y acceso a listas, definición de nuevas funciones, pocos predicados para detectar igualdades y funciones para evaluar llamadas a funciones.

INTRODUCCION A LA PROGRAMACION FUNCIONAL - LISP

- Los principales fundamentos de control son la recursión y las condicionales simples.
- Todo programa LISP consiste de una función.
- No posee asignaciones.
- La principal estructura de datos es la Lista.
- Muchos de los programas desarrollados dentro del ámbito de la Inteligencia Artificial se implementaron en LISP.
- Muchos entornos modernos para la creación de sistemas expertos están implementados en Lisp.

INTRODUCCION A LA PROGRAMACION FUNCIONAL - LISP

Características de Lisp (Vamos a trabajar sobre el Common Lisp**)**

Portabilidad: Excluye aquellas características que no puedan ser implementadas en la mayoría de las máquinas. Se diseñó para que fuera fácil construir programas lo menos dependiente posible de las máquinas.

Consistencia: muchas implementaciones LISP son internamente inconsistentes en el sentido de que el intérprete y el compilador pueden asignar distintas semánticas al mismo programa. Esta diferencia radica fundamentalmente en el hecho de que el interprete considera que todas las variables tienen alcance dinámico.

INTRODUCCION A LA PROGRAMACION FUNCIONAL - LISP

Expresividad: recoge las construcciones más útiles y comprensibles de dialectos Lisp anteriores.

Eficiencia: tiene muchas características diseñadas para facilitar la producción de código compilado de alta calidad.

Potencia: suministradas por multitud de paquetes que corren sobre el núcleo.

Estabilidad: se pretende que el corazón cambie lentamente y solo cuando el grupo de expertos encargados del estándar así lo decidan por mutuo acuerdo después de examinar y experimentar con las nuevas características

Expresiones LISP

- En Lisp todo son expresiones evaluables.
- La regla de oro de Lisp (lenguaje funcional) es la evaluación de sus expresiones. Existen diferentes reglas de evaluación dependiendo del tipo de expresión.
- Lisp siempre trabaja internamente con apuntadores, cada atributo NO contiene el dato sino un apuntador al mismo.
- Lisp en general, y a menos que se diga lo contrario, convertirá automáticamente las minúsculas en mayúsculas.

Tipos de Valores

Los tipos de *valores* que existen en LISP:

- el tipo **atom** (atómicos) como un número, un carácter, un símbolo, un tipo string.
- el tipo **cons**, como una lista.

Adicionalmente, tenemos

- una **estructura** definida por el usuario.
- tipos de datos **propios** del usuario.

Es totalmente dinámico (depende del momento en el que se hace la asignación).
En un lenguaje convencional es necesario declarar el tipo del contenido. En Lisp no es necesario hacerlo, aunque lo permite.

TIPO ATOM (1)

Los **átomos** son estructuras básicas en Lisp. Es un elemento indivisible que tiene significado propio.
Algunos ejemplos son:

- Los **números**: Lisp evalúa un número devolviendo ese mismo número. Un número se apunta a sí mismo. En general un átomo se apunta a sí mismo.
  - Enteros: **… -2, -1, 0, 1, 2, …**
    - Representan los enteros matemáticos.
    - No existe límite en la magnitud de un entero.
  - Racionales: **-2/3, 4/6, 20/2, 5465764576/764**
    - numerador / denominador.
  - Coma flotante: **0.0, -0.5**
  - Complejos: **#C(5 -3), #C(5/3 7.0) = #C(1.66666 7.0).**

TIPO ATOM (2)

- Los **caracteres** y **strings** son dos tipos de datos utilizados normalmente para manipular texto. En general, la función de evaluación recae sobre sí mismos.
  - **Caracteres:** **#\a, #\A**
    - son objetos que comienzan por: **#**
    - #\a es evaluado como #\a
  - **Caracteres especiales NO IMPRIMIBLES:**
    - **#\SPACE, #\NEWLINE**
  - **Strings:** **"A", "hola"**
    - Se pueden considerar como series o como vectores de caracteres. Pueden examinarse para determinar qué caracteres lo componen. La representación imprimible de un string comienza con " y finaliza por ".

TIPO ATOM (3)

Átomos Simbólicos. Un átomo simbólico es un átomo que representa algo más, como el nombre de una función o el valor de una variable.

- **Los nombres de variables**. Todo nombre de variable debe ser un símbolo atómico. El nombre imprimible se usa como el nombre de la variable y el tipo de valor asociado puede ser un átomo o una construcción tipo lista. Los nombres de variables permitidos son:

  A, CONT, VALOR-FINAL, UNA-LISTA, ENT45

La evaluación de una **VARIABLE** es más complicada que la de los átomos simples (enteros, fraccionarios, coma flotante, complejos, caracteres y string) vistos anteriormente.

En general, LISP intentará aplicar la regla de devolver el valor de la variable.

TIPO ATOM (4)

Los nombres de símbolos no distinguen entre mayúsculas y minúsculas, y pueden ser cualquier secuencia de caracteres de notación y alfanuméricos a excepción de los siguientes: ( ) . ' " ; . Un nombre de símbolo no puede estar compuesto sólo por caracteres numéricos ya que un número se representa a sí mismo y es por tanto una *constante*. Lo mismo sucede con los nombres entrecomillados.

- **Constantes especiales**. Existen dos constantes especiales que son símbolos reservados y corresponden con,

T	**true**
NIL	**false o lista vacía ()**

Las constantes especiales apuntan a ellas mismas.

TIPO CONS (1)

En Lisp hay muchas formas de agrupar elementos; por ejemplo los strings que ya han sido vistos. Sin embargo, la forma más importante y versátil de agrupar objetos es mediante ***listas***.

(*mesa silla lámpara estanterías*)

En general una LISTA es una estructura básica de datos y una estructura de un programa en Lisp. Una lista está delimitada por los paréntesis de abrir y cerrar (), y en su interior está formada por una secuencia de átomos y/o listas o cualquier combinación de ambas.

Además, una lista puede tener cualquier clase y número de elementos, separados entre sí por un espacio, incluso no tener elementos (NIL o ()).

El símbolo NIL o () representa la Lista vacía.

TIPO CONS – Ejemplos de Listas (2)

(1 2 3 4)	*; lista de enteros*
(a b c d)	*; lista de variables*
(#\a #\b #\c #\d)	*; lista de caracteres*
(4 algo de "si")	*; lista con entero, variables y string.*
(sqrt 2)	*; lista con una función*
(+ 1 3)	*"*
(a b (c d) e)	*; lista con átomos y sublistas*
NIL o ()	*; lista vacía*

Todos los programas en Lisp se describen mediante funciones que se definen y llaman con LISTAS.

TIPO CONS (3)

*Un **CONS** es una estructura de información que contiene dos componentes llamados el **CAR** y el **CDR**. La utilización fundamental de los CONSES es como representación de LISTAS.*

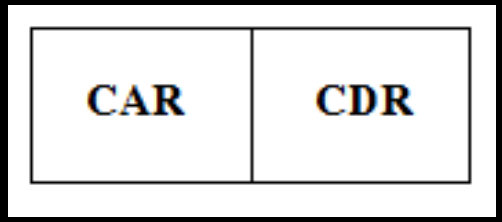
Una LISTA se define recursivamente ya sea como la lista vacía o como un CONS cuyo componente CDR es una lista.

Una lista es, por consiguiente, una cadena de CONSES enlazados por sus componentes CDR y terminada por un NIL, la lista vacía.

Los componentes CAR de los conses son conocidos como los elementos de la lista. Para cada elemento de la lista hay un CONS. La lista vacía no tiene ningún elemento.

TIPO CONS (4)

La estructura utilizada para representar la secuencia de elementos de una lista es la estructura CONS. Cada elemento de una lista se representa a través de una estructura CONS formada por dos partes, **CAR** y **CDR**.



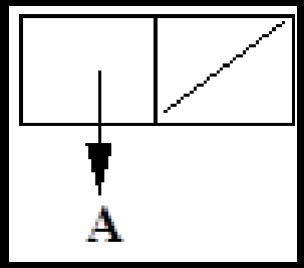
El CAR es la primera parte de la estructura y contiene un apuntador al primer elemento de la lista. El CDR es la segunda parte y contiene un apuntador a la siguiente estructura CONS que contiene los siguientes elementos de la lista o si no hay más elementos un apuntador a NIL. Una lista es unidireccional. Por ejemplo:

- El átomo **A** no es una estructura CONS.

Algunas LISTAS y su representación con ESTRUCTURAS CONS

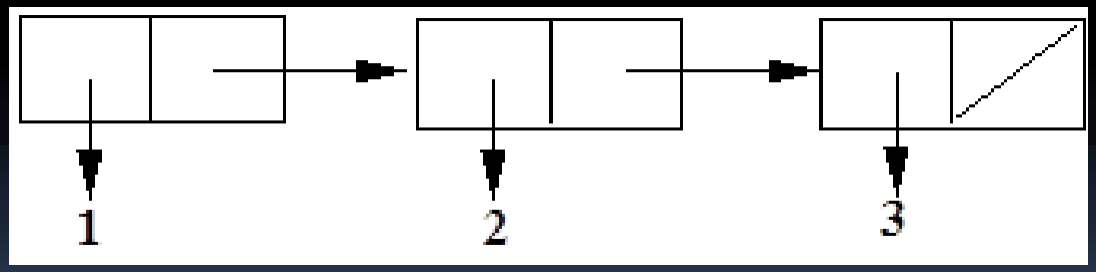
- () NO TIENE UNA ESTRUCTURA CONS

- (A)



Lista con un elemento A.

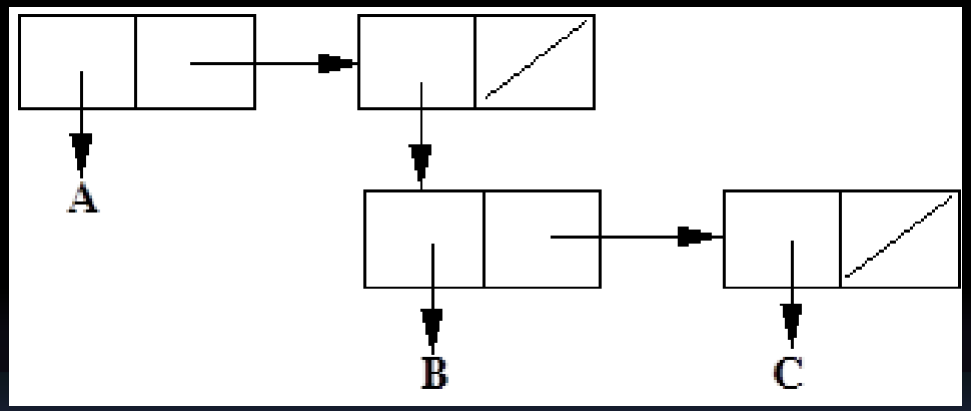
- (1 2 3)



Lista con tres enteros.

Algunas LISTAS y su representación con ESTRUCTURAS CONS

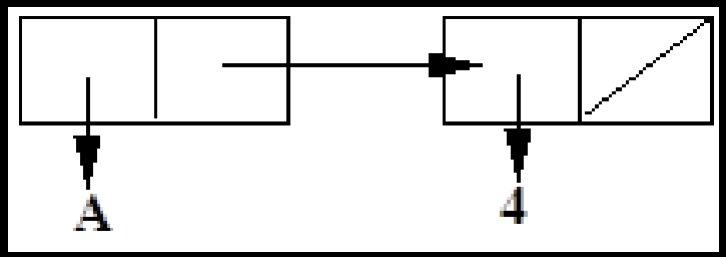
- (A (B C)) Lista con un elemento y una sublista de dos elementos.



Listas Punteadas

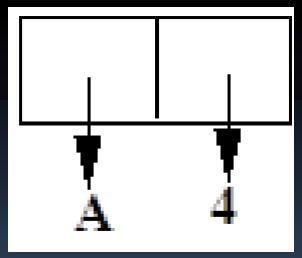
Otra forma de expresar listas es mediante ***LISTAS PUNTEADAS*** (dotted list). **Cuando el CDR de una estructura CONS apunta a un átomo en lugar de a una estructura CONS o NIL, la lista formada es una lista punteada.** Por ejemplo,

- Lista (A 4)



Lista común

- Lista punteada (A . 4)



Las listas punteadas a menudo son llamadas también *pares punteados*.

Listas Punteadas

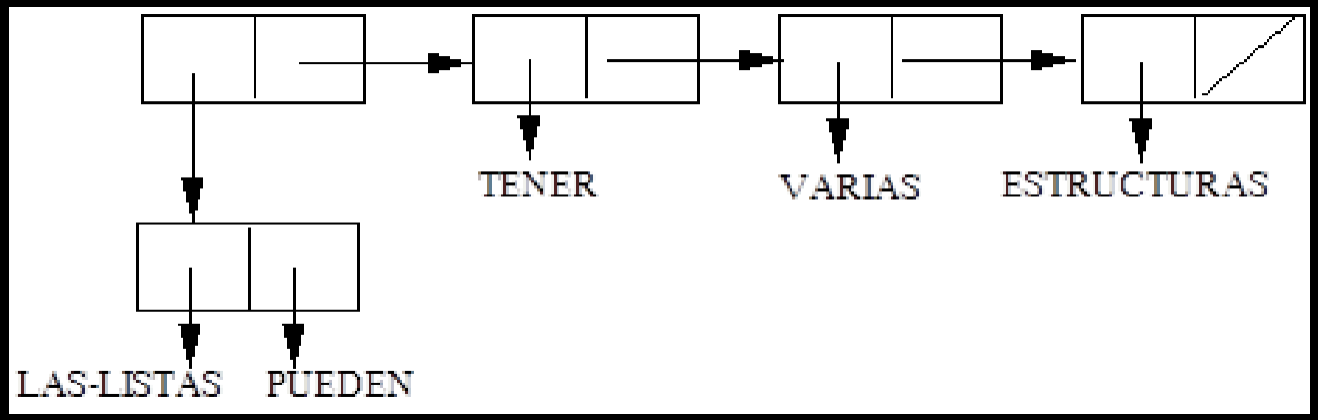
Cuando el último elemento de una lista no apunta a NIL, se genera una lista punteada. La ventaja de las listas punteadas es el ahorro de una casilla CONS. Sin embargo, se pierde en flexibilidad ya que el tratamiento de una lista no podrá depender de la marca de fin de lista. Además en una lista , que apunta como último elemento a NIL, permite la modificación de la misma añadiendo nuevas casillas CONS. Sin embargo una lista punteada no permite estas modificaciones ya que se eliminaría el último elemento de la lista punteada. Un ejemplo erróneo:

(LU MA MI JU VI SEMANA SABADO . DOMINGO FIN-DE-SEMANA)

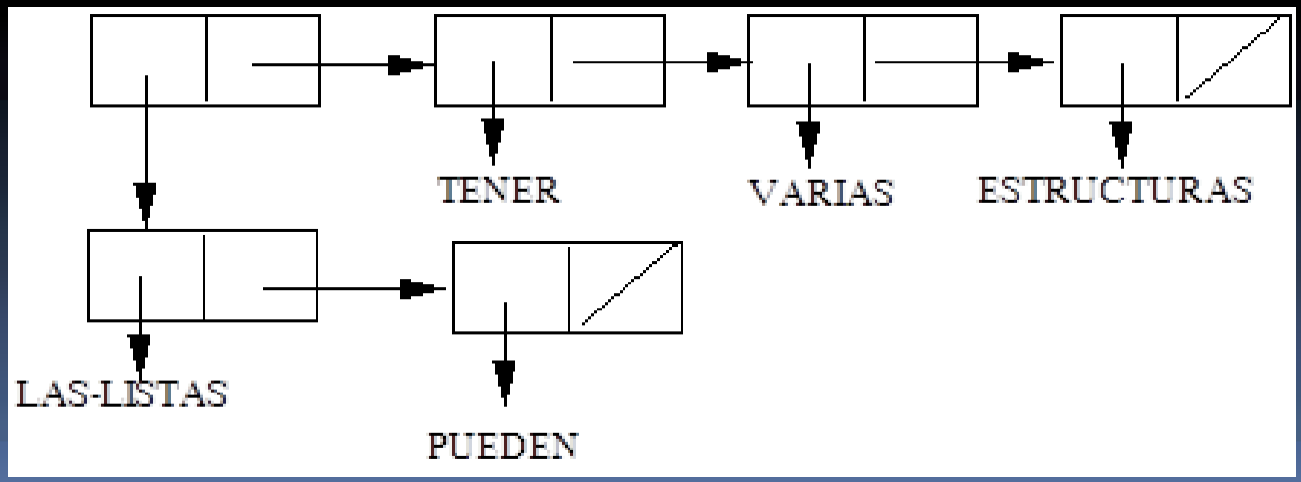
Esta lista no es sintácticamente correcta. Una lista punteada precisa que después del punto aparezca un objeto simple y después un paréntesis.

Ejemplos

- ((LAS-LISTAS . PUEDEN) TENER VARIAS ESTRUCTURAS)



- ((LAS-LISTAS PUEDEN) TENER VARIAS ESTRUCTURAS)



Evaluación en LISP – Lisp Listener

El ***Lisp Listener*** es el ciclo evaluador que realiza continuamente Lisp, siempre está a la espera de entradas. Cuando recibe una entrada, evalúa la expresión, espera al return, si la forma es atómica o espera a paréntesis de cierre si es una lista. Finalmente imprime el resultado y el ciclo comienza de nuevo. A este ciclo se llama ciclo **READ-EVAL-PRINT** expresado a través de las funciones Lisp **(PRINT (EVAL (READ)))**.

Por tanto, el ***evaluador*** es el mecanismo que ejecuta las formas y los programas LISP. Lisp siempre tratará de evaluar todo lo que se le suministre; esto es realizado a través de la función *eval;* **(eval form)** se evalúa la forma y se devuelve el valor. Así los **números**, **caracteres** y **strings** tienen reglas de evaluación simples cuya evaluación devuelve su propio valor.

Ejemplos

si se escribe el **3**, Lisp devuelve el **3**
si se escribe **"esto es un string"**, Lisp devuelve **"esto es un string".**

Viéndolo desde un intérprete Lisp, siendo el “prompt” del mismo **>**:

```
> 3 <return>          ;; necesario pulsar return, indicando el fin
3
> “esto es un string” <return>
“esto es un string”
> A <return>
Error: The variable A is unbound.
> T
T
> NIL
NIL
> (quote(+ 2 3))                    ;; la función quote se ve mas adelante
(+ 2 3)
> (eval quote(+ 2 3))
>5
```

Evaluación de expresiones

Básicamente, el intérprete de LISP evalúa una expresión *S*, representando posiblemente aplicaciones anidadas de funciones, de la siguiente forma:

Si ***S* es un átomo numérico *a***, entonces el resultado es simplemente dicho átomo numérico *a*.

Si ***S* es de la forma (f a1 a2 a3 … an)**, entonces el intérprete evalúa recursivamente cada argumento ai, obteniendo valores v1, v2, …, vn y luego calcula el resultado de aplicar f a dichos valores.

Ej.

```
> (+ 1 2 3 (* 3 4))  ;; f es la función suma 1, 2, 3 y (* 3 4) argumentos
18
```

FUNCION QUOTE o ‘

Como dijimos anteriormente, cuando LISP evalúa una expresión, evalúa recursivamente cada argumento de la misma.

Existen situaciones en las que resulta fundamental evitar la evaluación de ciertos argumentos en una aplicación. Por ejemplo, cuando dichos argumentos son listas, pero que no denotan la aplicación de una función sino simplemente una lista de elementos como estructura de datos.

La función **quote** provoca que un objeto NO sea evaluado. Esta será la forma de poder pasar una lista como datos.

La forma de escribirlo es **(QUOTE OBJETO)** o bien, **‘OBJETO**.

> (quote X)
X
> ‘X
X

Ejemplos

Cualquier entrada en Lisp tendrá una salida imprimible. Por ejemplo, las expresiones con quote devuelven siempre su propio valor.

> A <return>

Error: The variable A is unbound.

> 'A *<return>*

A

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

(+ 2 3)

>

Una forma de lista (sin quote) espera al cierre del paréntesis:

> (+ 2 3)

5

> (* (+ 2 3) (/ 12 4))

15

FUNCIONES PARA OPERAR CON NUMEROS
Funciones Aritméticas Básicas

Las formas de las expresiones matemáticas son muy familiares para nosotros. Por ejemplo, + suma, - resta, * multiplicación, / división, < menor que, >= mayor o igual que, SQRT raíz cuadrada, etc.

Cada una de las funciones mencionadas seguidamente, **requiere que sus argumentos sean todos números**. El pasarle un argumento no numérico provocará un error. Operan tanto sobre números enteros como sobre números reales, realizando los ajustes pertinentes cuando dichos argumentos fueran de diferente tipo.

Todas devuelven un resultado decimal cuando algún argumento no es entero.

Ninguna de estas funciones modifica el valor de ningún argumento, solo devuelven un resultado.

Funciones Aritméticas Básicas (1)

+ : devuelve la suma de todos los argumentos que se pasan.

(+ [arg arg arg] ...)

Si proporciona sólo un argumento número, esta función devuelve el resultado de sumarlo a cero. Ningún argumento, devuelve 0.

```
> (+ 1 2 3)
6
> (+ 1.0 2 3)
6.0
```

- **:** devuelve el resultado de todas las restas sucesivas de los argumentos.

(- [arg arg arg] ...)

Si utiliza más de dos argumentos número, esta función devuelve el resultado de restar del primer número la suma de todos los números desde el segundo hasta el último. Si sólo utiliza un argumento número, la función devuelve el valor resultante de restar el número a cero. Ningún argumento, devuelve error.

```
> (- 10 1 2 3)
4
> (- 10 1 2.0 3)
4.0
```

Funciones Aritméticas Básicas (2)

***** : devuelve el producto de todos los argumentos dados.

(* [arg arg arg] ...)

Si proporciona sólo un argumento número, esta función devuelve el resultado de multiplicarlo por uno. Ningún argumento, devuelve 1.

```
> (* 1 2 3)
6
```

```
> (* 1 2 3.0)
6.0
```

/ : devuelve como resultado un valor obtenido de la división del primer elemento con todos los demás.

(/ [arg arg arg] ...)

Si utiliza más de dos argumentos, esta función divide el primer número por el producto de todos los números del segundo al último y devuelve el cociente final. Si proporciona sólo un argumento número, devuelve el resultado de dividirlo por uno. Ningún argumento, devuelve error.

```
> (/ 30 2 4)
3
```

```
> (/ 30 2.0 4)
3.75
```

Mas Funciones Aritméticas. (2)

Puede obtenerse un entero como resultado de las funciones **TRUNCATE y ROUND.**

Truncate: Trunca una expresión tendiendo hacia 0.

Formato: **(truncate <expr>)**

Round: Redondea hacia el entero positiva mas cercano.

Formato: **(round <expr>)**

En los dos casos, <expr> es una expresión real sobre la cual se aplicará la función.

Las dos operaciones, entregan dos resultados, el primero con el resultado de la operación en sí misma, y el segundo con el remanente de la operación.

Ejemplos.

```
> (truncate 3.14)
3
0.14000000000000012

> (round 3.123)
3
0.12300000000000022

> (round 3.723)
4
-0.27700000000000014
```

Mas Funciones Aritméticas. (3)

- **FLOAT**, convierte un entero en un numero de coma flotante.

(float expr) **> (float 8)** **==>** **8.0**

- **RATIONAL**, convierte un numero real en racional.

(rational expr) **> (rational 2.5)** **==>** **5/2**

- **REM**, **MOD** : devuelve el remanente de la división de dos números.

(rem expr expr) **> (rem 7 2)** **==>** **1**

- **ABS**, devuelve el valor absoluto de una expresión.

(abs expr) **> (abs -8)** **==>** **8**

- **SIGNUM**, permite obtener el signo de una expresión. Devuelve 0 para el 0, 1 para los positivos y -1 para los negativos.

(signum expr) **> (singnum -8)** **==>** **-1**

Mas Funciones Aritméticas. (4)

- MAXIMO Devuelve el mayor de una lista de numeros.
  (MAX NUM NUM …) **>> (MAX 2 5 3 1)==> 5**

- MINIMO Devuelve el menor de una lista de números.
  (MIN NUM NUM …) **>> (MIN 2 5 3 1)==> 1**

- Máximo Común Divisor de una lista de números.
  (GCD NUM NUM …) **>> (GCD 12 34 56)==> 2**
  Si no recibe argumentos, devuelve 0. Si tiene un solo argumento, devuelve el mismo argumento.

- Mínimo Común Múltiplo de una lista de números.
  (LCM NUM NUM …) **>> (LCM 12 34 56)==> 2856**
  Si no recibe argumentos, devuelve error. Si tiene un solo argumento, devuelve el mismo argumento. Si el resultado es mayor que el limite de los enteros, devuelve un error.

Funciones Matemáticas. (1)

- **SQRT**, raíz cuadrada de un número.

(SQRT NUMBER)

Por ejemplo,

>> (SQRT 16) ==> 4.0
>> (SQRT 2) ==> 1.414…
>> (SQRT -4) ==> #c(0.0 2,0)

- **EXPT**, exponencial. Calcula el valor de un número elevado a otro número.

(EXPT NUMBER NUMBER)

Por ejemplo,

>> (EXPT 2 3) ==> 8
>> (EXPT 3 -2) ==> 0,11111111
>> (EXPT 3 0) ==> 1

Hasta aquí, todas las funciones vistas, no modifican el valor de los argumentos pasados como parámetros.

Funciones Matemáticas. (2)

- **INCF**, incrementa en una cantidad DELTA un valor **Var**, por defecto es 1.

(INCF Var [DELTA])

Var debe ser una variable. **Suponemos que C vale 5**, entonces

> (INCF C 10) **==> 15**
> C **==> 15**
> (INCF C) **==> 16**
> C **==> 16**

Mientras que si asumo que el valor de C es 5, y

> (- 1 C) **==> 4**
> C **==> 5**

- **DECF**, decrementa en una cantidad DELTA, una variable Var, por defecto es 1.

(DECF Var [DELTA])

> (DECF C 10) **==> -5**
> C **==> -5**
> (DECF C) **==> -6**
> C **==> -6**

Como se ve en los ejemplos, estas funciones modifican el valor de los argumentos pasados como parámetros.

Predicados de comparación de números.

Los predicados, son **FUNCIONES booleanas**, que devuelven solo los valores **T o NIL**.

- IGUALDAD **(= NUM NUM …)** **>> (= 3 3.0) ==> T**
- NO IGUAL **(/= NUM NUM…)** **>> (/= 3 3.0) ==> NIL**
- MENOR QUE, secuencia estrictamente creciente de números.
  (< NUM NUM …) **>> (< 3 5 8) ==> T**
  >> (< 3 5 4) ==> NIL
- MAYOR QUE, secuencia estrictamente decreciente de números.
  (> NUM NUM …) **>> (> 8 5 3) ==> T**
  >> (< 8 3 5) ==> NIL
- MENOR O IGUAL QUE, secuencia NO estrictamente creciente de núm eros.
  (<= NUM NUM …) **>> (<= 5 5 8) ==> T**
  >> (<= 9 9 4) ==> NIL
- MAYOR O IGUAL QUE, secuencia NO estrictamente decreciente de números.
  (>= NUM NUM …) **>> (>= 9 7 7) ==> T**
  >> (>= 8 9 8) ==> NIL

Predicados Numéricos.

Son predicados que se utilian para verificar exclusivamente argumentos numéricos. Aceptan 1 solo argumento.

- **NUMBERP**, verifica si el tipo de objeto es numérico. Devuelve true si el objeto es un número. El argumento puede ser de cualquier tipo.

  (NUMBERP OBJETO) **>> (NUMBERP 7) ==> T**
  >> (NUMBERP 'NOMBRE) ==> NIL

- **ODDP**, verifica un argumento, que debe ser entero, y devuelve cierto si el entero es impar.

  (ODDP ENTERO) **>> (ODDP -7) ==> T**
  >> (ODDP 5.8) ==> NIL

- **EVENP**, verifica un argumento, que debe ser entero, y devuelve cierto si el entero es par.

  (EVENP ENTERO) **>> (EVENP 8) ==> T**
  >> (EVENP 'NOMBRE) ==> ERROR

Predicados Numéricos.

- **INTEGERP**, verifica si el argumento es un munero entero. Devuelve true si el argumento es un entero. El argumento puede ser de cualquier tipo.

  (INTEGERP OBJETO)

  >> (INTEGERP 7) ==> T

  >> (INTEGERP 'NOMBRE) ==> NIL

- **ZEROP**, verifica un argumento, que debe ser numérico, y devuelve cierto si el número es cero.

  (ZEROP NUMBER)

  >> (ZEROP 0.0) ==> T

  >> (ZEROP 'NOMBRE) ==> ERROR

Consideraciones.

- Los términos de los operadores matemáticos pueden ser: solamente numéricos.
- Al ser funciones, un término de la operación matemática puede ser el resultado de otra función matemática.
- Si todos los términos de un operador matemático son átomos el resultado es un Átomo.
- Si existe al menos un término del operador matemático que es una Lista, el resultado es una Lista con la estructura de la lista de mayor profundidad.
- Si existe más de un término del operador matemático que es una Lista, éstas deben ser de la misma longitud en el 1er. Nivel, independientemente de la profundidad.
- La operación se ejecuta de izquierda a derecha, término a término y dentro de cada "término a término": calcula elemento a elemento si los hubiera.

Ejemplos. (1)

CARACTERÍSTICAS DE LOS TÉRMINOS	EJEMPLO:	SALIDA:
Átomos	> (+ 1 -2 3.5)	2.5
	> (- 1 '-2 3.5)	-0.5
	> (* 1 '-2 3.5)	-7.0
	> (/ 1 -2 3.5)	-0.14285714285714285
Listas de = long. en el 1er. nivel	> (+ '(1 -2 3.5) '(4 5 6))	(5 3 9.5)
	> (+ '(1 -2 3.5) '(4 5 6) '((2 4) (3) (((5)2))))	((7 9) (6) (((14.5) 11.5)))
	> (- '(1 -2 3.5) '(4 5 6))	(-3 -7 -2.5)
	> (- '(1 2 3.5) '(4 5 6) '((2 4) (3) (((5)2))))	((-5 -7) (-10) (((-7.5) -4.5)))
	> (* '(1 -2 3.5) '(4 5 6))	(4 -10 21.0)
	> (* '(1 2 3.5) '(4 5 6) '((2 4) (3) (((5)2))))	((8 16) (30) (((105.0) 42.0)))
	> (/ '(1 -2 3.5) '(4 5 6))	(0.25 -0.4 0.583)

Ejemplos. (2)

Listas de ≠ long. en el 1er. nivel	**> (+ '(1 2 3) '(4 5))**	**Error: arguments not all the same length**
Átomos y Listas	> (+ 1 '(2 -3 4) 5)	**(8 3 10)**
	> (+ 1 '(2 -3 4) 5 ‘(4 7))	**Error: arguments not all the same length**
	> (+ 1 '(2 -3 4) 5 ‘(4 (7) ((2))))	**(12 (10) ((12)))**
	> (- 1 '(2 -3 4) 5)	**(-6 -1 -8)**
	> (- 1 '(2 -3 4) 5 '(4 (7) ((2))))	**(-10 (-8) ((-10)))**
	> (* 1 '(2 -3 4) 5)	**(10 -15 20)**
	> (* 1 '(2 -3 4) 5 '(4 (7) ((2))))	**(40 (-105) ((40)))**
	> (/ 1 '(2 -3 4) 5)	**(0.1 -0.066 0.05)**

Ejemplos. (3)

CARACTERÍSTICAS DE LOS ARGUMENTOS			
Argumento1	**Argumento2**	**EJEMPLO:**	**SALIDA:**
número	Número	> (expt 3 2)	**9**
número	Función	> (expt 3 (- 6 4))	**9**
función	Función	> (expt (/ 21 7) (- 6 4))	**9**
número	Lista	> (expt 2 '(1 2 3))	**(2 4 8)**
Lista	Número	> (expt '(1 2 3) 2)	**(1 4 9)**
Lista	Lista	> (expt '(1 2 3) '(3 3 3))	**(1 8 27)**
Lista	Lista	> (expt '(3 3 3)) '(1 2 3))	**(3 9 27)**
Lista	**Lista**	**> (expt '(3 3) '(1 2 3))**	**Error: arguments not all the same length Happened in: #<Subr-EXPT: #1d66a9c> > (expt '(3 3 3) '(1 2))**